Software Architecture Discovery for Testability, Performance, and Maintainability of Industrial Systems

Dharmalingam Ganesan  2012
Software Architecture Discovery for Testability, Performance, and Maintainability of Industrial Systems

door
Dharmalingam Ganesan
geboren te Madurai, India
promotor: prof. dr. C. Verhoef

copromotoren: dr. R.L. Krikhaar
dr. M. Lindvall
Acknowledgements

My research work spans across the Atlantic Ocean. I was fortunate to work at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Germany from January 2003 to December 2007. It is my honor to thank Prof. Dr. Dieter Rombach, the founder of Fraunhofer IESE, for nurturing a world-class institute for applied research. I relocated from Fraunhofer IESE to Fraunhofer Center for Experimental Software Engineering (CESE) in Maryland, and started working for CESE in January 2008. It is my honor to thank Prof. Dr. Victor Basili, who was one of the founders of CESE, for many fruitful discussions. My special thanks to Prof. Dr. Rance Cleaveland, the Executive Scientific Director of CESE, for creating an excellent work environment and encouraging my thesis work. Mr. Frank Herman, the Managing Director of CESE, offered me several opportunities to work for NASA projects under his guidance. Frank always likes to share his deep 35+ years of technical expertise with me.

I would like to thank Prof. Dr. Chris Verhoef, my supervisor at the VU University Amsterdam, The Netherlands, for accepting me as a PhD candidate. Chris’s expertise on reverse engineering of real-world systems helped me to position the thesis that we believe is relevant to industrial practitioners as well researchers. I enjoyed the fact that Chris has versatile skills including IT economics, software architecture, source code analysis, regular expressions, UNIX commands, Vector Spaces, and other mathematically-based formal methods, too. Chris’s constructive comments on earlier versions of the thesis were precise to the point and very useful to improve the quality of the thesis. Chris and I had several productive Skype sessions for discussing the technical content of the thesis. Working with Chris has been a great pleasure.

This thesis would not have been possible if I had not seen the PhD thesis of Dr. Rene Krikhaar, on the bookshelf of Mr. Eric van Utteren, during my visit to Philips Research Laboratories, The Netherlands, in February 2001, when I was employed at Philips Bangalore, India. Eric, then the director of Philips Information and Software Technology department, introduced me to Rene, who gave me a copy of his practically inspired PhD thesis during our dinner meeting. I thank Eric for introducing Rene to me. Rene reviewed several earlier versions of this thesis and gave me constructive comments to improve the thesis. Rene helped me in translating the summary of the thesis to the Dutch language. Rene’s expertise on Relation Partition Algebra (RPA) helped me in utilizing it on several projects successfully. Rene and I had several productive Skype sessions to package the thesis. I am delighted and overjoyed that my meeting in 2001 lead to a PhD thesis with Rene as the co-supervisor of the thesis\(^1\).

During my work at Fraunhofer, I had the opportunity to analyze software architectures of more than two dozen industrial systems. This work eventually resulted in this thesis,

\(^1\) http://www.cs.vu.nl/~rkrikhaa/adam/ is the homepage of the project: Architecture Discovery and Analysis Method (ADAM). Within the ADAM project this thesis and its publications are deliverables. I thank Rene for creating this web page.
Acknowledgements

which could not have been written without the full support of my manager, Dr. Mikael Lindvall, who is the Director of the Software Architecture and Embedded Systems division at Fraunhofer CESE. Without Mikael’s tireless support of carefully reviewing, probing, and editing the technical papers that I wrote with him, it would have been difficult for me to reach this point. Mikael believed in me and offered several opportunities to work directly with customers. Mikael and I worked until midnight for several days when we were analyzing the NASA Space Network project, made of several programming languages and multimillion lines of code. We are passionate about our work and even talked about software architectures at the Greenbelt city tennis court and in our local grocery shops. Hope no else overheard our technical discussions!

During my work at Fraunhofer IESE in Germany, I worked under the guidance of Dr. Dirk Muthig in the department of product line architectures (PLA). Dirk introduced me to the world of PLA, and assigned me as the technical leader of several industrial projects. I was fortunate to receive a research excellence award for architecture reconstruction to support migration to a software product line for Hitachi, Japan, in year 2006.

I am honored to have the thesis reviewed and approved by an international reading committee that consists of scholars: Prof. Dr. Serge Demeyer (University of Antwerp, Belgium), Prof. Dr. Loe Feijs (Eindhoven University of Technology, The Netherlands), Prof. Dr. Wan Fokkink (VU University Amsterdam, The Netherlands), and Prof. Dr. Rick Kazman (University of Hawaii and Software Engineering Institute – Carnegie Mellon University (SEI-CMU), USA).

I thank the anonymous reviewers of technical papers that form the origin of the thesis’ chapters. Your comments were invaluable for improving the quality of the thesis. I enjoyed technical discussions of the DiscoTect method with Bradley Schmerl at CMU in the context of Chapter 4. Prof. Dr. Wil van der Aalst and the Exspect team at Eindhoven University of Technology, The Netherlands, kindly answered my questions related to the Exspect Colored Petri net (CP-net) tool. Dr. Dominic Widdows, Software Engineer at Google, is the author of “Geometry and Meaning” book, which connects Natural Language Processing with fundamental Geometry and Linear Algebra in a classical way. Dominic, I thank you for several e-mail discussions on Vector Spaces and Random Projection models.

Dr. Pieter van der Spek, Ms. Elly Lammers, Ms. Mieke Noordam, and Ms. Nicoline van Os patiently answered innumerable questions about the rules and regulations of a doctoral thesis at VU Amsterdam. Elly, you were extremely helpful during my visa application to the Netherlands Embassy.

I thank all my colleagues including Ms. Lyly Yonkwa, Dr. Arnab Ray and Ms. Myrna Regardie for discussing and reviewing the thesis. Dr. Thorsten Keuler helped me in collecting runtime traces using AspectJ discussed in Chapter 4. In some projects, I used the visualization capability of the SAVE tool to visualize the module dependency and organizational data. Dr. Jens Knodel and Ms. Lyly Yonkwa kindly offered their support for the SAVE tool. Lyly, it was a lot of fun to chat and play tennis with you. I thank my office manager, Ms. Mary Berry, for her encouragement and help on several occasions. Ms. Sonnhild Namingha helped me in correcting the English grammar of many of the papers I wrote during my employment at Fraunhofer IESE, Germany. I thank Dr.
Jaejoon Lee for his great sense of humor and discussions on software product lines during our employment at Fraunhofer IESE, Germany.

Rene mentioned in his thesis that research on software architectures can indeed only be performed in cooperation with people who actually build large systems. I fully agree and am fortunate that I was able to cooperate with engineers from several organizations. Without their full cooperation it would be next to impossible to carry out my work. I am also fortunate that several of our collaborators allowed me to write technical papers based on the analysis I did using their systems. Despite their regular project schedule commitments, they took the time to carefully review and edit the papers that I wrote together with them. I thank our following collaborators for their valuable input to this thesis.

The NASA IV & V Center: Ms. Lisa Montgomery and her team for supporting the research collaboration between Fraunhofer and NASA projects under the Office of Safety and Mission Assurance (OSMA), Software Assurance Research Program (SARP).

The NASA Goddard Space Flight Center (GSFC): Ms. Sara Godfrey for supporting the research collaboration between Fraunhofer and NASA GSFC, and introducing us to several GSFC missions.

The NASA Core Flight Software (CFS) team: Mr. David McComas, Mr. Maureen Bartholomew, Mr. Stevel Slegel, Ms. Barbara Medina, Mr. Kequan Lu, Mr. Alan Kudmore, and Mr. Charles Wildermann. The CFS is the subject of Chapter 2 and Chapter 3.

The Ricoh Japan team: Mr. Yutaro Nishimura, Mr. Masanori Kagino, and Mr. Hideharu Teranishi. Ricoh’s Multi-Function Peripherals (MFPs) is the subject of Chapter 4.

The NASA Goddard Mission Services Evolution Center (GMSEC) team: Mr. Lamont Ruley, Mr. Robert Wiegand, Mr. Vuong Ly, Ms. Tina Tsui, and Ms. Sharon Orsborne. The GMSEC is the subject of Chapter 5.

The NASA Space Network Access System (SNAS) team: Ms. Monica Ron and Mr. Dave Warren of Honeywell. The SNAS is the subject of Chapter 6.

The US Food and Drug Administration (FDA) team: Mr. Paul Jones, Dr. Raoul Jetley, and Dr. Yi Zhang. I also thank Alford R Taylor, Director Division of Electrical and Software Engineering at the FDA, for encouraging our architectural analysis research. The architectural analysis of the CARA medical device software was performed at the FDA campus. The CARA is the subject of Chapter 7.

The US Walter Reed Army Institute of Research team: Mr. Steve van Albert, Dr. Fred Pearce, and Mr. Jaime Lee for making the CARA medical device software available for our research at the FDA. The CARA is the subject of Chapter 7.

The Hitachi Japan team: Dr. Kentaro Yoshimura and Engine Control System (ECS) engineers and managers of the automotive business unit. The ECS is the subject of Chapter 8.
In addition to the above list of collaborators, during these years, I also worked closely with several other engineers from several organizations. Many of the systems that were analyzed are briefly summarized in Appendix A. I thank the following collaborators for their input to my research:

NASA White Sands Complex: Mr. Markland Benson, for taking part in the architectural analysis of the NASA’s Space Network project.

Temperature Sensitive Solutions (TSS) Sweden: Dr. Niclas Ohlsson, Mr. Magnus Jäderberg, and Mr. Robert Eriksson for taking part in the architectural analysis of the Cold Chain Information System (CCIS).

Biofortis Maryland: Dr. Jian Wang, Mr. Steve Chen, and Mr. Mark Brocato for taking part in the architectural analysis of feasibility of integrating open source components (Cranefoot and Madeline) into Biofortis’s medical information system.

Cranefoot Finland: Mr. Ville-petteri Makinen, the developer of the open source Pedigree (i.e., a family tree) software component, for kindly answering my questions. I am delighted that Ville has distributed the results of my architecture reconstruction effort as part of his open source code bundle.

Madeline Tool: Mr. Ed Trager, one of the lead developers of the open source Pedigree software component Madeline, for kindly reading my architecture reconstruction report of the Madeline software component.

GNSI Maryland: Mr. Pino Marinelli and Mr. Jangho Ki for taking part in the architectural analysis of GNSI’s medical information system.

Archivex Maryland: Dr. Steve Bova for taking part in the architectural analysis of the Archivex’s medical information system.

MarketMaker Software AG Germany: Dr. Martin Verlag and Ulrich Maurer for taking part in the architectural analysis of MarketMaker’s web-based financial product family.

Testo AG Germany: Mr. Gerald Meier for taking part in the architectural analysis of the Testo’s climate and temperature measurement product family.

Bueren and Partner Software Design Germany: Mr. Guenter Bueren and Mr. Christian Rueckert for taking part in the architectural analysis of several of your partners’ embedded systems.

Students: It was my privilege to supervise several students who performed internship or masters’ thesis at Fraunhofer. In particular, I thank Mr. Joerg Weinmann for developing the first version of the Dynamic-SAVE tool, Mr. Oliver Lehr for developing the first version of the CP-net tool, Mr. Jens Christ, Mr. Andreas Krause, and Christof Velte for evolving the CP-net tool and carrying out some aspects of the future work mentioned in Chapter 2, Mr. Felix Haeussler for developing the first version of the SAVE-LIGHT tool, Mr. Karl Andres Gislason, Mr. Jochen Kamuf for implementing several useful features to the SAVE-LIGHT tool, Mr. Palmi Valgeirsson, and Mr. Gunnar Cortes for helping me in extracting the return codes of functions using lightweight pattern matching. I thank all of you for contributing to the tools discussed in Chapter 10. Without your contribution, we could not have made progress in applying our approach to our collaborators’ systems. I was fortunate to teach parts of this thesis to the students
of the Professional Master’s program in Software Engineering at the University of Maryland, College Park. The attendees of this program, who were mostly professional engineers, had shown great interest on the topic of this thesis.

I thank all my friends including Jeanne Carriere Goshorn, Dan Clarkson, Kimberly Goshorn Clarkson, Jessamyn Strebs, Marilyne Dos Reis, and the family of Mikael Lindvall: Chatarina, Jenny, Malin, and Victor for many wonderful moments at Greenbelt, Maryland.

Last, but not least, I thank my family members for their full support throughout my carrier. Without my parents’ commitment, dedication, and determination, I cannot imagine what is presumably considered the highest academic degree. My brother was always willing to share his experience in real-time embedded software development. My brother and sister regularly encouraged me in all these years. I am indebted to my late grandparents.

Dharmalingam Ganesan

October 22, 2011
List of Figures

Figure 1-1: Model of Action Research Process [4]. .......................................................... 15
Figure 2-1: The NASA CFS Heritage. .............................................................................. 24
Figure 2-2: High-level structure of the CFS product line. .............................................. 25
Figure 2-3: CFS business goals and architecture decisions. ........................................ 25
Figure 2-4: The CFS example context.............................................................................. 26
Figure 2-5: Four steps in architectural rules verification.................................................. 27
Figure 2-6: Actual dependencies from the app layer to the core. ................................. 30
Figure 2-7: Dependencies among application modules................................................... 31
Figure 2-8: Dependencies on the software bus............................................................... 31
Figure 2-9: Bypassing the OS abstraction layer............................................................... 32
Figure 2-10: The wrapper function to exit a task under the VxWorks OS. ................... 34
Figure 2-11: The wrapper function to exit a task under the RTEMS OS. ..................... 35
Figure 2-12: Directory structure decomposition - a violation. ...................................... 36
Figure 2-13: Left: Planned template. Right: Violation. .................................................... 36
Figure 2-14: Conditional preprocessor directives.......................................................... 39
Figure 3-1: The modular structure of the CFS Product Line. ........................................ 45
Figure 3-2: The context diagram of the software bus in the CFS.................................. 46
Figure 3-3: Two major steps in the analysis of unit test architecture............................... 49
Figure 3-4: Dependencies between core modules......................................................... 50
Figure 3-5: The “big picture” of unit tests...................................................................... 51
Figure 3-6: Structure of unit tests for the ES module...................................................... 51
Figure 3-7: Interface specification of the create pipe function........................................ 52
Figure 3-8: The stub implementation of the create pipe function................................. 52
Figure 3-9: The key data structure for controlling return values................................... 53
Figure 3-10: Forcing the create pipe function to return CFE_ERROR............................. 53
Figure 3-11: Forcing a function to return success and error in a row............................ 53
Figure 3-12: Code snippet for receiving messages from the software bus..................... 54
Figure 3-13: A stub to generate different return codes in a row.................................. 55
Figure 3-14: LOC of test functions of the ES module.................................................... 60
Figure 3-15: Calling the same function two times.................................................................61
Figure 3-16: Hook function returning success and error in a row. .................................62
Figure 3-17: Redirecting function calls to the hook function for testing. .....................62
Figure 3-18: Programming to abstract interfaces. ...............................................................66
Figure 3-19: Code fragment for loading a library at runtime. .............................................67
Figure 3-20: Testing the upper bound of a configuration parameter. ..............................68
Figure 3-21: Test uses the internal data structure CFE_SB.PipeTbl.................................68
Figure 3-22: The same return code is used for different error types...............................69
Figure 3-23: Test code for function shown in Figure 3-22. .................................................70
Figure 4-1: The DiscoTect Architecture (figure from [251]). ............................................79
Figure 4-2: Steps of the approach.......................................................................................81
Figure 4-3: OSGi framework - Bundle concept...................................................................83
Figure 4-4: Creating an “Intermediate class loader”.............................................................84
Figure 4-5: Overview of the weaving process......................................................................85
Figure 4-6: Aspect definition with a concrete pointcut.......................................................86
Figure 4-7: Aspect definition using an abstract pointcut...................................................87
Figure 4-8: Merged meta-model of the approach.................................................................88
Figure 4-9: Execution view on CP-nets................................................................................91
Figure 4-10: The approach in action.....................................................................................91
Figure 4-11: CP-net for initializing a pipe............................................................................94
Figure 4-12: Hierarchical CP-nets for extraction of the P-F style.......................................95
Figure 4-13: The CP-net for filter creation (Rule_CreateFilter).........................................96
Figure 4-14: The CP-net for Rule_GetFilterChildren.........................................................97
Figure 4-15: The CP-net for Rule_CreateReadPort..............................................................97
Figure 4-16: The CP-net for Rule_AttachFilter.................................................................97
Figure 4-17: The CP-net for P-F (Rule_CreatePipeAndFilter)...........................................98
Figure 4-18: The CP-net for Rule_Filter_2_Filter_Call.......................................................99
Figure 4-19: The root level CP-net......................................................................................99
Figure 4-20: A sample extracted C-C view........................................................................100
Figure 4-21: A sample extracted message sequence view...............................................101
Figure 5-1: Typical elements of a software bus.................................................................113
Figure 5-2: Conceptual elements of dynamic analysis .............................................. 118
Figure 5-3: Dependencies to middleware and socket libraries. .......................... 120
Figure 5-4: The setup for dynamic analysis of the NASA GMSEC. .................. 122
Figure 5-5: An example C-C view discovered using runtime events. ................. 123
Figure 5-6: A sequence diagram with timing information.................................. 124
Figure 5-7: GMSEC business goals and architectural decisions. ....................... 127
Figure 6-1: The RPA operators used in this chapter........................................... 141
Figure 6-2: RPA query for abstracting dependencies between packages. .......... 142
Figure 6-3: Query for lifting the similarity relation to the package level. .......... 143
Figure 6-4: RPA query to identify all classes required for each process. .......... 144
Figure 6-5: RPA query to identify packages required for each process. .......... 144
Figure 6-6: Queries for partitioning modules into components....................... 145
Figure 6-7: Abstracted relations from components to queues. .......................... 146
Figure 6-8: RPA queries to identify server-side runtime components............... 146
Figure 6-9: RPA queries to extract the tasks synchronization view. ................. 147
Figure 6-10: RPA queries to extract processes to a database usage............... 148
Figure 6-11: Conceptual dimensions of the ADAM.......................................... 150
Figure 6-12: The module view of the SNAS..................................................... 152
Figure 6-13: (Snippet) Summary generated using the knowledge base. .......... 153
Figure 6-14: Analysis questions for discovering the IPC structure.................. 153
Figure 6-15: Discovered Server-side sockets. ................................................. 154
Figure 6-16: Discovered Client-Side sockets. ................................................. 154
Figure 6-17: Discovered Runtime View.......................................................... 155
Figure 6-18: Dependencies on database tables. ............................................. 160
Figure 6-19: The Data Access Objects (DAOs) of snif. .................................. 161
Figure 6-20: DB interactions in the snif subsystem. ....................................... 161
Figure 6-21: The abstraction of the DAO layer. ............................................. 163
Figure 6-22: Methods of the snif DB manager class...................................... 164
Figure 6-23: Design for testing without accessing databases. ........................ 165
Figure 6-24: Using dummy data for testing.................................................... 165
Figure 6-25: Creating a database connection pool for performance.............. 166
List of Figures

ix

Figure 6-26: Logic for a DB connection pool................................................................. 166
Figure 6-27: DB connection in the sdif subsystem......................................................... 167
Figure 6-28: Singleton design pattern for database connections. .............................. 168
Figure 6-29: Database abstraction in the sdif subsystem................................................. 169
Figure 6-30: DB interaction steps in the sdif subsystem................................................. 169
Figure 6-31: Low testability and constructors................................................................. 170
Figure 6-32: A snippet of the knowledge base for Java GUI libraries.......................... 171
Figure 6-33: A performance risk due to the threading model....................................... 171
Figure 6-34: Low testability and the lack of a GUI abstraction...................................... 173
Figure 6-35: Proposed refactoring of Figure 6-34.......................................................... 174
Figure 6-36: IP address validation without a GUI............................................................ 174
Figure 6-37: (Snippet) knowledge base for analyses of the OS concern......................... 175
Figure 6-38: A OS abstraction issue.................................................................................. 175
Figure 7-1: Reverse Architecting Process...................................................................... 189
Figure 7-2: Summary of the CARA using the knowledge base.................................... 190
Figure 7-3: (Snippet) Static structure of OS variants...................................................... 192
Figure 7-4: Wrapper view of DSCUD COTS abstraction............................................... 194
Figure 7-5: (Snippet) Call graph of COTS interaction................................................... 194
Figure 7-6: Avoiding hardware interaction for testing................................................... 195
Figure 7-7: VxWorks syntax to send messages on a queue........................................... 196
Figure 7-8: (Snippet) VxWorks syntax to create a task.................................................. 197
Figure 7-9: Definition of RPA notations.......................................................................... 198
Figure 7-10: RPA queries for derived relations of Table 7-6.......................................... 199
Figure 7-11: (Snippet) Task creation structure (i.e., $TT_{create}$). .................................. 199
Figure 7-12: (Snippet) Partition of files among tasks.................................................... 200
Figure 7-13: Tasks and their entry files of execution...................................................... 201
Figure 7-14: (Snippet) Inter-task communication structure............................................ 202
Figure 7-15: (Snippet) Tasks and global variables usage............................................... 204
Figure 8-1: Overview of PuLSE....................................................................................... 211
Figure 8-2: Steps in the reconstruction process............................................................. 212
Figure 8-3: Mapping - An example................................................................................. 213
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-4</td>
<td>Ownership architecture – An example.</td>
</tr>
<tr>
<td>8-5</td>
<td>Overview of ECS.</td>
</tr>
<tr>
<td>8-6</td>
<td>Memory size of ECS.</td>
</tr>
<tr>
<td>8-7</td>
<td>Conceptual architecture of ECS.</td>
</tr>
<tr>
<td>8-8</td>
<td>A portion of developer distribution.</td>
</tr>
<tr>
<td>8-9</td>
<td>Implemented software architecture.</td>
</tr>
<tr>
<td>9-1</td>
<td>The specified module decomposition structure.</td>
</tr>
<tr>
<td>9-2</td>
<td>The actual module decomposition structure.</td>
</tr>
<tr>
<td>9-3</td>
<td>The build process of the SNAS.</td>
</tr>
<tr>
<td>9-4</td>
<td>Contents of the jar file for each subsystem.</td>
</tr>
<tr>
<td>9-5</td>
<td>Implemented package dependency of the TSAFE.</td>
</tr>
<tr>
<td>9-6</td>
<td>Conventional design with socket connections.</td>
</tr>
<tr>
<td>9-7</td>
<td>Software bus design with standarized interfaces.</td>
</tr>
<tr>
<td>9-8</td>
<td>The abstraction of thread APIs in the NASA’s GMSEC.</td>
</tr>
<tr>
<td>9-9</td>
<td>The OSAL in the NASA’s CFS.</td>
</tr>
<tr>
<td>10-1</td>
<td>An automaton for stripping C/C++/Java style comments.</td>
</tr>
<tr>
<td>10-2</td>
<td>An example SAVE-LIGHT view.</td>
</tr>
<tr>
<td>10-3</td>
<td>Structural dependency between two subsystems.</td>
</tr>
<tr>
<td>10-4</td>
<td>Sequence diagram using Dyn-SAVE for Figure 10-3.</td>
</tr>
<tr>
<td>10-5</td>
<td>The view after expanding the “System” of Figure 10-3.</td>
</tr>
<tr>
<td>10-6</td>
<td>Sequence diagram synchronized with Figure 10-5.</td>
</tr>
<tr>
<td>10-7</td>
<td>A typical term-doc matrix.</td>
</tr>
<tr>
<td>10-8</td>
<td>Geometric views of a term-doc matrix.</td>
</tr>
<tr>
<td>10-9</td>
<td>Friends of the “password” concept.</td>
</tr>
<tr>
<td>10-10</td>
<td>Top 20 friends of DbSic.java.</td>
</tr>
<tr>
<td>10-11</td>
<td>A demo of the vector NOT operator.</td>
</tr>
<tr>
<td>10-12</td>
<td>Visualizing all documents in two dimensions.</td>
</tr>
</tbody>
</table>
List of Tables

Table 1-1: Mapping the research questions to chapters ................................................................. 9
Table 1-2: Mapping the research questions to industrial systems ................................................. 10
Table 2-1: Sample rules and related quality properties ................................................................. 28
Table 2-2: Mapping of architecture to source code ........................................................................ 28
Table 2-3: Sample relations extracted from code ............................................................................. 29
Table 2-4: Summary of clone measurement ......................................................................................... 33
Table 2-5: Analysis of unused interfaces .......................................................................................... 38
Table 2-6: Statistics of rules and violations ......................................................................................... 40
Table 3-1: Questions for reviewing the existing unit test code ......................................................... 47
Table 3-2: The number of stubs used for testing each core module ............................................... 56
Table 3-3: LOC comparison of the production code and the stub code ........................................... 57
Table 3-4: Interface coverage by unit tests ......................................................................................... 57
Table 3-5: Number of functions unit tested directly .......................................................................... 58
Table 3-6: LOC comparison of the production code and the test code .......................................... 58
Table 3-7: LOC of directly tested production code and the test code ............................................. 59
Table 3-8: (Sample) The original behavior vs. the behavior for testing ........................................... 63
Table 3-9: Answers to the analysis questions ..................................................................................... 63
Table 4-1: Type definitions for CP-nets ............................................................................................. 89
Table 4-2: Interpretation of CP-net Elements .................................................................................... 90
Table 4-3: Data types of P-F Elements .............................................................................................. 93
Table 4-4: Data types for the Places of CP-nets ............................................................................... 95
Table 4-5: Effort Data for architecture compliance checking .......................................................... 104
Table 4-6: Size of the MFP Prototype under Study ......................................................................... 105
Table 5-1: An excerpt of middleware vendors’ APIs ....................................................................... 115
Table 5-2: Answers to the analysis questions ................................................................................... 125
Table 6-1: Distribution of internal and external dependencies ......................................................... 131
Table 6-2: Some relations of the knowledge base .............................................................................. 140
Table 6-3: Some sets of the knowledge base .................................................................................... 140
Table 6-4: Definitions of extracted sets from the source code ....................................................... 142
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-5</td>
<td>Definitions of extracted relations from the source code</td>
<td>142</td>
</tr>
<tr>
<td>6-6</td>
<td>Sample relations for identifying runtime components</td>
<td>144</td>
</tr>
<tr>
<td>6-7</td>
<td>Sets for identifying runtime components</td>
<td>144</td>
</tr>
<tr>
<td>6-8</td>
<td>Extracted sets from the source code</td>
<td>144</td>
</tr>
<tr>
<td>6-9</td>
<td>Extracted relations from the source code</td>
<td>145</td>
</tr>
<tr>
<td>6-10</td>
<td>Extracted relations between function and queues</td>
<td>145</td>
</tr>
<tr>
<td>6-11</td>
<td>Extracted relations between functions and semaphore ids</td>
<td>147</td>
</tr>
<tr>
<td>6-12</td>
<td>Tools used in the ADAM</td>
<td>150</td>
</tr>
<tr>
<td>6-13</td>
<td>Text similarity of the server-side of mocclient</td>
<td>158</td>
</tr>
<tr>
<td>6-14</td>
<td>Text similarity of the client-side of snif</td>
<td>159</td>
</tr>
<tr>
<td>6-15</td>
<td>Summary of files reviewed for each goal</td>
<td>177</td>
</tr>
<tr>
<td>7-1</td>
<td>Snippet of the knowledge base</td>
<td>190</td>
</tr>
<tr>
<td>7-2</td>
<td>Definition of extracted sets of static structure data</td>
<td>193</td>
</tr>
<tr>
<td>7-3</td>
<td>Definition of extracted relations of static structure data</td>
<td>193</td>
</tr>
<tr>
<td>7-4</td>
<td>Definition of extracted sets of runtime structure data</td>
<td>196</td>
</tr>
<tr>
<td>7-5</td>
<td>Definition of extracted relations of runtime structure data</td>
<td>197</td>
</tr>
<tr>
<td>7-6</td>
<td>Automatically derived runtime structures</td>
<td>198</td>
</tr>
<tr>
<td>8-1</td>
<td>An example of ECS variants</td>
<td>216</td>
</tr>
<tr>
<td>8-2</td>
<td>Containment and commonality metrics</td>
<td>222</td>
</tr>
<tr>
<td>10-1</td>
<td>Dependencies from the origin folder to the target folder</td>
<td>258</td>
</tr>
<tr>
<td>10-2</td>
<td>Snippet <em>abstract</em> information for files under gui</td>
<td>259</td>
</tr>
<tr>
<td>11-1</td>
<td>Some behavioral inconsistencies among the OS wrappers</td>
<td>274</td>
</tr>
</tbody>
</table>
# Table of Contents

Acknowledgements ........................................................................................................... i  
List of Figures .................................................................................................................. vi  
List of Tables .................................................................................................................... xi  
Table of Contents ........................................................................................................... xiii  

Chapter 1 Introduction ...................................................................................................... 1  
1.1 Background ............................................................................................................. 1  
1.2 Business Rationales for Software Analysis ............................................................ 2  
1.3 Two Scenarios Covered by the Thesis .................................................................... 4  
1.4 Research Questions ............................................................................................... 6  
1.5 Mapping the Research Questions to Chapters ...................................................... 9  
1.6 Contributions ......................................................................................................... 12  
1.7 Research Methodology ......................................................................................... 14  
1.8 Origin of the Chapters .......................................................................................... 16  
1.9 Closing Remarks ................................................................................................... 20  

Chapter 2 Verifying Architectural Design Rules of a SPL ........................................ 21  
2.1 Abstract .................................................................................................................. 21  
2.2 Introduction .......................................................................................................... 21  
2.3 The CFS Product Line .......................................................................................... 23  
2.4 Verification of Architectural Rules ....................................................................... 26  
2.5 Brief Summary of Results and Lessons Learned ................................................ 40  
2.6 Closing Remarks ................................................................................................... 41  

Chapter 3 An Analysis of Unit Tests of a SPL ........................................................... 42  
3.1 Abstract .................................................................................................................. 42  
3.2 Introduction .......................................................................................................... 42  
3.3 The CFS Product Line Architecture .................................................................... 44  
3.4 Process for Reviewing Unit Tests ......................................................................... 46  
3.5 Unit Testing of Core Modules ............................................................................. 49  
3.6 Unit Testing of Application Modules ................................................................... 60  
3.7 SPL Architecture and Testability ......................................................................... 65  
3.8 Good Practices for Unit Testing and Analysis of SPLs ....................................... 70  
3.9 Related Work ........................................................................................................ 73  
3.10 Closing Remarks ................................................................................................. 75  

Chapter 4 Architecture Compliance Checking at Runtime ....................................... 76  
4.1 Abstract .................................................................................................................. 76  
4.2 Introduction .......................................................................................................... 76  
4.3 Motivation ............................................................................................................. 78
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5</td>
<td>Avoiding Performance Problems</td>
<td>240</td>
</tr>
<tr>
<td>9.6</td>
<td>Closing Remarks</td>
<td>241</td>
</tr>
<tr>
<td>10</td>
<td>Reverse Engineering Tool Suites</td>
<td>243</td>
</tr>
<tr>
<td>10.1</td>
<td>Extraction Tools</td>
<td>243</td>
</tr>
<tr>
<td>10.2</td>
<td>Analysis Tools</td>
<td>257</td>
</tr>
<tr>
<td>10.3</td>
<td>Closing Remarks</td>
<td>267</td>
</tr>
<tr>
<td>11</td>
<td>Epilogue</td>
<td>269</td>
</tr>
<tr>
<td>11.1</td>
<td>Quality by Design Instead of Quality by Tests</td>
<td>269</td>
</tr>
<tr>
<td>11.2</td>
<td>Compliance with Specified Architectural Design Rules</td>
<td>270</td>
</tr>
<tr>
<td>11.3</td>
<td>Discovery of Software Architectures</td>
<td>275</td>
</tr>
<tr>
<td>11.4</td>
<td>Organizational Aspects</td>
<td>278</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>Samenvatting</td>
<td>282</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Catalog of Analyzed Systems</td>
<td>285</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Relation Partition Algebra (RPA)</td>
<td>304</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Colored Petri Net (CP-net)</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>Curriculum Vitae</td>
<td>332</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 Background

In our modern society, software is everywhere. It is in photocopies machines, for example, for scanning and emailing physical documents [95]; it is in spacecraft, for example, for collecting data about the moon’s surface [97], [101], and [190]; it is in many of our cars, for example, for automatically controlling vehicles’ speed [107]; it is in hospitals, for example, for monitoring and maintaining patients’ blood pressure [100]. High quality is non-negotiable, given the critical roles played by software for the success of end-products [239], [108], and [126]. Thus, software testing becomes an important quality assurance activity in the software development life-cycle [281], [109], [158], and [245]. Often high-quality software has to be delivered in a short period of time but with a low engineering effort [264].

In addition, customers demand new features or customization of existing features for their products. Managing common and variable features among products for different customers is yet another challenge for organizations (e.g., [97], [137], [106], and [269]). On top of all these challenges, requirements change during software development [302], [144], and [171]. Therefore, the general software life-cycle often includes a maintenance or evolution phase in which new features are added, existing features are modified or deleted [20] and [158]. These changes have to be done without breaking the existing functionality as well as with minimal changes and rework [49], [296], [184], [201], and [297]. All these practical constraints call for a systematic engineering approach to develop, test, and evolve software-based systems in order to meet business goals, such as a low-cost, high-quality, early in the market, etc.

In recent years, the software architectural design phase has become a key “high-level” design activity for engineering of high-quality, industrial-strength, software-based systems [225], [260], and [299]. In this early phase of the software life-cycle, architects develop an abstract representation of the system to be built. This abstract representation captures the major building blocks of the system and how they interact [261]. Here, we briefly introduce three major topics of the software architecture literature, namely architectural views, architectural styles, and analysis of quality properties, which are relevant for this thesis.

Architectural Views: The existing literature contains several methods and notations for documenting the software architecture (e.g., [31], [55], [93], [170], and [248]). They all agree on documenting the software architecture using a collection of architectural views, where each view is used to document the system from a particular perspective [131]. For example, the structural or module view could be used to explain how the system is partitioned into a collection of subsystems and their interfaces; the behavioral view could be used to explain how the subsystems interact together at runtime for various usage scenarios of the system; the build view could be used to explain how the
subsystems are compiled and linked together to produce the final executable or shared libraries.

**Architectural Styles:** Another major topic in the software architecture literature is the usage of architectural styles to build systems (e.g., [43], [259], and [260]). Architectural styles offer standardized, reusable, solutions to recurring design problems by imposing a set of constraints on the structure and the behavior of the software. For example, in the pipe-and-filter architectural style, the system is structured as a collection of components, called filters, which communicate using an intermediate pipe as the connector [198]. Each filter should behave by reading data from the input pipe and writing data to the output pipe. Basically, architectural styles enforce constraints on the structure and the behavior of the components and connectors of the software architecture.

**Analysis of Quality Properties:** The existing literature offers methods for analyzing software architectures in order to evaluate quality properties such as performance, maintainability, reliability, testability, etc. (e.g., [149], [178], [217], [237], and [313]). One benefit of evaluating the software architecture early, before it has been implemented, is that if the proposed architecture does not satisfy specified quality properties, it is not expensive to choose a different architecture because it is still “just” a collection of abstract models of the system to be built. Because of this and other benefits related to reasoning about software architecture, it has become the key artifact for managing change, developing a family of products, and achieving relevant quality properties to meet the business goals of the organization.

### 1.2 Business Rationales for Software Analysis

In this thesis, we propose methods for analysis of existing software. In Appendix A of the thesis, we discuss the business rationales for software analysis of nearly two dozen industrial systems that we analyzed. In this section, we briefly characterize the business rationales for software analysis.

"If you printed all the software there is in ten-point font, you could wrap the earth in it ten times over," says Prof. Paul Klint at University of Amsterdam. Because there is so much software in many critical systems, it is becoming ever more important to remove structural and runtime defects, and for that we need powerful analysis methods and tools that can help us systematically understand and improve software [158]. The need to remove defects is apparent due to the many reported failures in the field [309]. For example, Dershowitz has a webpage of “Software horror stories” involving more than 100 real-world failures due to software [64]. Many of those failures involve loss of human lives [181].

In the case of the NASA’s Mars Climate Orbiter [194], the cost of the mission includes $327.6 million total for both orbiter and lander, $193.1 million for spacecraft development, $91.7 million for launch, and $42.8 million for mission operations. The spacecraft encountered Mars at an improperly low altitude, causing it to incorrectly enter the upper atmosphere and disintegrate. The reason for its failure is due to the failed translation of English units into metric units. Specifically, the flight system software on the Mars Climate Orbiter was written to calculate thruster performance
using the metric unit Newtons (N), while the ground crew was entering course correction and thruster data using the Imperial measure Pound-force (lbf).

Kersten, Verhoef, and Verniers noted that software plays a significant role in the financial sector [152] and [151]. The total software costs for the banks in the Netherlands are estimated at around 20-22% of the total operational costs. They highlighted the importance of managing financial risks of multi-billion dollar software projects using software analysis [291], [292] and [295]. They also noted that the software costs are rising monotonically. Software failures in the financial sector are also expensive. For example, an employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly tried to sell 610,000 shares at 1 yen [277]. Because of market rules, the accidentally sold shares could not be bought for 1 yen, but may have been sold as low as 572,000 yen each. The episode prompted a fall of 3.4% in shares of the parent company. Direct loss as a consequence of accidentally selling the company about 42 times and the aftermath of that resulted in a loss of about 225 million dollars. The fall of 3.4% in shares represents a billion dollar loss based on 2011 data. The system exchange executives acknowledged that flaws in their electronic trading system prevented the trader from correcting its order and minimizing losses.

One can attribute insufficient software analysis as one of the prime reasons for these unfortunate failures, even though the reasons for failures are often very simple. In fact, the investigation team of the Mars Climate Orbiter found that the lack of interface-level analysis and integration testing of modules that were developed by independent teams was the reason for its failure due to the failed translation of English units into metric units [194]. The investigation of the stock market episode found that the system has architectural design anomalies such as complex modular structure, and problems with input validation and error handling as well as database design issues [277]. To sum up, based on these existing real-world evidence due to software failures, it is prudent of us to perform proper analysis of software in order to detect them before they cause damage in the field.

Many of our project partners who are developing mission and safety critical software reported that their software testing efforts consume 50% to 70% of the entire development effort. Thus, project partners are always looking for strategies to reduce the testing effort, as also reported by van der Spek and Verhoef [264]. As discussed later in this thesis, software analysis is helpful to make testing easier. Some of our project partners have the need to maintain software they acquired and/or inherited from other contractors. Naturally, there is a strong demand for methods and tools to help new contractors get familiar to the previously unknown software. Some of the project partners own systems that are similar to each other and are managed independently as standalone systems. They need software analysis of their existing systems and propose a strategy to migrate existing standalone systems into a common reusable architecture in order to control costs and reduce time to market. In some cases, project partners already developed flexible architectures with plug and play of software components even at runtime. They need an independent assessment of their software from different perspectives including structural issues and related testing, performance, and maintenance risks. In one of the projects, the manager needs a software analysis to identify testing and maintenance risks due to software change requests of a multi-million dollar project.
Software analysis can sometimes be perceived as expensive. However, from our experience we found that with proper methods and tools in place, we are able to analyze large systems (i.e., up to 10 Million Lines of Code) made of several programming languages in a cost efficient way in comparison to the multi-million dollars it took to develop the system. Naturally, depending on the criticality and complexity of the software and the complexity of analysis questions, the analysis effort varied anywhere from days to a couple of months. Nevertheless, this analysis effort is still not costly in the context of multi-million dollar development projects. Even if the analysis requires highly-skilled analysts and several months of auditing, the analysis cost outperforms the cost of not doing the analysis, which is extremely expensive, similar to the two painful endeavors of aerospace and finance software, and result in difficult to repair reputational damage. The return of investment of a software analysis is typically a function of critical findings. If the analysis detects critical risks such as architectural design problems that can crash the system, result in poor runtime performance, or security issues, then the investment is often immediately justified. Thus, we always recommend and strive to find critical issues in the software under analysis to justify the return on investment in a software analysis.

1.3 Two Scenarios Covered by the Thesis

In our experiences with software-based systems, we have come across two major scenarios, which have become the focus of this thesis.

In the first scenario, the system under analysis has had an explicit software architectural design phase, as described above. However, it was difficult to determine whether or not the resulted implementation was consistent with the specified architecture because of the abstraction gap between architectural elements (e.g., components, connectors, etc.) and code-level concepts (e.g., routines, variables, data structures, etc.). It is very well possible that the implementation violates specified architectural rules during evolution. By architectural rules, we mean constraints on both the structural and the behavioral views of the system. Adding to the challenge is the fact that a few of the (modern) systems we analyzed allow software components to load and unload at runtime, without the need for stopping and restarting the system. For such types of systems, it is not only important to analyze that the implementation follows structural constraints, but also very important to analyze behavioral constraints, because wrong behaviors affect the proper functioning of the system, which is not desirable especially in mission and safety critical systems such as flight software and medical device software. Thus, we need a method to analyze whether or not the implemented system follows the specified architectural rules.

In the second scenario, the system under analysis did not go through an explicit software architectural design phase. In other words, the software architecture is implicitly present in the source code, making it difficult to see, analyze, and improve quality properties. In some systems, our collaborators do have architectural design diagrams, but these diagrams are typically at a “high level” of abstraction. Their relationship to source code, testability, performance, and maintainability risks is often tenuous due to missing details and traceability. For small systems, it is not necessarily a challenge to read through code and extract architectures. However, it is not practically
possible to do the same for large systems because there is too much code to review manually, and cross-cutting quality properties like safety or security are often scattered over the code, so even with automated tools it is not feasible to recover such quality properties from the source code. Given the importance of the software architecture for achieving and improving the software quality, there is a strong practical need for methods to discover the software architecture from the implementation and analyze quality properties, identify risks (e.g., low testability, performance, and maintainability), and suggest risk mitigation strategies.

Given the fact that testing consumes at least 50% of any software development effort [307], analysis of testing issues, and questions regarding how to facilitate testing by improving the implemented architecture, received significant attention in our engagements with customers. Industrial systems often have a fair amount of test code, which also need to be maintained and evolved [246]. Therefore, there is also a practical need for methods that can discover the architecture of tests and identify risks in the test architecture.

We also observed that there is a strong need for methods that analyze the implemented architecture and detect performance risks due to architecturally significant decisions. For example, the threading model used by the implementation in order to read incoming data from a socket and dispatch data to data processors is considered architecturally significant, because if the same thread is used to read from the socket and synchronously dispatched to data processors, then there is a risk that low performing data processors might affect the rest of the system.

In several cases, we have encountered the need to understand the heritage of the software from an organizational point of view. For example, in order to reason about the detected architectural violations and offer an appropriate improvement suggestion, we found it useful to understand organizational aspects, such as the distribution of teams, assignment of developers to modules, etc.

Some of the industrial systems described in this thesis were, in fact, developed as software product lines (a.k.a. software product family), meaning that a family of systems can be derived from a common architecture with configurable and reusable components [54]. In a software product line context, analyses of quality properties are critical because a) the quality of all derived products is influenced by the quality of the reusable components, and b) other teams and/or business units may not reuse the components developed for reuse if they are not convinced that they possess high quality, thus, affecting the return on investment in a product line [106]. Therefore, architectural analysis methods should also consider product line specific issues, for example, managing of variability issues at the code-level.

This thesis offers a practically inspired and comprehensively investigated architecture reconstruction approach, which addresses these practical needs, for analyzing quality properties, especially testability, performance, and maintainability, at the architecture level. Some of the industrial systems discussed in this thesis were developed as software product lines as well as systems that allow components to freely enter and leave at runtime. In the next section, we formulate the research areas and questions that are related to the two scenarios, which were mentioned above.
1.4 Research Questions

Our research is based on industrial experiences with architectural analyses of nearly two dozen industrial systems. Our research questions emerged from the practical needs of real-world challenges we have come across for the past decade. In a sense, we followed the action research model to a large extend. Action research is problem centered, client centered, and action oriented; please refer to Section 1.7 to learn about action research.

Our research area is related to the computer-assisted review of quality of implemented systems. We took an architecture-centric approach for evaluating quality properties of implemented industrial systems. Our central goal is to offer constructive recommendations for developers and architects so that software quality can be built-in in the first place. To this end, we list the research questions that are addressed in this thesis.

1.4.1 Built-in Quality in the First Place

Many of our research questions focus on extracting the architecture from the implementation and reasoning about architectural violations (i.e., deviation from the specified or intended architecture) and quality properties, in particular testability, performance, and maintainability after the system is implemented. Of course, we believe that such activities are of a strong practical interest to improve software quality of existing systems [167], [290] or for conducting software forensics analysis [100]. However, this type of analysis as well as the detected problems are conducted and identified too late, meaning that the system is already implemented and deployed in the field, and, thus, it takes significant effort and motivation to change existing implementations. Hence, it is natural to follow the prevention is better than cure principle, and attempt to build quality in from the beginning.

Based on the architectural analysis of nearly two dozen real-world industrial systems, we developed a large body of architectural knowledge including principles of architectures, build processes, and organizational aspects that make it difficult or even impossible to introduce architectural violations at the source code level, facilitate testing, and help in avoiding performance and maintenance issues. This leads us to the central research question:

- RQ1: How can we avoid problems such as architectural violations at the source code level, testability, performance, and maintainability risks in the very first place?
  - That is, what characteristics of architectures, build processes, and organizations impede or even totally prevent architectural violations, facilitate testing, and avoid or reduce performance and maintenance risks?

In order to reason about this research question, we study a collection of real-world systems and characterize good practices and principles that could be employed to build-in quality instead of tested-in quality for developing new systems or improving existing systems. Thus, we refine our research question into the three interleaved areas.
1.4 Research Questions

- Verification of Architectural Design Rules
- Discovery of Software Architectures and Analyses of Quality Properties
- Organizational Aspects

1.4.2 Compliance Checking of Architectural Design Rules

We have come across some real-world systems which went through a detailed architectural design phase, and thus contain architectural specifications. The challenge was to analyze whether or not the implementation indeed followed those specified architectural rules, including structural and behavioral rules. In addition, some of these systems were built using flexible, standard architectural styles, namely the pipe-and-filter and publish/subscribe styles, enabling a systematic development of a family of products. It was even possible to plug-and-play software components at runtime, without stopping and restarting the running system. For these types of systems, in addition to static analysis (i.e., without running the system) of the source code, we had to perform dynamic analysis (i.e., monitoring the running system) because the actual architecture is only known at runtime. These characteristics lead us to the following research questions:

- RQ2: How can we analyze that the implementation conforms to the specified architecture?

In this thesis, we develop methods for analyzing the structural and behavioral constraints of architectural styles. Let us recall the fact that architectural styles impose constraints on both the structure and the interaction behavior of involved components and connectors. One positive side-effect of this “architectural style driven reverse engineering” is that analysis techniques and technical infrastructures could be reused for all systems based on the same architectural style, as illustrated in this thesis using case studies. In addition, we can focus the reverse engineering activities to architectural constraints of styles, and ignoring or filtering out irrelevant information.

Thus, we can refine this high-level question into the following specific questions:

- RQ2.1: How can we statically analyze that the specified structural rules of architectural styles are followed by the implementation?
- RQ2.2: How can we dynamically analyze that the specified behavioral rules of architectural styles are followed by the implementation?
- RQ2.3: How can we combine static and dynamic analyses for compliance checking of static and behavioral rules of architectural styles?

1.4.3 Discovery of Software Architectures and Analyses of Quality Properties

The practical need behind our analysis is that our customers wanted an “independent eye” to evaluate the quality of the implementation, identify risks, and offer mitigation
strategies. In many of those cases, the software architecture is hidden deep in the implementation, making it very difficult to see the architecture and evaluate quality properties, such as testability, performance, and maintainability. Often, these systems are “old”, large, inherently complex, use multiple programming languages and technologies, and different types of build processes. In addition, several of those systems also had a large amount of the test code, whose architecture is also implicitly present in the test code. Our task was to make the architecture explicit, evaluate quality properties, and identify quality risks.

These tasks lead us to the following research questions for this thesis:

- **RQ3**: How can we efficiently discover software architectures and analyze quality properties without reviewing inhibitive many source code files?

When we extracted and visualized module views of software systems, we realized that, even well-engineered systems look like a dense graph. This was because all concerns such as persistence, GUI, licensing, error-handling, etc. were part of the reverse engineered model, making it difficult to see hidden layers and pinpoint testability and performance risks. Therefore, we decided to develop a reverse engineering method that allows us to analyze the system with respect to a given concern, and see the hidden inner beauty of the implemented architecture. Thereby, we can scale our reverse engineering method to real-world systems. Thus, we can refine the above question as follows:

- **RQ3.1**: How can we analyze the systems’ implementation and discover architectural views for various concerns, such as persistence, GUI, OS variants, etc.?
- **RQ3.2**: How can we identify architectural design decisions that facilitate or impede testing?
- **RQ3.3**: How can we identify implemented architectural design decisions that attribute to performance risks?
- **RQ3.4**: How can we assess the maintainability of the test code?

### 1.4.4 Organizational Aspects

In several endeavors with our customers, we observed that there is a strong relationship between software architectures and organizational aspects such as business goals, the structure of teams, the assignment of developers to modules, and also the heritage of subsystems, for example, merge or reuse of existing subsystems which were developed by different contractors. Thus, our architectural discovery and analysis activities also take into the consideration such organizational aspects. For example, in a case study, we found that one of the subsystems affected the common look-and-feel because it interacts with the database in a different style – relative to other subsystems - had its own implementation of logging, string, and other utilities.

After we analyzed the distribution of developers to subsystems, it became clear that this subsystem was a) implemented by a different group of developers, and b) reused source code from a different system. We would not have understood this reason unless we
analyzed the system from an organizational point of view. In addition, we tried to establish links between implemented software architectures and business goals. We did so by discussing our results with key project members such as project managers, technical leads, and senior developers. We found it useful to document the connection between important architectural decisions and business goals so that all members of the team understand the importance of software architectures. Thus, the following questions are of interest in the context of architectural analyses of implemented systems.

- RQ4: Do organizational aspects influence the implemented architecture?
  - RQ4.1: How are the implemented architectural decisions related to business goals of organizations?
  - RQ4.2: Can we use the work assignment relation to identify and reason about architectural issues that impact understandability and maintainability of systems?
  - RQ4.3: Can we use the work assignment relation between developers and files to understand the modular structure of the implementation?

1.5 Mapping the Research Questions to Chapters

Here, we map research questions to the various chapters of the thesis, see Table 1-1. We also briefly explain commonalities among chapters and how the chapters complement each other.

Table 1-1: Mapping the research questions to chapters

<table>
<thead>
<tr>
<th></th>
<th>RQ1</th>
<th>RQ2.1</th>
<th>RQ2.2</th>
<th>RQ2.3</th>
<th>RQ3.1</th>
<th>RQ3.2</th>
<th>RQ3.3</th>
<th>RQ3.4</th>
<th>RQ4.1</th>
<th>RQ4.2</th>
<th>RQ4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 3</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 5</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Chapter 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We approached these research questions in close collaboration with industry. In fact, these research questions were identified and formulated during our architectural analysis activities of several industrial systems, which are listed in Appendix A of the thesis. However, we selected only a subset of those systems and listed them in Table 1-2 because a) these systems went through a relatively in-depth analysis and significantly contribute to the research questions, and b) the analysis results involving these systems were published and hence are disclosed in the thesis.
Chapter 2: Verifying Architectural Design Rules of a SPL. In this chapter, we cover the research questions RQ2.1 and RQ4.1. The high-level research questions are a) how can we analyze whether or not the specified product line architectural rules are followed in the implementation? and b) how are the implemented decisions related to business goals? These research questions were investigated using the NASA’s core flight software product line (CFS) as the case study. This chapter was published at the international conference on software product line (SPLC), in 2009 [97].

Chapter 3: An Analysis of Unit Tests of a SPL. In this chapter, we cover the research questions RQ3.2 and RQ3.4. The high-level research questions are a) how can we analyze the maintainability of test code? and b) how can we identify architectural design decisions that facilitate or impede unit testing? In addition, this chapter explains the importance of enforcing architectural rules to make the test architecture easy to understand and evolve. These research questions were investigated using the NASA’s core flight software product line (CFS), which was also used in Chapter 2 for verification of architectural rules. This chapter complements the previous chapter by focusing on the testing aspects of the CFS architecture, which was extracted in Chapter 2. This chapter was published at the international conference on software product line (SPLC), in 2010 [101].

Chapter 4: Architecture Compliance Checking at Runtime. In this chapter, we cover the research question RQ2.2. The high-level research question is: how can we monitor a running-system, which may allow components to freely enter and leave, and detect violations of the behavioral rules of architectural styles at runtime? This research question was investigated using Ricoh’s photocopy machine software prototype as the case study. In this study, we discovered violations of the pipe-and-filter architectural style used for building the system as well as other types of risks, including the possibility of misusing the APIs of components that could even halt the system at runtime. We formalized the approach using Colored Petri nets (CP-nets), see Appendix C for an overview of CP-nets. This chapter complements previous chapters by focusing on behavioral analysis of software components in the context of the pipe-filter architectural style. This chapter was published in the journal of Information and Software Technology (IST), in 2009 [95].

Chapter 5: An Analysis of the Publish-Subscribe Style. In this chapter, we cover the research questions RQ2, RQ3.1, and RQ4.1. The high-level research questions are a) how can we discover architectural violations of systems based on the publisher-subscriber style? b) how can we analyze the system with respect to publisher-subscriber concepts?, and c) how are the implemented architectural decisions related to business goals? These research questions were investigated using the NASA’s GMSEC software product line as the case study. We discuss how flexible architectural styles, which

<table>
<thead>
<tr>
<th></th>
<th>RQ1</th>
<th>RQ2.1</th>
<th>RQ2.2</th>
<th>RQ2.3</th>
<th>RQ3.1</th>
<th>RQ3.2</th>
<th>RQ3.3</th>
<th>RQ3.4</th>
<th>RQ4.1</th>
<th>RQ4.2</th>
<th>RQ4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA CFS</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Ricoh</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NASA GMSEC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NASA SNAS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US Army CARA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hitachi</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
allow plug-and-play of components, impede analyzing and validating quality properties. In particular, one problem we detected was caused by behavioral inconsistencies between software components that implement the same interface. This problem affected the goal of “safe” plug-and-play of components. It is worth noting that this chapter replicates some of the analysis techniques of Chapter 4 to a different architectural style – that is, the publisher-subscriber style. This chapter was published at the IEEE international working conference on reverse engineering (WCRE), in 2010 [103].

Chapter 6: Architecture Discovery and Analysis Method (ADAM). In this chapter, we cover the research questions RQ3, RQ4.2, and RQ4.3. The high-level research questions are a) how can we efficiently discover the software architecture hidden in the source code, without reviewing inhibitably many files? and b) how can we identify potential risks related to performance, testability, and maintenance, using the discovered architecture? In addition, this chapter explains the relationship between organizational aspects and the discovered architectural issues that impact quality. It is worth noting that Chapter 3 also focuses on the evaluation of the testability of the system, however, this chapter includes an analysis of testability in the presence of databases and GUIs concepts. These are two major concepts that often affect testing if they are not properly addressed by the architecture. The method introduced in this chapter is based on the premise that the architecture of a system is influenced by external entities, such as COTS software, frameworks, and OS and programming language libraries. This chapter offers evidence on the usefulness of the method by demonstrating it on an industrial system, developed by Honeywell for the NASA’s Space Network Access System (SNAS) as the case study. At the time of finalizing the thesis, this chapter was under review by the editorial board of the ACM Transactions on Software Engineering (TOSEM).

Chapter 7: Architecture Discovery of Medical Device Software. In this chapter, we cover the research questions RQ3.1 and RQ3.2. The high-level research question is: how can we efficiently discover static as well as runtime structures hidden in the source code for the purpose of evaluating testability risks. This chapter leverages the idea of a knowledge base that was introduced in Chapter 6. Further, this chapter focuses on evaluating testability at the architecture level by reverse architecting a suite of structures. Each structure offers an abstract, yet precise, view of the system with respect to one concern. For example, one runtime structure can show how modules are partitioned into concurrent runtime tasks, and another runtime structure can show how tasks communicate with each other. This chapter offers evidence on the usefulness of the approach by demonstrating it on safety critical medical device software called the Computer-Assisted Resuscitation Algorithm (CARA), developed at the US Army Walter Reed Research Center. This chapter was published at the IEEE Working Conference on Software Architectures (WICSA), in 2011[100].

Chapter 8: Discovering Organizational Aspects for Migration to a SPL. In this chapter, we cover the research question RQ4.3. The high-level research question is: how can we discover organization views and understand their relationship with implemented architectures, using the relation between developers and the files they worked on? The research question was addressed using two versions of Hitachi’s engine control system (ECS) for automobiles. The context of the case study was in introducing a software
product line using existing versions. We analyzed “owners” of component variants and how we can bring together components owned by different teams into a product line. This chapter was published at IEEE WCRE, in 2006 [107].

Chapter 9: Quality by Design - Some Recommendations. The purpose of this chapter is to use the results of all case studies as the supporting evidence for answering the central research question RQ1. That is, based on several in-depth case studies of previous chapters, we offer concrete recommendations for avoiding architectural problems in the first place, which could be reused in other projects either for developing new systems or to improve existing systems. Equivalently, in this chapter, we offer generally applicable recommendations for build-in quality instead of test-in quality.

Chapter 10: Reverse Engineering Tool. The purpose of this chapter is to give a summary of the tools we used in our architectural analysis projects. We developed several tools, but we also reused many existing tools. Our tool-box covers extraction and analysis phases of reverse engineering [168] and [37]. The extraction tools are used for extracting various types of data such as the Include relation between files, the Call relation between functions, clone data between files, the relation between developers and the files they worked on, etc. In addition to this static data, we also developed tools for extracting dynamic data such as the Call relation at runtime, timing, etc. The analysis tools are used to lift the collected detailed data to an architectural level, showing the modular structure as well as the runtime behavior of the system under analysis. The presentation tools are used to visualize analysis the abstracted architecture and its source code base.

1.6 Contributions

We believe our central contribution is related to the collection and the discussion of constructive recommendations that aid in avoiding architectural problems in the first place before they occur. In particular, we recommend architectural practices that prevent violations of architectural rules, improve testability, and minimize performance and maintenance risks. These recommendations are elaborated in Chapter 9. In order to collect constructive recommendations to developers, we did several architectural analysis activities of industrial systems in a close collaboration with our customers, following the action research model. These activities themselves given rise to the following contributions:

Our first contribution is an approach for verification of architectural rules of systems based on architectural styles [95], [97], and [103]. From the definition of architectural styles, we show how to derive and check whether or not the implementation follows both the structure and behavioral constraints of styles. Using the proposed method, we discovered structural and critical behavioral problems at runtime, which would not have been detected without applying the proposed method. For several case studies of industrial systems, using the proposed method, we discovered high-priority bugs and component interoperability problems due to the violations of the behavioral constraints of architectural styles. This contribution is elaborated in Chapter 2, Chapter 4, and Chapter 5.
Our second contribution is related to architectural analyses of the test architecture used for testing the system under analysis [102]. Similar to the source code, the test code has to be maintained and evolved. Thus, the architecture of the test code has to be analyzed and improved where possible. We developed an approach for analyzing the architecture of the test code. This approach was used to investigate the question of: “What architectural decisions facilitate or impede testing”. In one of the case studies, we applied the approach and located the high-level design decisions that made the software harder to test, and made the test code complex to maintain. This contribution is elaborated in Chapter 3.

Our third contribution is based on the observation that the module dependency diagram, extracted from the source code tends to look complex even for the case where the system under analysis is well-engineered [99]. This is because all concerns (e.g., persistence, logging, error handling, licensing, security, and OS variability) are still part of the dependency diagrams. Therefore, we developed an approach for discovering software architectures from the source code and analyzing its quality properties, such as testability, performance, and maintainability. The key premise of the method is that architectural decisions of implemented systems are inspired and influenced by external entities (e.g., COTS, Frameworks, and programming language libraries). Based on experiences with architectural analyses of several industrial systems, we have developed a knowledge base of external entities that contains names of header files, routines, classes, etc. that are used for not only discovering architectural features but also quality risks. We show how to use the knowledge base to discover architectures hidden in the implementation, without reviewing inhibitive many files. In one of the case studies of an industrial system, using the proposed approach, we discovered testability, performance, and maintenance risks of a large NASA system by reviewing not more than 4% of source code files. This contribution is elaborated in Chapter 6.

Our fourth contribution is related to the discovery of static and runtime structures from the source code to reason about testability (i.e., ease of testing) at the architecture-level. We formalize, using Relation Partition Algebra (see Appendix B), an approach for discovering several runtime structures such as a) how static module structures are transformed into runtime components (e.g., tasks), b) how the runtime components communicate using intermediate connectors (e.g., queues), and c) how the runtime components synchronize. In addition, we explain how we can make use of the extracted runtime structures for reasoning about testability at the architecture level. Further, we also demonstrate how the extracted structures are used to configure state-of-the-art static software verification tools for a comprehensive analysis. We successfully applied the proposed approach in the context of the architecture-level safety analysis of medical device software called the CARA, developed at the US Army Walter Reed Research Center. We performed this research at the US Food and Drug Administration (FDA) campus to understand and demonstrate the benefits of integrating safety analysis of medical device software with architecture analysis during the pre-market and post-market approval process. This contribution is elaborated in Chapter 7.

Our fifth contribution is related to the understanding of the relationship between organizational aspects and the implemented software architecture [106] and [107]. For example, which teams develop what parts of the system? Do different teams implement the same concern with the same common look-and-feel? We developed a simple, yet
effective, approach for obtaining answers to such questions. In one of the case studies, we applied the proposed method and understood which developers are “owners” of which parts, and how many developers are working on different parts, etc. This case study has shown us that it may be possible to predict the high-level structure of the system under analysis by using the relation between developer and the files they worked. This relation is often stored in the source code change history log of configuration management systems. This contribution is elaborated in Chapter 8.

To sum up, the thesis contributes a practically inspired approach, which was investigated in close collaboration with industry, for:

1. Enabling build-in quality instead of test-in quality using a large body of architectural knowledge derived from analysis of industrial systems. We suggest constructive recommendations for developers to: a) avoid or minimize architectural violations at the code level, b) facilitate improved testability, and c) minimize performance and maintenance risks.
2. Discovering the software architecture from the implementation, and analyzing quality risks, in particular, risks related to performance, testability (with emphasis on unit testing), and maintainability.
3. Checking the structural and behavioral constraints of the specified architectural style with respect to its implementation.
4. Identifying and analyzing the test architecture, and thereby improving our understanding of relationships between software architectures and testing.
5. Understanding and analyzing organizational aspects of the system under analysis and relationship to software architectures.

1.7 Research Methodology

At Fraunhofer, often our projects are performed in a close collaboration with customers using the “industry-as-laboratory” paradigm, as proposed by Potts [235]. This research approach should be viewed as a complement to the traditional “research-then-transfer” approach (which in its worst form is similar to what Glass calls “advocacy research” [117]) as a solution to the problem of lack of influence. In the “industry-as-laboratory” approach researchers identify problems through close involvement with industrial projects, and create and evaluate solutions in an almost indivisible research activity. The result would be that problems investigated are “real problems”, that the need for a technology transfer process decreases, and that research becomes problem focused. The vehicle for such research would be case studies, the study of real system-development projects and their context. Glass’ suggestion is that software practice and research should work together, both to obtain input regarding the hard problems and then in the evaluation phase when new ideas are about to be tested.

1.7.1 Action Research Model

Our research approach somewhat resembles the principles of the action research (AR) model, coined by Lewin [182]. Action research is concerned with change. Regular
research methods such as controlled experiments (and even case studies) keep the researcher and the study object strictly separated so that the researcher’s influence on the behavior of the subject is minimized. Thus the goal is not to change anything. AR is a research method in which change is desirable. In AR, the researcher actively influences and changes the behavior of the subject. Actually, if the researcher cannot demonstrate that any change was achieved, many supporters of AR would consider the study a failure.

Figure 1-1 summarizes the steps and processes involved in planned change through action research, as given in [4].

Lewin’s description of the process of change involves three steps [4]: Unfreezing: Faced with a dilemma or disconfirmation, the individual or group becomes aware of a need to change. Changing: The situation is diagnosed and new models of behavior are explored and tested. Refreezing: Application of new behavior is evaluated, and if reinforcing, adopted.

Action research is depicted as a cyclical process of change. The cycle begins with a series of planning actions initiated by the client and the change agent working together. The principal elements of this stage include a preliminary diagnosis, data gathering, feedback of results, and joint action planning. In the language of systems theory, this is the input phase, in which the client system becomes aware of problems as yet unidentified, realizes it may need outside help to effect changes, and shares with the consultant the process of problem diagnosis.

The second stage of action research is the action, or transformation, phase. This stage includes actions relating to learning processes (perhaps in the form of role analysis) and to planning and executing behavioral changes in the client organization. Included in this stage is action-planning activity carried out jointly by the consultant and members of the client system. Following the workshop or learning sessions, these action steps are carried out on the job as part of the transformation stage.

The third stage of action research is the output, or results, phase. This stage includes actual changes in behavior (if any) resulting from corrective action steps taken following the second stage. Data are again gathered from the client system so that progress can be determined and necessary adjustments in learning activities can be made.
1.7.2 Action Research for Architecture Discovery and Analysis

Most of our research work is funded by industrial customers. Before we get funding, typically a rigorous proposal needs to be written. The proposal should already highlight the research motivation, the research goal, technology infusion or transfer plan, expected benefits and impacts, and quarterly deliverables. This proposal template forces us to think up-front and work together with our customers to understand their problems and needs. We organize workshops with our customers in order to identify problems and needs. In our research context, typical problems and needs include requirements such as evaluating code quality, checking whether the implementation is consistent with architectural structure and behavioral rules, and constructive recommendations for improving code quality. This phase is essentially about planning and defining the goal for enabling change in development and quality assurance processes.

Once the problems and needs are identified, our customers offer us the artifacts such as source code, requirements documents, design documents, and test cases. Using our reverse engineering methods and tools, as introduced in this thesis, we analyzed the artifacts of our customers or partners. Our findings and recommendations for improvements are discussed in a workshop mode involving developers, testers, architects, and managers. Customers typically take actions of our recommendations and after a few months we get updated artifacts for analysis. An action could be, for example, fixing an architectural violation that was pointed out by our analysis. We also analyze the updated artifacts in order to make sure that our recommendations are implemented in the right way. This cycle of analysis, actions, improvement, and feedback continues in a similar fashion to the action research model, see [187], [97], and [103] for more information.

Since action research is all about change (one can consider it a failure if no change occurred) it’s important to explicitly describe the change that occurred. We observed various forms of change in the organizations we are fortunate to collaborate. First, they learned about our technologies and adopted them after “seeing” the value added to the quality of their artifacts. In fact, many of the tools discussed in this thesis are installed and used frequently in our customers’ site. Second, they “talk” about software architectures, abstractions, testability and performance risks in a much more matured and disciplined way. Third, we get updated artifacts in order to make sure that they fixed the issues reported by us. Fourth, despite the busy development life-cycle, our customers actively took part in reading, editing, and commenting technical papers that were written together with them as co-authors. Last, but not least, many of our customers continue to work with us for several years in architectural analysis projects. In our opinion, all these facts are good indications that something has truly changed. Thus, we believe in order to cause change researchers should work closely with the customer.

1.8 Origin of the Chapters

Almost all of the remaining chapters of the thesis are based on a suite of peer-reviewed international journal and conference papers, which are listed below. In addition, we
also list miscellaneous papers, a few technical reports, and masters’ thesis that support this thesis, for example, by contributing tools to apply the approach in practice.


   

   


5. Dharmalingam Ganesan, Mikael Lindvall, and Monica Ron. *External Dependencies-driven Architecture Discovery and Analysis of Quality Properties*. At the time of finalizing the thesis, this chapter was under review by the editorial board of ACM Transactions on Software Engineering (TOSEM), see Chapter 6.


8. Based on the analysis of several industrial systems, we derive generally applicable recommendations for built-in quality so that we can make use of the body of architectural knowledge for not only improving existing systems but also during the architectural design of new systems. These practically relevant recommendations for improved testability, performance, and maintainability are the highlights of Chapter 9. A paper is in progress based on the content of this chapter.

9. In the course of our research, we developed a suite of reverse engineering tools. We also reused existing tools from other resources such as the existing literature (e.g., [169]) and open source projects. We present an overview of our tool suite in Chapter 10.

10. We present the epilogue of the thesis by revisiting the list of research questions and discuss how we addressed them. We also discuss open issues for future research. The epilogue concludes the thesis in Chapter 11.

**1.8.1 Miscellaneous Papers — not included in the thesis** *(sorted by year)*

The following papers inspired our research. However, we did not include the following papers because they were preliminary versions written at our early stage of research, or the papers are not necessarily well cohesive to the technical scope of the thesis.


Introduction


1.8.2 Technical Reports and Masters’ Thesis


1.9 Closing Remarks

In this chapter, we addressed the following: First, we offered the background and motivation of the thesis. Second, we discussed the two practically significant scenarios of the thesis. Third, we discussed the research questions that were derived from those two scenarios. Fourth, we mapped the research questions to different chapters of the thesis, and discussed technical cohesiveness of the remaining chapter of the thesis. Fifth, we highlighted the contributions of the thesis. Sixth, we introduced the action research methodology. Finally, we listed the international journal and conference papers that form the origin of the remaining chapters of the thesis.
Chapter 2
Verifying Architectural Design Rules of a SPL

2.1 Abstract

This chapter presents experiences of verifying architectural design rules of the NASA Core Flight Software (CFS) product line implementation. The goal is to check whether the software product line (SPL) implementation is consistent with the CFS’ architectural rules derived from the developer’s guide. The results indicate that consistency checking helps a) identifying architecturally significant deviations that were eluded during code reviews, b) clarifying the design rules to the team, and c) assessing the overall implementation quality. Furthermore, it helps connecting business goals to architectural principles, and to the implementation. This chapter is the first step in the definition of a method for analyzing and evaluating product line implementations from an architecture-centric perspective.

Keywords: Business goals, Architectural Rules, Implementation, and Flight Software.

2.2 Introduction

It is a well-known fact that the software architecture is critical to the success of a software product line. This is reflected in the fact that organizations often spend significant effort on the design of the software product line architecture and strive to choose the most beneficial architectural styles, patterns, and decomposition strategies (e.g., [31], [55], [234], [19], and [157]). These architectural design decisions are made to efficiently establish the core for a family of products, by taking advantage of their commonalities and carefully managing variability. One important factor determining the success of a software product line (SPL) is whether the “designed” variability is indeed present and maintained in the implementation. Thus, the challenge is to verify that the resulting implementation is consistent with the intended architectural design principles (e.g., [215], [233], [272], and [271]).

There are several reasons why the implementation might deviate from the architecture: a) the architecture is an abstract entity not directly expressible using standard programming languages, b) the architecture is typically not documented well enough for developers to be able to fully comprehend and follow during the implementation, c) performance and other non-functional issues, not easily detectable at design time, have to be resolved with code-level workarounds, for example, an unanticipated requirement to run the software on slow processors would often require code-level workarounds that

---

2 Based on the paper published at the IEEE International Conference on Software Product Line (SPLC 2009), IEEE Computer Society Press [97].
might compromise some of the architectural decisions that were made in the beginning, and d) complexity in managing source code level variation points. These characteristics make it difficult and tedious to ensure that architectural principles are met through traditional inspections and reviews.

The Flight Software Systems Branch at NASA Goddard Space Flight Center (GSFC) has spent considerable resources over the past few years developing the Core Flight Software (CFS) and positioning it as the future flight software platform for NASA missions [214]. The CFS follows a product line approach with the goal to support systematic reuse. The business goals of the CFS are the main drivers for creating and maintaining the product line architecture. Consequently, many of the architectural decisions are directly influenced by the business goals of the CFS. Thus, the CFS team needs to ensure that the business goals and the implementation are aligned by verifying that the implementation indeed follows the architectural principles, and the team has been doing so as part of rigorous code reviews.

The CFS developer’s and deployment guide specifies the architecture principles in terms of structural and behavioral properties of the CFS product line architecture. Thus, it is possible to derive various types of architectural rules from the architecture principles (e.g., from the architectural principle of layering, the rule that a lower layer cannot use a higher layer can be derived). The scope of this chapter is the subset of rules that are related to the module structure (i.e., the module architecture [263]) of CFS and its development environment (i.e., the code architecture), and that are verified statically (i.e., without executing the system). The derived rules are categorized into module dependency-restriction rules, decomposition-restriction rules, redundancy-restriction rules, and miscellaneous rules, including visibility-restriction rules, conditional preprocessor directives-usage rules, and interfaces usage rules. These rules directly and indirectly address various concerns, such as runtime adaptability, portability, testability, and performance of the product line.

The analysis and verification of these architectural rules are conducted against the most recent source code version of the CFS product line, which includes the modules of the core framework and a set of optional reusable application modules. The results of the verification of these rules show that most of them are indeed followed in the implementation. Naturally, some deviations were detected through this process, and the high-priority issues are currently being addressed.

Discussions with the CFS team revealed that the software is written and reviewed by experienced engineers who have been developing flight software for about 15-20 years. Nevertheless, the team believes that this tool-supported independent verification of architectural rules complements such reviews because it identified architectural deviations that escaped the manual review process. Thus, it helps to preserve and protect the carefully designed variability points, with the effect that it increases the confidence of the overall product line quality. Furthermore, the analysis helps in establishing an explicit mapping among business goals, architectural principles, and implementation decisions [270].

This chapter is the first step in the definition of a method for analyzing and evaluating product line implementations from an architecture-centric perspective. Complementary analysis of rules that are outside the scope of this chapter (e.g., rules related to the
behavior of the system) will be added to the method in the future. Contributions of this chapter are: 1) a demonstration of how a collection of architecturally-relevant rules are verified and analyzed in order to make sure the implementation is aligned with business goals, 2) data and insights from a product line implementation contributing towards developing a benchmark for evaluating the implementation quality of product lines.

2.3 The CFS Product Line

In this section, the heritage and business goals of the CFS product line are briefly discussed. In order to illustrate the importance of verifying the architecture rules against the implementation, the relationship between business goals and architectural rules is presented.

2.3.1 The Heritage of the CFS

In the past, when developing a new mission, the Flight Software (FSW) lead for the mission would obtain existing FSW and artifacts from heritage missions that they knew well (see Figure 2-1). As the figure shows, the FSW branch at NASA GSFC has several “heritage architectures” to choose from. Once a fitting heritage mission had been selected, changes were made to the software artifacts in order to implement the requirements of the new mission. In addition, changes in the flight hardware or changes in the operating system caused changes throughout the software system. This ad hoc reuse implied that, for example, integration of new modules required extensive manual coordination. The conclusion was that this model, which was based on reuse of selected heritage architectures, was not flexible enough for collaboration within GSFC, with other NASA centers, or with outside entities. In addition, this reuse model forced the on-orbit FSW maintenance team to understand, in detail, each heritage architecture and its implementation because of the differences between the missions. Thus, cost advantages from this type of reuse were not visible.

Prior to the year 2000, FSW was developed in multiple branches. A single FSW branch was established in the year 2000 creating the foundation for a new FSW product line as a response to the software reuse problem. A heritage analysis, among the past missions, as illustrated in Figure 2-1, showed that the requirements for Command and Data Handling (C&DH) Flight Software are indeed very similar from mission to mission. This heritage analysis was the starting point for the GSFC FSW branch’s establishment of guidelines for the CFS product line, with the primary goal of not “re-inventing the wheel” in each mission project. In 2003, the development of the flight software product line started. In 2009, the CFS was used by several projects. The CFS team turned the existing variants into one overall system where the points of variation are migrated into a product line, this strategy is often called a “reactive product line” as in [75], for example.
2.3.2 Business Goals and Architecture of the CFS

The CFS product line is being developed to achieve the following business goals: a) reduce time to deploy high quality flight software, b) reduce project schedule and cost uncertainty, c) enable collaboration across organizations, and d) establish common standards and tools across the FSW projects and NASA wide, f) establish the use of a single platform for advanced concepts and prototyping, and g) directly facilitate formalized software reuse.

In order to achieve the business goals, the CFS team developed a software product line architecture based on solid software engineering principles, such as abstraction, information hiding, and modularity [222]. A layered architecture that hides the internal details of OS and hardware platform has been defined [219]. Core modules configurable for mission-specific needs were developed for reuse, forming the core layer of the CFS. A set of optional reusable modules (also called application modules) were also developed on top of the core layer. These application modules are optional, meaning that they need not be used in all missions. Mission-specific functionalities are introduced by plugging-in modules into the application layer. A detailed API specification explaining how to use the core or generic modules, (shown in the core layer in Figure 2-2) was also documented. Runtime module registration mechanisms have been created for integrating modules with little human effort. The CFS facilitates this mechanism by using a publish-and-subscribe architectural style with a software message bus (see Figure 2-2) as the middleware.
2.3 The CFS Product Line

Application Layer

Mission-specific application  ...  Reusable application  Reusable application  ...

Core Layer

Time Service  Event Service  Software Bus  ...

Figure 2-2: High-level structure of the CFS product line.

In fact, there is a many-to-many mapping between the business goals of the CFS and the architectural decisions that were made (see Figure 2-3). Thus, in order to meet the business goals, it is vital to verify that the implementation is consistent with the architecture rules, and resolve discrepancies, if any.

The CFS Business Goals

- Reduce time to deploy high-quality software
- Reduce project schedule and cost uncertainty
- Directly facilitate formalized software reuse
- Enable collaboration across organizations
- Simplify sustaining engineering and maintenance
- Platform for advanced concepts and prototyping
- Common standards and tools across NASA

Supporting the Goals

- Layered architecture
- Reusable core modules with standard API’s
- Plug and play modules
- Publish-and-subscribe style
- Standard Middleware/ Software bus
- Run-time module registration and integration
- OS and Hardware Abstraction

Figure 2-3: CFS business goals and architecture decisions.

The interest in the CFS has been spreading fast within the aerospace community. For example, Lunar Reconnaissance Orbiter (LRO), Global Precipitation Measurement (GPM), Magnetospheric Multi-Scale (MMS) at NASA GSFC, Radiation Belt Storm Probes (RBSP) at Johns Hopkins University/Applied Physics Laboratory (JHU/APL), and Lunar Atmosphere and Dust Environment Explorer (LADEE) at NASA Ames, are all using the CFS, and many more are expected in the near future. The CFS relies on
project and project-independent funding. For example, the GPM project is helping to fund the CFS application modules development. Future missions are expected to contribute to new capabilities development as well as to support CFS sustaining engineering.

2.4 Verification of Architectural Rules

2.4.1 General Process for Verification

The verification process involves two teams: the CFS team and the Fraunhofer team. The process started in early 2008 and has been ongoing for more than one year. In order to perform this verification, the CFS team sends the requirement specification, the developer’s and deployment guide, and the source code bundle to the Fraunhofer team. All core modules and the application modules developed by the CFS team are part of the source code bundle. Figure 2-4 shows an example CFS context, showing the inter-module communication via the software bus.

![CFS example context](image)

Figure 2-4: The CFS example context.

The Fraunhofer team then performs an independent verification of the architectural rules using their reverse engineering methods and tools. After the analysis, both teams get together for detailed discussions on the results of verification. These meetings often lead to follow-up analysis as new questions arise. After each meeting, the CFS team addresses the high-priority architectural issues, and the Fraunhofer team prepares answers for the additional questions raised by the CFS team. This process is repeated periodically based on the progress of the CFS development, for example, after the implementation of new application modules. This verification task is non-intrusive and non-biased, because on the one hand it does not affect the CFS team’s development process, and on the other hand the analysis is independently performed by an external organization (i.e., the Fraunhofer team).

2.4.2 Overview of the Approach

The steps for verifying the architectural rules are depicted in Figure 2-5. Here, these four major steps are briefly explained.
Step 1 - Derive Architectural rules: The CFS requirement specification describes the functional requirements addressed by the core modules. The application guide describes, in detail, the APIs of the core modules. Furthermore, the way the core modules should be used by application modules are also described in this document. In addition, the application guide contains samples demonstrating how to develop and integrate new application modules with the core modules. The deployment guide explains how to deploy the CFS for individual missions. From these documents, it is possible to derive a number of architecturally-significant rules. This manual step has to be conducted only once since the architectural rules do not change frequently as compared to, for example, source code.

A fragment of these rules and the related quality properties they address are placed in Table 2-1. This table was derived from the CFS documentation and discussions with the CFS team. Note that the association between rules and quality properties is applicable for the CFS and might differ for other contexts. The absence of a quality property for a rule does not necessarily imply that the attribute is irrelevant. Only the highly relevant quality properties are associated to each rule in the table. The output of this step is a collection of architectural rules. In the future, the CFS team will offer the collected architectural rules to teams that use the CFS in their missions. It is expected that this will create better awareness and a clarification of the relationship between rules and quality attributes.
Verifying Architectural Design Rules of a SPL

Table 2-1: Sample rules and related quality properties

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Questions</th>
<th>Quality properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency rules</td>
<td>1. Do generic modules depend on specific modules?</td>
<td>Runtime adaptability, Buildability,</td>
</tr>
<tr>
<td></td>
<td>2. Do modules bypass the OS and hardware abstraction layer?</td>
<td>Portability, Testability,</td>
</tr>
<tr>
<td></td>
<td>3. Do modules follow the publish-subscribe architectural style?</td>
<td>Performance</td>
</tr>
<tr>
<td>Decomposition Rules</td>
<td>1. Do modules directory structure adhere to the template structure?</td>
<td>Buildability, Comprehensionability</td>
</tr>
<tr>
<td></td>
<td>2. Do modules follow the decomposition guidelines?</td>
<td></td>
</tr>
<tr>
<td>Redundancy-restriction rules</td>
<td>Do modules copy-and-paste from other modules?</td>
<td>Maintainability</td>
</tr>
<tr>
<td>Miscellaneous rules</td>
<td>1. Do modules expose their internal details?</td>
<td>Changeability</td>
</tr>
<tr>
<td></td>
<td>2. Do modules implement necessary interfaces, for example the logging interface?</td>
<td></td>
</tr>
</tbody>
</table>

Step 2 – Map Architectural Concepts to Source Code Concepts: The derived architectural rules are abstract and not necessarily explicit in the source code. For instance, the concept of the OS abstraction layer is architectural, and the corresponding source code concepts need to be clarified for developers to be able to use them. The CFS documents provide such mapping, for example, they describe which directory of the CFS implements the OS abstraction layer.

Table 2-2: Mapping of architecture to source code

<table>
<thead>
<tr>
<th>Architecture Concept</th>
<th>Source Code Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>Directory</td>
</tr>
<tr>
<td>Modules</td>
<td>Sub-directory</td>
</tr>
<tr>
<td>Interface</td>
<td>Function Signature</td>
</tr>
</tbody>
</table>

Similarly, architectural concepts, such as application layer and core layer are also explained with a mapping to source code concepts, such as directories, files, and functions (see Table 2-2).

Step 3 – Verify Architectural Rules: In this step, the source code of the CFS is verified with respect to the architectural rules using the mapping defined in the previous step. Tool support is needed because a) the source code is too large for manual review, and b) whenever the source code changes, verification needs to be performed.
2.4 Verification of Architectural Rules

Again, a few tools are employed for verification. First, the Fraunhofer SAVE tool [275] and [250] has the capability to easily import the source code and extract code relations [50] and [215] as shown in Table 2-3.

Table 2-3: Sample relations extracted from code

<table>
<thead>
<tr>
<th>Relation Type</th>
<th>From</th>
<th>To</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Function</td>
<td>Function</td>
<td>The Call relation is from a caller to a callee.</td>
</tr>
<tr>
<td>Include</td>
<td>File</td>
<td>Header File</td>
<td>The Include relation is between a file and a header file.</td>
</tr>
<tr>
<td>Refer_Variable</td>
<td>Function</td>
<td>Variable</td>
<td>The Refer_Variable is between a function and a global variable referenced by that function.</td>
</tr>
<tr>
<td>Part_of</td>
<td>Function, Variable, File</td>
<td>File, Directory</td>
<td>The Part_of relation models the hierarchical decomposition. For example, functions are part of files, whereas files are part of directories.</td>
</tr>
</tbody>
</table>

The tool offers a GUI to define the mapping and graphical editors for specifying the architectural rules. Using regular expressions, it is possible to define a map, for example, that all files under cfs-apps directory are part of the application layer. Other tools, such as the Relation Partition Algebra (RPA) tool [78], [233] are used to complement the SAVE tool, for example to verify rules related to interface usages and module visibility-restrictions. An overview of RPA is given in Appendix B. The output of this verification step is a collection of inconsistencies between the architectural rules and the source code.

**Step 4 – Analyze and Resolve:** The focus of this step is to analyze and resolve the architectural deviations. Depending on the criticality of the deviation, these deviations are resolved at the source code-level. In certain cases, the deviations are exceptions to architectural rules, and need not be resolved. Often the exceptions are temporary and need updates later on in the process. For example, performance is sometimes a problem with a third party component which is not (yet) at the right speed. As soon as a new version is ready the "work around" is no longer necessary and the new component should be used and the exception could be solved. This process is iterated either when new rules are introduced or when the source code is changed.

**2.4.3 Module Dependency-Restriciton Rules**

This section presents a few dependency-restriction rules of the CFS product line.
2.4.3.1 Generic to Application Modules Dependencies

As mentioned above, the product line architecture of the CFS has two major layers, namely the application layer and the core layer where the application layer is supposed to use the core layer and not the other way around, similar to [75]. The core layer is developed for reuse in different missions. Conceptually, the application modules in the application layer need not be present in all missions. Any core module that uses application modules not only compromises the conceptual integrity of generic and specifics, but also the common look-and-feel of build rules (i.e., Makefile rules [79]) as build targets have to be adjusted accordingly. Thus, it is important to ensure that there are no dependencies introduced from the core layer to the application layer. Otherwise, it might be difficult to build and test the core layer independently of any missions. As shown in Figure 2-6, the cfs-core layer is being used by the cfs-apps layer in the implementation. Of course, the modules within the application layer and the core layer are allowed to have self-dependencies, as depicted with a loop in Figure 2-6.

![Figure 2-6: Actual dependencies from the app layer to the core.](image)

2.4.3.2 Application to Application Modules Dependencies

The CFS has been architected to allow runtime plug-in of modules, even after the launch of a mission. In order to support this capability, the CFS team used the publisher-subscriber architectural style. In the implementation of this style, it is imperative that the modules in the application layer do not depend on each other directly at compile time. In other words, if two modules need to interact, they should use the services of the software bus module, defined in the core layer.
Apart from the runtime adaptation capability, the CFS build process defines uniform build rules for compiling each module of the application layer into an executable. Also, the modules are designed so that they are tested independently of other modules. Currently, the CFS team has developed around 10 applications within the application layer. As shown in Figure 2-7, no two application modules are communicating directly in the implementation, with an exception of a utility module (cfs_lib) which is correctly being used directly. As shown in Figure 2-8, the applications indeed communicate using the software bus only. All modules register to the software bus and exchange messages by publishing and subscribing to appropriate messages. Thus, the static structure of the publish-subscribe architecture style is in place, enabling the run-adaptation of individual (e.g., new patches or updates) modules without restarting the whole CFS.

2.4.3.3 Dependencies on OS and Hardware Variants

One of the goals of the CFS is to support many different operating systems (e.g., VxWorks, RTEMS, and UNIX) and hardware variants (e.g., X86, PowerPC) because
they are expected to be needed by various missions. To address this need, an abstraction layer that encapsulates the underlying OS and hardware variants has been introduced. A common API for all these variants is documented in detail [219]. This API contains interfaces for using the file system, memory, and network. All the applications and core modules should be agnostic to the underlying OS and hardware. Thus, the architectural rule states that none of the modules should use the C libraries directly for accessing OS and hardware resources, and instead should go through the modules in the abstraction layer, for portability reasons. Thus, the developer must use the corresponding OSAL functions to ensure that the hardware characteristics associated with each memory address are properly taken care of. For example, attempting to write to EEPROM using the standard C function memcpy will fail. Using OS_MemCpy will succeed because the EEPROM will be configured for writing before the copy is performed. However, this rule is compromised by the core layer because it bypasses the OS abstraction layer (OSAL) and uses those C functions directly (see Figure 2-9). The RPA and Grep tools detected that the memset and the memcpy functions are used instead of OS_Memset and OS_Memcpy defined in the OSAL layer. The CFS team is currently fixing these violations.

![Diagram](image)

Figure 2-9: Bypassing the OS abstraction layer.

### 2.4.4 Module Redundancy-Restriction Rules

One of the goals of the CFS product line is to minimize redundancy in the source code to facilitate software evolution [201]. The CFS team believes that implementing a product line using clones (copy-and-paste) is, in general, not a sound strategy. Before the introduction of a product line, the flight software maintenance team needed to understand each heritage architecture in detail. This increased the maintenance cost and also the additional complexity due to the cloned variants. Furthermore, cloned code also increases the testing effort because the size of the test code also grows whenever functions are cloned. In addition, the source code of the CFS is also offered to some of its customers, and the presence of clones does not give a positive impression of the overall product line quality. The presence of clones also indicates that there are potential architectural design problems with respect to commonality and variability management [86]. According to the flight software team, implementing a product line using clones is not a good strategy in general, because after a few variants it is extremely difficult to keep track of multiple code versions and their evolutions. In
2.4 Verification of Architectural Rules

general, a product line made of clones will not be cost effective [27] and [106]. Thus, the CFS product line was verified with respect to the presence of code clones.

CloneFinder [57], a commercial clone finder tool, is used to locate clones. The collected clone data is at the file level. The clone finder tool outputs the detected clones as clone-groups. Each clone group contains a list of files together with line numbers of the clone. In order to analyze the collected clone data in a systematic way from an architecture perspective, the data is overlaid on the structural view of the CFS using the SAVE tool. This visualization helps analyzing clones hierarchically. That is, starting from the layer-level to module-level. The analysis shows that there are no clones between the application layer and the core layer, even though both the layers are developed by the same team. The analysis of clones within the application layer showed that one function is cloned in four application modules. Similarly, one function within the core layer is cloned in two core modules. These routines could be easily moved to utility modules. Intra-module clones were also analyzed. The analysis showed that there are clones inside the memory management (MM) module of the application layer. Analysis of these clones revealed that there are only small differences between the implementations of eight, sixteen, and thirty two bit memory addresses that could be abstracted.

The OS abstraction layer (OSAL) contains clones. This layer implements the common API [219] and consists of 150 functions for different operating systems, namely VxWorks, RTEMS, Mac OSX, and Linux. The clone analysis showed that 14 functions are copied in all four OS variants. For instance, functions such as get task id, get queue id, get semaphore id by the given name are duplicated. Another 13 functions are copied among the three OS variants. In addition, there is an overall high similarity between the Mac OSX and Linux implementations of the API, supported by the evidence that around 25 functions are copied-and-pasted between these two OS variants.

Table 2-4 summarizes the clone measurement data. False-positives are reported as clones, but are not really clones. For example, functions for read and write operations on a file would look the same if “read” is replaced by “write”.

<table>
<thead>
<tr>
<th>Layer</th>
<th># of Clone Group</th>
<th>True-Positives</th>
<th>False-Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>48</td>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>Core</td>
<td>29</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>OSAL</td>
<td>69</td>
<td>60</td>
<td>9</td>
</tr>
</tbody>
</table>

Automatic clone finder tools have no knowledge of the software architecture and falsely report such functions as clones. The architecture of the CFS has been designed to have a common look-and-feel among modules, and the source code is manually developed based on a template. For example, in a publish-subscribe architectural style with a software bus (SB), each application is responsible for “boiler-plate” code such as publishing messages on the SB, subscribing and receiving messages from the SB, unsubscribing to previously subscribed messages, handling errors due to the SB, to
name a few. Because of the software architecture, such “boiler-plate” code is difficult to avoid in the source code. The CFS team was considering code generators that can produce template code for each application that interacts with the SB. Many of the false-positives were due to the “boiler-plate” code. After manual analysis of the clone data, such clone pairs were filtered out, resulting in a remaining set of true-positives. There are just a few true clones in the application and the core layer, and most of them are intra-module clones. A high number of true-positive clones are present in the OSAL.

Many clones in the OSAL are due to the differences in the API signature among OS types. For example, consider the OSAL implementations for deleting a task under the VxWorks and the RTEMS OSes, see Figure 2-10 and Figure 2-11. These two implementations for deleting a task differ because, for example, to delete a task under the VxWorks the implementation calls the taskExit function, whereas the corresponding implementation calls the rtems_task_delete function under the RTEMS.

```
void OS_TaskExit()
{
    uint32 task_id;

    task_id = OS_TaskGetId();

    semTake(&(OS_task_table_sem));

    OS_task_table[task_id].free = TRUE;
    OS_task_table[task_id].id = UNINITIALIZED;
    strcpy(OS_task_table[task_id].name, "")
    OS_task_table[task_id].creator = UNINITIALIZED;
    OS_task_table[task_id].stack_size = UNINITIALIZED;
    OS_task_table[task_id].priority = UNINITIALIZED;

    semGive(&(OS_task_table_sem));

    taskExit(OS_SUCCESS);
}
```

Figure 2-10: The wrapper function to exit a task under the VxWorks OS.

We do not consider the OS_TaskExit functions (see Figure 2-11 and Figure 2-10) as critical clones because it is very difficult to abstract the differences due to API signatures between different OS types. There is not much value in abstracting such clones. On the other hand, it appears that the OSAL could benefit from moving the common OS wrapper functions that are exactly the same among the OSAL implementations into a single folder. However, this would require changes in the build process to compile both common functions as well as the functions that are specific to the selected OS type. To sum up, the CFS implementation has very few clones in the
application and the core layers, and the ones that do exist were investigated by the CFS team.

```
void OS_TaskExit()
{
    uint32 task_id;
    task_id = OS_TaskGetId();

    sem_wait( &(OS_task_table_sem));

    OS_task_table[task_id].free = TRUE;
    OS_task_table[task_id].id = UNINITIALIZED;
    strcpy(OS_task_table[task_id].name, "")
    OS_task_table[task_id].creator = UNINITIALIZED;
    OS_task_table[task_id].stack_size = UNINITIALIZED;
    OS_task_table[task_id].priority = UNINITIALIZED;

    sem_post( &(OS_task_table_sem));

    rtems_task_delete(RTEMS_SELF);
}
```

Figure 2-11: The wrapper function to exit a task under the RTEMS OS.

### 2.4.5 Module Decomposition-Restriction Rules

As mentioned above, the CFS source code is delivered to missions who instantiate variation points by configuring the build process and macros defined in the header files of the CFS. In order to facilitate the configuration process, the developer’s guide offers rules related to decomposition of modules in the directory structure (i.e., the code architecture). For example, the guide specifies in which folder the mission-specific header files and module documents have to be present, including their naming conventions.

From the CFS development team point of view, if all modules share a uniform look-and-feel, it not only helps program comprehension and evolution, but also enables developers to easily get familiar with their colleagues’ implementations. Furthermore, test-suites and build scripts should also be organized in a similar way. The developer’s guide provides guidelines and templates related to the structure of modules and sub-modules in the application layer. Thus, the goal of our verification is to check whether the modules of the application layer are consistent with the CFS template.

Here, a few examples of verification results are presented. The CFS template specifies that all application modules should have the directory structure as follows: The application name should be the same as the name of the sub-directory. Each application should contain a directory with the name `fsw`, which in turn contains the `src`
directory, the mission_inc directory for configuring mission parameters, and the platform_inc directory for configuring platform parameters. All the application modules were verified with respect to this directory structure decomposition. The results indicate that all but one application module followed this rule (see Figure 2-12). This deviation is marked with a red cross, meaning that the mission_inc directory is not present in the sc application module.

Similarly, the template explains the pattern to be followed for externally visible interfaces of each application module. It also explains how the external interface should be implemented and decomposed internally using a pseudo application module called QQ. It is expected that each module follows the structure shown in Figure 2-13 (left), where QQ_AppMain is the only entry point to the module, and it calls QQ_AppInit, and so on as shown in Figure 2-13.

All modules were verified against the QQ template, and some violations were detected (see Figure 2-13 right). The red crosses show that the LC application module misses two routines, namely LC_VerifyCmdLength and LC_HouseKeepingCmd. Code
analysis shows that the actual functionality of these routines is in fact implemented but with different routine names. Refactoring [83] to maintain the common look-and-feel is being considered by the team.

2.4.6 Miscellaneous Rules and Analysis

This section summarizes the rules related to the visibility of the internals of individual modules, unused interfaces of core modules, and the usage of #ifdef/#ifndef/#if preprocessor symbols for the source code level variability management.

2.4.6.1 Module Internals Visibility-Restiction Rules

Since the CFS team offers the source code of the application and core modules to missions, it is very important to restrict the visibility of the internal details of individual modules. Otherwise, mission-developers might use the internals of such modules directly without using appropriate interfaces, and thus it might be difficult to later replace changed and updated CFS modules, without impacting mission-specific modules. Since the C language does not offer the concept of private and protected code elements (as in Java), the CFS developer’s guide offers coding rules. For example, one of the rules states that intra-module interfaces that are not to be used directly by mission specific applications should be declared in a header file with its name having the suffix “_priv”. Moreover, no publicly visible header file should include a private header file; otherwise the private details are still visible indirectly to other modules for use. Similarly, none of the higher-level layer interfaces should expose its lower-level interface. Using the RPA and the grep tools, the Include relation of the CFS source code were verified. The analysis showed that one of the core modules’ private header file was indirectly visible to external modules because a public header includes it. The CFS team is addressing this issue.

2.4.6.2 Core Modules Interface-Usage Analysis

The interfaces of the core modules of the CFS were developed after the commonality and variability analysis among the requirements of past missions. Thus, it is logical to expect that either all the public interfaces of the core modules are used by the application modules or the unwanted interfaces are conditionally removed (e.g., using preprocessor directives). The interface-usage analysis of the CFS core modules showed that some of the interfaces are neither used by the existing 10 application modules nor used by other core modules (see Table 2-5). The analysis showed that these unused core interfaces cannot be automatically removed, that is, there are no variation points to delete this unwanted functionality. The analysis also pointed out some redundancies in the interfaces of modules, implying that the service offered by an interface can also be obtained by using a combination of other interfaces. Such redundant interfaces could be easily removed to keep module interfaces minimal.
Table 2-5: Analysis of unused interfaces

<table>
<thead>
<tr>
<th>Core Module</th>
<th># of Offered Interfaces</th>
<th># of Unused Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Bus Services</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>Executive Services</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>Time Services</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>Table Services</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>File Services</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Event Services</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Essentially, unused interfaces and their implementations would remain as dead code, which is considered risky in the flight software domain because a) during testing it is difficult to demonstrate 100% code coverage due to dead code, and b) in the conservative world of flight software, stakeholders are concerned that the dead code could be accidentally activated and could cause malfunction [53]. In future analysis, mission-specific modules will be also included, and based on the feedback the CFS team will investigate ways to introduce variation points at the interface-level for deleting unwanted interfaces and implementations.

2.4.6.3 #ifdef/#ifndef/#if/elsif Complexity Analysis

The purpose of this analysis is to check how complex the implementation is with respect to the usage of conditional preprocessor directives (e.g., #ifdef, #ifndef, #if, and #elif statements) for implementing variation points. A variation point could be a binary value (e.g., Log is on or off), a numeric value, or a string value. Custom scripts were developed to measure the number of variation points per module. The measurement shows that there are around 150 variation points within the core layer and 125 in the application layer. The usage of variation points in conditional preprocessor directives was also measured using the ifnames tool, which emits the list of files where a preprocessor symbol is used [134].

The histogram (see Figure 2-14) shows that 80% of the CFS source files do not have any conditional preprocessor statements at all, excluding the guards for header files for avoiding multiple inclusions. A notable exception is one file that refers 79 variation points in a sequence of #if statements. Further analysis showed that the file validates the legal range of all variation points within a module. Overall, the use of conditional preprocessor statements has been very well under control in the CFS implementation which consists of 170 KLOC. The collected data was also used to measure cross-module interferences of variation points, which makes the source code very complex to test, comprehend, and evolve. The analysis showed that only 20 variation points out of 150 in the core layer where referred in the application layer, suggesting that the source code of application modules is using only a fraction of variation points of core modules.
Several researchers have investigated maintenance issues due to preprocessor statements (e.g., [76], [188], [265], [262], [17], [71], and [160]). They all concluded that programmers use preprocessor statements as the main implementation technique for handling configurations or variants of systems. We agree to the existing literature that preprocessors impede program understanding. We have analyzed around two dozen systems (see Appendix A) and most of them have extensive conditional preprocessor statements in contrast to the CFS. Note that our conclusion regarding the low usage of conditional preprocessor statement in the CFS is rather unusual in comparison to the existing literature.

In [41], Peter Brown wrote in the 1970s about the use of macro-processors to construct portable software using conditional compilation for targeting different platforms. Most of the systems we have come across heavily used preprocessor statements primarily to manage OS and hardware variants. In contrast, the CFS architecture has an abstract OS interface and alternative implementations for each OS type. Because of this architectural decision, the excessive usage of preprocessor statements is avoided, as shown in Figure 2-14. Thus, the key lesson is that the architecture should explicitly address variability; otherwise the source code would suffer from extensive usages of conditional preprocessor statements, which impedes maintenance.

Also, as expected by the architectural design of the CFS application layer, there is no interference of variation points from one application module to another, and the variation points of the application layer are not referred in the core layer. This measurement indicates that there is a clear separation of concerns in the implementation.

This analysis showed that variability is implemented in the CFS using several different strategies depending on the type of variability needed. The C preprocessor is used only for configuring mission-specific parameters and removing unwanted functionalities implemented in the core and optional modules. The CFS design uses tables (a collection of parameters that are loaded during runtime) as an integral part of its design. These parameters are used to configure applications during compile-time and runtime. One common API [219] and multiple implementations are developed to handle the operating system and hardware variants. The build process is designed to choose and compile the right implementation files for various operating system and hardware
variants. Missions could also build and distribute the CFS modules on different CPUs; the build process can be easily configured to select the mission-specific header files, and modules are not aware of their peer modules CPUs, as all communications go through the middleware (i.e., the Software Bus module).

### 2.5 Brief Summary of Results and Lessons Learned

Table 2-6 presents some statistics on the number of rules checked and detected violations. The first row, for example, shows that 12 different dependency-restriction rules were checked and out of those 3 rules were violated. Overall, 45 rules were checked and 14 violations were detected. Examples of detected issues were by-passing the OS abstraction layer, unexpected dependencies among modules, inter-module clones, exposing internal details of modules, redundant definitions of configuration parameters (i.e., those mentioned in #define statements) in multiple files, and a few inconsistent interface naming conventions. In addition, the verification identified a module that does not use a particular interface to release memory table resources, could result in subtle performance problems. The main reason why certain rules are violated is that some of the modules were reused from past missions and restructured to fit the CFS architecture, however, in the restructuring process the rules related to internal module structure did not get enough attention. At the time of finalizing the thesis, work was planned to apply automated source code restructuring techniques to resolve detected violations (e.g., [289] and [255]).

<table>
<thead>
<tr>
<th>Rule Type</th>
<th># of Rules Verified</th>
<th># of Rules Violated</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency</td>
<td>12</td>
<td>3</td>
<td>SAVE, RPA</td>
</tr>
<tr>
<td>Redundancy</td>
<td>6</td>
<td>3</td>
<td>Clonefinder, SAVE</td>
</tr>
<tr>
<td>Decomposition</td>
<td>5</td>
<td>3</td>
<td>RPA, SAVE</td>
</tr>
<tr>
<td>Visibility of Secrets</td>
<td>5</td>
<td>1</td>
<td>RPA, SAVE</td>
</tr>
<tr>
<td>Variability-point Interference</td>
<td>3</td>
<td>1</td>
<td>ifnames, SAVE</td>
</tr>
<tr>
<td>Interface Naming Conventions</td>
<td>15</td>
<td>3</td>
<td>RPA</td>
</tr>
</tbody>
</table>

Table 2-6 lists the set of tools used in this process. In order to support an architecture-centric analysis, the data collected using different tools are imported to the SAVE tool and visualized using hierarchical structural views. For example, the clone data collected from the CloneFinder tool is imported to SAVE for visualizing and analyzing clones, among layers, modules, and sub-modules. Configuration parameters and their usage collected by the ifnames tool are imported to SAVE for visualizing and analyzing the location of variation points across layers, modules, and sub-modules.
Lessons Learned

L1) Verifying the architectural rules helps connecting business goals, to architectural principles, and to the implementation. In this process, the teams that develop software for reuse and the teams that use the reusable software are made aware of how to develop the product line for reuse and how to reuse it in the right way.

L2) When measuring redundancy using automated clone detectors, it is worth spending effort in reviewing the detected clones. Clone detectors have no architectural knowledge, thus they might falsely report similar code patterns as clones. It is easy to upset the development team with wrong clone data.

L3) In some cases, it is not a big deal if pairs of functions are structurally similar to each other as shown in Figure 2-10 and Figure 2-11, for example. Such pairs of functions might either be impossible to abstract in an elegant way or it is difficult to justify the benefit of resolving such clone pairs.

L4) Overlaying the collected source code level data such as clones and number of variation points onto the structural view facilitates architecture-centric analysis by showing the “big-picture” first and then details.

L5) Quality properties are achieved using appropriate architectural principles [308]. For instance, runtime adaptability using patches can only be enabled if the implementation does not deviate from the intended architectural design. Hence, it is vital to verify the consistency between the architecture and implementation.

2.6 Closing Remarks

This chapter analyzed the CFS product line architecture by verifying that architectural rules related to the module architecture and the code architecture were indeed met in the implementation. Overall, 45 rules were checked and 14 violations were detected. It is worth noting that the CFS, a safety-critical software product line, undergoes extensive code reviews. Nevertheless, some of the detected violations escaped the manual review process. Thus, this tool-supported verification of architecturally-significant rules complements traditional inspections by finding additional issues. The method presented in this chapter is the first step towards the goal to define a method for analyzing and evaluating product line implementations from an architecture-centric perspective. This method will be applied to several product line implementations in the near future and will be improved based on the lessons learned. In addition, complementary analysis of rules that are outside the scope of this chapter will be part of the future research. For example, rules that deal with the behavior of the system, such as task scheduling, inter-task communication, and ordering of runtime events. Some of these behavioral rules are covered elsewhere in [52].
Chapter 3
An Analysis of Unit Tests of a SPL

3.1 Abstract

This chapter presents an analysis of the unit testing approach developed and used by the Core Flight Software System (CFS) product line team at the NASA Goddard Space Flight Center (GSFC). The goal of the analysis is to understand, review, and recommend strategies for improving the CFS’ existing unit testing infrastructure as well as to capture lessons learned and best practices that can be used by other software product line (SPL) teams for their unit testing. The results of the analysis show that the core and application modules of the CFS are unit tested in isolation using a stub framework developed by the CFS team. The application developers can unit test their code without waiting for the core modules to be completed, and vice versa. The analysis found that this unit testing approach incorporates many practical and useful solutions such as allowing for unit testing without requiring hardware and special OS features in-the-loop by defining stub implementations of dependent modules. These solutions are worth considering when deciding how to design the testing architecture for a SPL.

Keywords: Unit Testing, Stub, Metrics, Software Architecture, and Flight Software.

3.2 Introduction

Unit testing is often the first level of testing of a software system. Unit testing is performed by the developers on the smallest testable unit, which we hereafter will refer to as the module (e.g., one or more functions or procedures or classes) in part to match the vocabulary used by the CFS. Unit testing is motivated by the fact that the cost of finding and fixing a bug at the time of unit testing is many times cheaper than finding and fixing bugs that are found during integration testing, system testing or in the field. For example, Barry Boehm reported that early prevention efforts provided a 5:1 to 10:1 payoff at TRW Inc. [15]. In addition, unit tests facilitate regression testing whenever software changes because unit tests allow developers to check that they did not break existing functionality. However, unit testing is difficult in practice for reasons including a) the module under test (MUT) often depends on other modules, making it difficult to isolate and test it independently from other modules and b) the MUT may depend on unique features and functions provided by the operating system as well as on functions provided by specific hardware. These dependencies make it difficult to set up a

---

3 To appear in Science of Computer Programming – a special issue of invited papers of the International Conference on Software Product Line (SPLC 2010) [102]. The previous version of this chapter was listed as the second best paper out of 91 papers at the SPLC 2010.
controlled unit test environment where the MUT can be tested while guaranteeing that the reason for failed unit test cases can be found in the MUT and not in one of the modules, the OS, or the hardware it depends on.

In the context of a software product line (SPL) unit testing needs to address two major concerns: The first concern regards the capability to test individual reusable core modules without being dependent on the behavior of any other core module, which might not yet be implemented, might not support all possible scenarios that need to be tested, and/or might impede unit testing for other reasons. This capability is important because the goal of unit testing is to test an individual module and to produce early and quick feedback regarding the test results. Second, the capability to unit test individual application modules without any of the core modules or any of the other application modules being ready and running in a predictable and correct way. This capability enables the application developers to unit test their code in a controlled environment where all modules their code depends on, are guaranteed to behave in a controlled way and always return expected results. Thus, the testing is focused on the behavior of the MUT only and can be performed even if the dependent modules are not yet implemented.

These two concerns are vital to SPL organizations. Especially important is the fact that apart from developing reusable modules, the core team must typically also deliver a unit testing framework to the application team so that they can test their modules without the core modules. The core team must also demonstrate the quality of their unit tests to the application team in order to build confidence regarding the quality of the core modules. Furthermore, when the application teams configure parameters (e.g., features and modules to enable, maximum number of files to open, thresholds for timeouts, etc.), of core modules or when they modify the source code of core modules, the delivered core unit tests help the application team to validate the correctness of the software.

Because flight software is mission-critical, the flight software branch at the NASA Goddard Space Flight Center (GSFC) has developed a practical approach for unit testing of its Core Flight Software System (CFS), which is a SPL for flight software. The Lunar Renaissance Orbit (LRO) mission [191], that is at the time of finalizing this thesis orbiting the moon, is one successful instantiation of the CFS. The CFS is delivered to application developers together with a comprehensible unit testing framework in order to facilitate testing.

This chapter discloses the architecture of the unit testing strategy that is used in the CFS with the hope that other organizations may benefit from these ideas and concepts. The unit testing strategies described in this chapter are sufficiently general and therefore applicable to other SPLs. The central ideas of the unit test architecture include a) the ability to manipulate return codes and “pass-by-reference” arguments of functions that are defined in dependent modules. The return codes and arguments are determined by the function under test (FUT) depending on the need, and b) the ability to easily define so-called stub implementations so that application modules can be unit tested without the implementation of core modules.

In order to analyze the unit tests of the CFS, we define a straight-forward and effective set of analysis questions, which are answered with the help of modular structures and
metrics that are extracted from the production code and the unit test code (see Section 3.4). The results of the analysis show that there is a pattern for developing stubs and a consistent strategy for unit testing of each core and application module in isolation. Using code-level metrics, the analysis also identified some improvements of existing tests to make them easier to understand and evolve.

**Contributions of this chapter.** While the SPL community has a growing collection of articles related to modeling and managing variability [138], as discussed in the related work section, there are not many papers focusing on unit testing at the module level in the context of a SPL. To this end, we hope this chapter makes the following contributions:

1. An illustration of how to unit test the core and application modules of a SPL in isolation using stubs; the subject of Sections 3.5 and 3.6.
2. A set of characteristics of architectural design that facilitate or impede unit testing; the subject of Section 3.7.
3. Experiences and good practices for unit testing based on the analysis of the CFS unit testing strategy, summarized in Section 3.8.

**3.3 The CFS Product Line Architecture**

This section briefly introduces the CFS product line architecture as a context for understanding the architecture of unit tests introduced in the later sections of this chapter. Readers interested in business goals and heritage of the CFS are referred to Section 2.3.2 for details.

The CFS has a layered modular structure, see Figure 3-1. The top layer has a catalog of reusable mission independent modules (applications), which may be used in one or more missions. The second layer (the Core Flight Executive (cFE) services layer) is the core of the CFS and must be used in all missions. This core layer offers several services, for example, a software bus module for inter-application communication, and an executive service module, which manages the lifecycle of each application on the top level. The third layer consists of an OS abstraction layer (OSAL) which offers a common API (Application Programming Interface) for all operating systems supported by CFS (e.g., VxWorks, RTEMS, and UNIX). The OSAL is also released as an open source library [219].
The OSAL offers three types of APIs, namely Real Time Operating System APIs, File System APIs and Hardware APIs that abstract the underlying hardware using a hardware abstraction layer. The Real Time Operating System APIs cover functionality such as Tasks, Queues, Semaphores, Interrupts, etc. The File System API abstracts the file systems and has the ability to simulate multiple embedded file systems on a desktop computer for testing. The Hardware APIs allow port and memory based I/O access in order to provide a common way of accessing hardware resources. Similar to the OSAL, there is also a board support package layer (BSP) which loads the configured OS and boots the CFS.

The CFS team offers the cFE services and its lower layers, including a catalog of CFS applications that can be reused, to various missions both inside and outside the NASA. Each cFE core service is configurable by choosing the values for appropriate constants declared in the interface or header files of each service.

All CFS modules are fully implemented in the programming language C [150]. Each module has a set of C files with configuration parameters and public API functions declared in header files. There are dedicated makefiles [79] for each module, which compiles all its files and produces a shared object file. Later, all core modules’ shared object files are linked into one shared core library. Missions reuse this shared library and develop applications using the APIs offered by the core modules. Missions can also add their own application modules to the top level application layer. However, applications are not allowed to communicate with each other via API calls in order to preserve the built-in flexibility for creating new variants. For example, the CFS can, in a flexible way, substitute applications, and restart or remove problematic applications. Inter-application communication is instead conducted by passing messages through the software bus module of the core service layer. That is, applications communicate by subscribing to and publishing messages from the software bus.

It is the responsibility of the software bus to deliver messages to all subscribed applications. At run-time, applications can also unsubscribe to previously subscribed
messages by using the APIs of the software bus. Figure 3-2 shows the context diagram of the software bus. Each module (a bubble) runs in a separate task, and communicates with other modules by publishing and subscribing to messages using the software bus APIs. The software bus is an abstract connector built on top of OS queues in such a way that the applications are completely unaware of the low-level communication mechanism.

Figure 3-2: The context diagram of the software bus in the CFS.

As shown in Figure 3-2, a typical CFS-based NASA mission has three types of modules (see bubbles), namely the modules of the core service layer (marked with * in Figure 3-2) that are part of all missions, optional CFS applications that missions can reuse from the catalog, and mission-specific applications. The software bus module facilitates integration of different categories of modules. It frees the application modules from the burden of inter-module coordination concerns so that modules can focus on their functionality.

In Chapter 2, the CFS source code was analyzed with respect to its compliance to architectural rules. The detected violations of the architectural rules have by now been removed and released as part of a recent version of the CFS. The previous analysis concluded that the CFS implementation is indeed consistent with the specified architecture. That is, layering is in place, and all CFS applications communicate only using the software bus.

In the remaining sections, we focus on the CFS’ unit testing strategy. This chapter will give an in-depth demonstration of how we can make use of the SPL architecture to organize such unit testing.

### 3.4 Process for Reviewing Unit Tests

This section introduces the straight-forward method we followed for reviewing the unit tests of the CFS. The review was applied in an independent way: the CFS team provided the artifacts to the analysis team, which has not been involved in any way with the development or regular testing of CFS. The analysis team then used their reverse engineering and software architecture competency to review the unit tests of the CFS. The analysis team independently then built an understanding of how the unit
testing is performed and identified issues with the current practice. These issues were then presented to the CFS team in technical meetings with the CFS engineers, project leaders, test leaders, etc. The CFS teams then addressed the most important issues and updated relevant source files and unit tests. This process has been going on for 2 years at the time of finalizing the thesis and is reiterated whenever new releases of the CFS and its test suites have been developed.

The main goals of the review process are a) to obtain a good overview of the current state of the test suites, and b) to obtain better software by identifying and improving problematic areas. In order to do so, the review process attempted to formulate answers to the questions listed in Table 3-1. These questions were formulated based on the vast experiences of the analysis team with source code of more than two dozens of industrial systems, see Appendix A. The approach to answering the questions was based on extracting the modular structure of unit tests from the test code. In addition, a set of metrics such the size of the production code, the size of the test code, the number of stubs, and the size of stubs were also computed. After that, the extracted structure and metrics were analyzed to understand the strengths and weaknesses of unit tests and the architecture of the production code.

Table 3-1: Questions for reviewing the existing unit test code

<table>
<thead>
<tr>
<th>Question</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Can each core module be tested independently of all other core modules it uses?</td>
<td>To a) understand whether modules have unit tests, b) if there are architectural design issues that make unit testing hard.</td>
</tr>
<tr>
<td>2. Can each application module be unit tested without running the core modules it uses (or any other applications)?</td>
<td>To understand whether application modules can be unit tested while neither waiting for the core modules to be delivered nor changing the application code for testing, assuming the correctness of other modules it uses.</td>
</tr>
<tr>
<td>3. How are configuration parameters of each module being handled during unit testing?</td>
<td>To understand how to unit test the behavior with respect to each configuration parameter (e.g., maximum number of messages in the software bus).</td>
</tr>
<tr>
<td>4. How easy is it to create mock or stub implementations of dependent modules?</td>
<td>To understand how complex it is to set-up so-called mock or stub implementations of dependent modules. Ideally, mock implementations are simple and their return values are easy to manipulate to traverse all paths.</td>
</tr>
<tr>
<td>5. Can modules be unit tested without access to special hardware and/or OS?</td>
<td>To understand whether modules can be unit tested on standard desktop applications without requiring developers to access special hardware and real-time</td>
</tr>
<tr>
<td>6.</td>
<td>How easy it is to set-up the unit tests of a module?</td>
</tr>
<tr>
<td>7.</td>
<td>Are there dedicated test programs for each public function of a module?</td>
</tr>
<tr>
<td>8.</td>
<td>How lengthy and complex is each test program?</td>
</tr>
<tr>
<td>9.</td>
<td>How are the tests results collected and reported for further analysis?</td>
</tr>
<tr>
<td>10.</td>
<td>Is there a standard principle or pattern for unit testing?</td>
</tr>
</tbody>
</table>

The review process used for analyzing the unit tests has two main steps as shown in Figure 3.

### 3.4.1 The Data Extraction Step

This step involves parsing the existing source code and test suites to extract relations (e.g., Call relation and Include relation) and metrics (e.g., number of tests, number of stubs, LOC of tests). The relations and metrics are obtained automatically using parsers developed at Fraunhofer, and are stored in a relational format as databases tables. The makefiles are also needed for the analysis because a) they contain information related to compiler switches, preprocessor symbols, and header files, and b) they contain information related to which object file is linked with the other object files. The ifnames tool [134] is used to extract all conditional preprocessor symbols
(excluding header file guards), which are basically configuration parameters supported by the system. These configuration parameters are later used to understand how test suites handle them.

Figure 3-3: Two major steps in the analysis of unit test architecture.

3.4.2 The Analysis, Query, and Visualization Step

In this step, the extracted data is analyzed using SQL-like queries written based on the Relation Partition Algebra (RPA) toolkit. RPA supports relation and set theoretic operators for querying of the extracted data, see Appendix B for a brief overview of RPA. For example, it is possible to extract all functions defined in the source code that are not referenced by any of the test suites. RPA also supports reachability analysis. It is possible, for example, to identify all functions that are transitively (i.e., indirectly) tested.

While querying is useful to extract information, visualization is very powerful to reveal patterns in the structure of unit tests. Module level dependency diagrams and dependencies of test suites to source code modules were visualized using the Fraunhofer SAVE tool [187], whereas the call graph of test cases were visualized interactively using the Prefuse toolkit [236].

3.5 Unit Testing of Core Modules

In this section, first we present a brief overview of the implemented module dependency structure, which was extracted from the source code. Second, we present the structure of CFS’ unit tests. Third, we introduce the counter-value pattern used for unit testing each module in isolation using stubs. Fourth, we discuss limitations of the counter-value pattern and how these limitations were handled. Finally, we discuss a
few straight-forward metrics that were used for evaluating comprehensibility and maintainability of unit tests.

### 3.5.1 Module Dependency Structure of the Production Code

Figure 3-4 shows the dependencies between the core modules that were extracted from the source code. The Executive Services (ES) module is responsible for initializing all modules, create new tasks for all modules, or exit their execution. Similarly, all modules use the Software Bus to send and receive messages except for the File Service. The Event Services (EVS) module helps modules log important events, and thus all modules use the EVS module. The File Services (FS) module helps other modules write and read data from files. The Timing Services module helps with various timing-related tasks. The Table Services (TBL) module helps application layer modules register tables (i.e., similar to C structures) to share data with other application modules.

Unit tests described in this chapter are written by the developers of the MUT. Thus, these unit tests are also called white-box unit tests because they are devised to execute all lines of the MUT. Thus the test cases are dependent on the implementation and not only on the interfaces of the MUT, which would be called black-box unit testing.

![Diagram](image)

**Figure 3-4: Dependencies between core modules.**

### 3.5.2 Module Dependency Structure of the Test Code

We automatically extracted relations from the test code of each core module as described in Section 3.4. We visualized the extracted relations to understand how each module is being unit tested in isolation. We found that the unit tests of all modules adhere to the “big picture” shown in Figure 3-5. The test runner is the main function that executes the test cases one at a time. Each test case invokes the module under test, which in turn uses the stub implementations of functions defined in dependent modules.
3.5 Unit Testing of Core Modules

For example, Figure 3-5 shows an instance of the “big picture” for the unit tests of the Executive Services (ES) module. Arrows denote dependencies (e.g., function calls). Dotted arrows are dependencies established at link time. The ES module depends on the Event Services (EVS), the Software Bus (SB), the Time Services (Time), the Table Services (TBL), the File Services (FS) modules as well as on the OS and the Board Support Package (BSP). Instead of using the real implementations of dependent modules, corresponding stubs are used. The view in Figure 3-6 is consistent with the source code dependencies of the ES module shown in Figure 3-4.

![Diagram of unit tests for the ES module.](image)

**Figure 3-6: Structure of unit tests for the ES module.**

### 3.5.3 Designing Stubs using the Counter-Value Pattern

Using stubs during unit testing is well known in the literature. Common frameworks, such as JMock [141] for Java, TypeMock.NET [283] for C#, and GoogleMock [121] for C++, provide support to control the return value for each subsequent function call. The counter-value pattern, developed by the NASA CFS team, provides a subset of this functionality and is tailored to the C language. The test suite for a MUT uses stub implementations of functions defined in other modules in order to fully run each function of the MUT and in order to provide an environment that produces guaranteed results for each possible function call.

Let us look at an example of how to control return values of the functions of dependent modules. Consider the interface specification of the create pipe function of the software bus module (see Figure 3-7).
Figure 3-7: Interface specification of the create pipe function.

The original implementation returns one of the four possible return values: Success, Bad Argument, Max Pipes Met, and Pipe Creation Error. However, the original implementation is “heavy-weight” because it creates real queues using the OS abstraction layer. If we want to unit test a function defined in another module that uses the create pipe function, the developer or tester needs to be able to manipulate return values so that paths that depend on the return value can be tested. Such a stub of the create pipe function is shown in Figure 3-8. As we can see, it does not do too much in contrast to the original implementation. Nevertheless, it is remarkably useful from the testing point of view because of the capability it offers to control return values using the SB_CreatePipeRtn instance of the UT_SetRtn_t data structure, which is defined in Figure 3-9, with two member variables count and value. Each stub implementation has its own instance of the UT_SetRtn_t type. Developers manipulate the instance of this data type in order to make the stub behave exactly as they need for the test case they design. The stub implementation for each function returns a return value based on the state of the count initialized using this function.

Figure 3-8: The stub implementation of the create pipe function.
3.5 Unit Testing of Core Modules

Developers can force a function to return the specified value of interest during the first invocation. For example, Figure 3-10 shows how to force the create pipe function to return the **CFE_ERROR** as the return code the first time it is called.

```
// forces CreatePipe to return CFE_ERROR (-1)
UT_SetRtnCode(&SB_CreatePipeRtn, -1, 1);
```

Figure 3-10: Forcing the create pipe function to return **CFE_ERROR**.

Developers can also easily control the return code for path coverage. For example, suppose we want to force the create pipe function to return **CFE_SUCCESS** for the first invocation and return an error for the second invocation, all we need to do is just call the **UT_SetRtnCode** function as shown in Figure 3-11 in the test code.

```
// forces CreatePipe to return CFE_SUCCESS
UT_SetRtnCode(&SB_CreatePipeRtn, -1, 2);
```

Figure 3-11: Forcing a function to return success and error in a row.

### 3.5.4 Limitations of the Counter-Value Pattern

In this subsection, we discuss limitations of the counter-value pattern and how such limitations were addressed in the CFS.

The analysis found that a few stub implementations do not use the counter-value pattern for controlling return codes. One limitation of this pattern is that it is impossible to advise a stub to return three different values when called by the MUT three times in a consecutive fashion. The problem is that the **UT_SetRtnCode** would need to be called in between each call to **CFE_SB_RcvMsg**. However, the developer is not allowed to alter the production code, which makes it impossible to use the counter-value pattern. For example, let us consider the scenario of reading messages from the software bus in an infinite loop, similar to the code snippet under test shown in Figure 3-12. In order to test this code fragment the stub implementation of the **CFE_SB_RcvMsg** must be capable of returning return codes such as **CFE_SUCCESS**, **CFE_SB_TIMEOUT**, and **CFE_ERROR** in a row.
However, since this is not possible to do using the counter-value pattern, a different strategy must be used. So, in order to test the code of Figure 3-12, the stub implementation uses a static local variable (see NumRuns in Figure 3-13) for keeping track of the number of times it was called. In the C language, the static local variables are initialized only once. The stub will return CFE_SUCCESS, CFE_SB_TIME_OUT, and CFE_ERROR as return codes in a row. In the third run, the stub will reset the NumRuns to zero so that other functions that call the stub will be able to follow the sequence of return codes in a row. Although this stub is hard-coded, it is being reused for several test scenarios because all usages of the receive message function follow the structure as shown in Figure 3-12.

```c
while (Status == CFE_SUCCESS) {
    // Wait for the next Software Bus message.
    Status = CFE_SB_RcvMsg(&CFE_ES_TaskData.MsgPtr,
                           CFE_ES_TaskData.CmdPipe,
                           TimeOut);

    if (Status == CFE_SUCCESS) {
        // Process Software Bus message.
    } else if (Status == CFE_SB_TIME_OUT) {
        // Process time out error
        ...
    } else {
        // Process software bus error
        ...
    }
}
```

Figure 3-12: Code snippet for receiving messages from the software bus.
3.5 Unit Testing of Core Modules

The second limitation stems from the fact it is not possible to instruct the counter-value pattern to return the same error code (except for success) several times in a row. This is because stub implementations based on the counter-value pattern return success in all runs except the last run. However, there is probably no need to support that capability because it is often the case that the control flow stops after the first error. If the FUT does not terminate the control flow after the first error, then the control-value pattern is not a good fit for such functions. In such a scenario, one could use a static local variable strategy to return different return codes for different runs, similar to Figure 3-13.

In some cases, the FUT interacts with functions of dependent modules using “pass-by-reference” of arguments. This means that if the called function modifies such arguments, the changed arguments are passed back to the caller. In such cases, the stub implementation of the dependent function should be able to modify argument values as required by its callers. For example, in Figure 3-13, the BufPtr argument is a pointer to a structure with the two members: message id and message code. These two members of the structure are assigned the values of the two global variables that are part of the unit test stub, namely the UT_RcvMsgId and UT_RcvMsgCode. Unit tests manipulate these two global variables of the test stub using setters. Using this simple strategy, the tests can control how the values of arguments are modified to exercise different paths of the FUT.

```c
int32 CFE_SB_RcvMsg (CFE_SB_MsgPtr_t *BufPtr, CFE_SB_PipeId_t PipeId,
                    int32 TimeOut) {

    CFE_SB_Msg_t message;
    int32 status = CFE_SUCCESS;
    static int NumRuns = 0;

    if (NumRuns == 0){
        CFE_SB_SetMsgId(&message, UT_RcvMsgId);
        CFE_SB_SetCmdCode(&message, UT_RcvMsgCode);

        *BufPtr = &message;
        NumRuns++;
    }else{
        if (NumRuns == 1){
            status = CFE_SB_TIME_OUT;
            NumRuns++;
        }else{
            if (NumRuns == 2){
                status = CFE_ERROR;
                NumRuns = 0;
            }
        }
    }
    return status;
}
```

Figure 3-13: A stub to generate different return codes in a row.
3.5.5 Metrics

To gain insight into the comprehensibility and maintainability of tests and stubs, we collected a few metrics from the source code and the test code. In this subsection, we will interpret these metrics.

3.5.5.1 Number of Stubs

We measured how many stubs are needed to test each module in isolation, see Table 3-2. The diagonals are “Not Applicable (NA)” because a module does not use its own stub. We found that most of the stub code is reused by several modules. Such a reuse of stubs is possible because of the way the core modules are designed and composed to work together. Each core module is used by other core modules via the (almost) same set of APIs. For example, the Software Bus (SB) module requires eleven stub functions of the Executive Service (ES) module. The Event Services module (EVS) requires ten stub functions of the ES module, out of the ten stubs nine of them are reused by the SB module. One conclusion from the analysis is that if a module is used by other modules via the same set of APIs then the stubs of the module are reusable during unit testing. Otherwise “too many” stubs have to be developed to test each module in isolation. Basically, to facilitate unit testing the architecture should be designed in such a way that each module is used by other modules via a standardized set of APIs.

<table>
<thead>
<tr>
<th>Module</th>
<th>Stub SB</th>
<th>Stub ES</th>
<th>Stub EVS</th>
<th>Stub Time</th>
<th>Stub TBL</th>
<th>Stub FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>NA</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ES</td>
<td>10</td>
<td>NA</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>EVS</td>
<td>8</td>
<td>10</td>
<td>NA</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Time</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>NA</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TBL</td>
<td>9</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>FS</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>NA</td>
</tr>
</tbody>
</table>

3.5.5.2 Size of the Production Code and the Stub Code

Table 3-3 compares the number of lines of code in the production code vs. the stub code. As expected, the stubs are indeed lightweight in comparison to the production code. For example, the table service module (TBL) has only fifteen lines of stub code compared to 3325 lines of production code. One reason for the limited need of stub functionality is that only the Executive Service uses the TBL module, and it only uses one TBL function, recall Figure 3-4.
Table 3-3: LOC comparison of the production code and the stub code

<table>
<thead>
<tr>
<th>Module</th>
<th>LOC</th>
<th>LOC of Stub</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>2032</td>
<td>225</td>
</tr>
<tr>
<td>ES</td>
<td>6126</td>
<td>257</td>
</tr>
<tr>
<td>EVS</td>
<td>1916</td>
<td>91</td>
</tr>
<tr>
<td>Time</td>
<td>2377</td>
<td>33</td>
</tr>
<tr>
<td>TBL</td>
<td>3325</td>
<td>15</td>
</tr>
<tr>
<td>FS</td>
<td>1057</td>
<td>84</td>
</tr>
</tbody>
</table>

Using the extracted code relations that were introduced in Section 3.4, we measured test coverage at the function level. For each core module, we measured the number of public APIs and the number of tests that invoke them. Table 3-4 shows that there are dedicated test programs for each public function of each core module with two exceptions: two ES functions and a TBL function have no unit tests. Further analysis revealed that one of the two ES functions has no return type (i.e., void) and no arguments (i.e., void), and it is a oneliner function that changes the state of an internal global variable. The other two functions are not single liners. However, all these functions are invoked by other public functions from their own modules. If something goes wrong in these functions, they will propagate the error upwards. Ideally, these functions also would have dedicated tests so that error localization is much easier than otherwise.

Table 3-4: Interface coverage by unit tests

<table>
<thead>
<tr>
<th>Core Module</th>
<th># of Functions in Interface</th>
<th># Directly invoked in Unit Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ES</td>
<td>33</td>
<td>31</td>
</tr>
<tr>
<td>EVS</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Time</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>TBL</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>FS</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

In order to gain insight into how internal functions are unit tested, we measured the number of functions that are defined for each core module and the number of tests that directly invoke them. Table 3-5 shows that not all internal functions are directly tested. However, further analysis showed that all internal functions are transitively (i.e., indirectly) tested using the test programs of the public APIs.
Table 3-5: Number of functions unit tested directly

<table>
<thead>
<tr>
<th>Core Module</th>
<th># of Functions Defined</th>
<th># Directly invoked in Unit Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>86</td>
<td>45</td>
</tr>
<tr>
<td>ES</td>
<td>117</td>
<td>68</td>
</tr>
<tr>
<td>EVS</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>Time</td>
<td>72</td>
<td>42</td>
</tr>
<tr>
<td>TBL</td>
<td>60</td>
<td>41</td>
</tr>
<tr>
<td>FS</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

3.5.5.3 Size of the Production Code vs. the Test Code

Just like the production code, the test code also must be maintained. To gain better insight regarding maintenance and complexity of the test code, we measured the lines of code (excluding comments and empty lines) for both the production and the test code for each core module, see Table 3-6.

Table 3-6: LOC comparison of the production code and the test code

<table>
<thead>
<tr>
<th>Module</th>
<th>Production LOC</th>
<th>Test LOC</th>
<th>Ratio of Test and Production LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES</td>
<td>6126</td>
<td>1406</td>
<td>0.23</td>
</tr>
<tr>
<td>EVS</td>
<td>1916</td>
<td>430</td>
<td>0.22</td>
</tr>
<tr>
<td>SB</td>
<td>2032</td>
<td>4365</td>
<td>2.15</td>
</tr>
<tr>
<td>FS</td>
<td>1057</td>
<td>166</td>
<td>0.16</td>
</tr>
<tr>
<td>TBL</td>
<td>3325</td>
<td>2555</td>
<td>0.77</td>
</tr>
<tr>
<td>Time</td>
<td>2377</td>
<td>586</td>
<td>0.25</td>
</tr>
<tr>
<td>Total</td>
<td>16833</td>
<td>9508</td>
<td>0.56</td>
</tr>
</tbody>
</table>

One conclusion from this data is that the production code can be unit tested with tests that are half the size, in lines of code, compared to the production code. Note that the precondition of this conclusion assumes that the LOC measurement takes into account all functions of the MUT even if they are not directly tested. Let us recall the fact that many of the internal functions are indirectly tested using test cases developed for externally visible functions. Thus, there are no dedicated test cases for many internal functions, resulting in a low ratio between the number of lines of the test code and the production code.

We also measured the LOC of the production code after excluding those functions that were not directly tested. In Table 3-7, Stubs refers to the set of stub modules of functions defined in the core layer and the OSAL layer. We treated stub modules as part of the test code base.
3.5 Unit Testing of Core Modules

Table 3-7: LOC of directly tested production code and the test code

<table>
<thead>
<tr>
<th>Module</th>
<th>LOC of Directly Tested Functions</th>
<th>LOC Test Code</th>
<th>Ratio of Test and Production LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES</td>
<td>3373</td>
<td>1406</td>
<td>0.42</td>
</tr>
<tr>
<td>EVS</td>
<td>311</td>
<td>430</td>
<td>1.38</td>
</tr>
<tr>
<td>SB</td>
<td>904</td>
<td>4365</td>
<td>4.83</td>
</tr>
<tr>
<td>FS</td>
<td>226</td>
<td>166</td>
<td>0.73</td>
</tr>
<tr>
<td>TBL</td>
<td>2560</td>
<td>2555</td>
<td>1.00</td>
</tr>
<tr>
<td>Time</td>
<td>1165</td>
<td>586</td>
<td>0.50</td>
</tr>
<tr>
<td>Stubs</td>
<td>NA</td>
<td>1383</td>
<td>NA</td>
</tr>
<tr>
<td>Total</td>
<td>8539</td>
<td>10891</td>
<td>1.28</td>
</tr>
</tbody>
</table>

An outlier in Table 3-6 and Table 3-7 is the software bus (SB) module which has much more test code than production code because of the services it provides and which deserves further explanation. The SB module provides services for sending and receiving inter-module messages. If a module wants to send a message, it uses the APIs of the SB module. If a module wants to receive a message of a particular type (or subject), the module first uses the SB module’s APIs to create a message pipe and then subscribe to the SB module by specifying the pipe id and the message type. If the SB module receives messages from publishers, it will route each message to the appropriate message pipes of subscribers.

One of the features of the SB module lies in the support for several configuration parameters, such as the maximum number of message pipes that can be created by each module, the depth of message pipes, the maximum length of each pipe name, etc. Thus, the SB module has to be tested for nominal and off-nominal behaviors related to these configuration parameters. Therefore, several test cases were developed by the CFS team to test off-nominal behaviors, such as deleting a message queue owned by some other module, unsubscribing to a message that was not subscribed before, unsubscribing more than once without an intermediate subscribe, etc. That is why the test code size of the critical SB module is almost five times than that of the production code.

The overall conclusion is that there is more code needed for unit testing than there is production code. However, it is important to remember that the measurement data takes into account the lines of code of only the directly tested functions with dedicated test cases for them. In discussions with the CFS team they conclude that in their experience the size of the unit test code is almost always greater than the size of the production code. However, there are differences between individual engineers’ approach to unit testing. Some engineers take more time to design their unit tests and others take a “brute force” approach and often cut and paste similar test cases rather than create modular tests.

3.5.5.4 Size of unit test functions

We visualized the extracted Call relation of the unit test functions of each core module and measure the LOC of unit test functions. We observed that, in contrast to the software bus module, the test cases of other modules are less modular in the sense that
unit tests of other modules tend to mix test cases of nominal behavior with off-nominal behavior (e.g., deleting a task before creating one). This is one of the reasons for a low ratio between the test code and the production code because test initialization code is reused for several scenarios. For example, all tests of the executive module (ES) are squeezed into eleven test functions, as shown in Figure 3-14. Some of the tests are lengthy because several test scenarios are covered within one function. The CFS team acknowledges the fact that it is better to have several small tests instead of a few very large tests during the debugging process.

To sum up, with the help of code relations and metrics that were collected from both the production code and the test code, we analyzed how modules were unit tested in isolation and identified issues related to modularity of unit tests. These issues were reviewed by the CFS team and were added to the change request database.

3.6 Unit Testing of Application Modules

This section introduces key ideas of the Unit Testing Framework (UTF), developed by the CFS team, for testing the modules of the application layer without the need for running the core layer as well as the OSAL layer. This section addresses questions number: 2, 5, and 6 in Table 3-1.

In Section 3.3, we noted that all CFS applications use the software bus (defined in the core) to communicate, and they all follow the principles of the publisher-subscriber architectural style. In order to unit test modules of each CFS application, stub implementations must be created by the core team so that the application team can easily validate their code. This section briefly introduces the central ideas of the Unit Test Framework (UTF) developed by the CFS team. Some concrete code snippets are used to explain how the concepts such as publishing, subscribing to messages can be easily simulated using this UTF framework during unit testing, without running the software bus and all layers below it, recall Figure 3-1.
3.6.1 The Unit Testing Framework (UTF)

The UTF is a tool to be used by application developers when they unit test their application code. It allows them to exercise all logical paths in their application code in an environment which is independent of flight hardware and the Real Time Operating System. Such an environment is desirable because access to flight hardware can be limited and because RTOS’s tend not to have easy-to-use tools such as debuggers which can be beneficial to verifying that correct paths are executed. In addition, the UTF’s software environment provides the ability to simulate hardware events which would be difficult or impossible to simulate while running on the operational hardware and RTOS.

One of the core ideas of the UTF framework is the so-called function hook, which allows application developers to write custom code of the functions defined in the core. For those API calls where the need for customization has been identified, the UTF provides the capability for the developer to associate a customized function with a particular API which will generate return values. Consider the source code snippet below. It was defined in an application module, which uses the software bus module function (CFE_SB_Subscribe). In order to unit test this code and exercise all paths, the CFE_SB_Subscribe function must be forced to return success the first time, the third time, and the fifth time, and so on, and return an error on the second time, the fourth time, and so on.

```c
int32 Result;
Result = CFE_SB_Subscribe(...);
if (Result != CFE_SUCCESS) {
    ...
    return Result;
}
Result = CFE_SB_Subscribe(...);
if (Result != CFE_SUCCESS) {
    ...
    return Result;
}
```

Figure 3-15: Calling the same function two times.

In order to exercise the path in which the second call to CFE_SB_Subscribe failed, the tester must associate a function with the CFE_SB_Subscribe to return CFE_SUCCESS (or 0) the first time, and a non-zero value the second time. The tester wants to emulate the behavior such as having the CFE_SB_Subscribe API alternately return CFE_SUCCESS and CFE_SB_ERROR (see Figure 3-16).
In order to redirect all calls to the original CFE_SB_Subscribe by the above hook, the tester has a simple API as shown below: In the test driver, the tester has to invoke the UTF utility which links the customized routine with the cFE API, see Figure 3-17 in that the first argument is the index of the subscribe function and the second argument is the function pointer to the hook implementation of the subscribe function.

```c
int32 CFE_SB_SubscribeHook(...) {
    static uint32 Count = 0;
    if (Count++ % 2 == 0)
        return(CFE_SUCCESS);
    else
        return(CFE_SB_ERROR);
}
```

Figure 3-16: Hook function returning success and error in a row.

In the test driver, the tester has to invoke the UTF utility which links the customized routine with the cFE API, see Figure 3-17 in that the first argument is the index of the subscribe function and the second argument is the function pointer to the hook implementation of the subscribe function.

```c
UTF_SB_set_function_hook(CFE_SB_SUBSCRIBE_HOOK,
    (void *)&CFE_SB_SubscribeHook);
```

Figure 3-17: Redirecting function calls to the hook function for testing.

The CFS team has developed a user manual explaining for each public core function, the behavior of the corresponding stub, see Table 3-8 for samples. Figure 3-15 shows the UTF implementation of the CFE_SB_Subscribe. The UTF provides wrapper functions for all cFE API functions as well as for some core layers’ internal functions required by these APIs. The UTF also provides utility functions that allow the programmer to interact with these UTF wrappers. Testers can add function hooks as explained above allowing testing without access to special hardware, operating system, etc. This also illustrates the importance of developers to follow specified software architectural rules. If they follow the rules (e.g., programming to abstract interfaces), they can benefit from the features offered by the UTF. For example, the testers of application modules can easily use the lightweight software bus implementation that supports the capability to read messages from files (instead of running the software bus) in order to test different paths that depend on the types of messages, (see Table 3-8).
3.6 Unit Testing of Application Modules

Table 3-8: (Sample) The original behavior vs. the behavior for testing

<table>
<thead>
<tr>
<th>API Call</th>
<th>cFE Core Functionality</th>
<th>Unit Testing Framework Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFE_SB_Subscribe</td>
<td>Subscribes to the supplied message on the supplied pipe using default values for the Quality and message limit.</td>
<td>Does nothing. Returns the value of the hook function if set, the preset return code if set, or CFE_SUCCESS.</td>
</tr>
<tr>
<td>CFE_SB_SendMsg</td>
<td>Sends the supplied message on the software bus (SB).</td>
<td>Returns preset return code if set, or uses UTF_put_packet utility to send the message and returns CFE_SUCCESS.</td>
</tr>
<tr>
<td>CFE_SB_RcvMsg</td>
<td>Reads a message from the software bus using the supplied parameters.</td>
<td>Reads data using the UTF_get_data utility, then returns the preset return code if set.</td>
</tr>
</tbody>
</table>

A key point here is that using UTF, the testers of application modules do not have to wait for the real implementation of core modules to be available, but they can use the lightweight stub implementation offered by the core team and start testing in parallel to the development or testing of core modules. As demonstrated in the CFS case the capability to easily define hook-in functions is crucial especially if testers need to simulate hardware behaviors without installing the real hardware.

3.6.2 Brief Comparison of the UTF and the Counter-Value pattern

The UTF was developed after the counter-value pattern was developed. They both share a common objective of facilitating of unit-testing using stubs. However, there are some notable differences between them. In the context of the publish-subscribe architecture that is implemented using a software bus, unit testing of modules often requires the software bus to contain messages of interest. This set-up is difficult to simulate using the counter-value pattern. The UTF solves this problem, in that it is possible to pass real messages to the MUT by using utility functions that simulate message queues. The tester can even pass a text file of messages to the UTF, which will take care of passing the messages during execution of the MUT. These capabilities are beyond the scope of the counter-value pattern.

3.6.3 Answers to Analysis Questions

In Table 3-9, we present the answers and comments to the analysis questions defined in Section 3.4.

Table 3-9: Answers to the analysis questions

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer and Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Can each core module be tested independently of all other core modules it uses?</td>
<td>Yes. Because of the design of simple stubs, it only takes 3 minutes or so to run all the unit tests of the core modules.</td>
</tr>
<tr>
<td>2. Can each application module be unit tested without running the</td>
<td>Yes. The Unit Testing Framework (UTF) helps each individual application be tested independently,</td>
</tr>
<tr>
<td>Question</td>
<td>Answer</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>core or other application modules it uses?</td>
<td>using the function hook concepts and lightweight stub implementations of core modules.</td>
</tr>
<tr>
<td>3. How are configuration parameters of each module being handled during unit testing?</td>
<td>There is a unit test program for each configuration parameter that checks the behavior for upper and lower bound constraints. Some of the internal details of a module are used to support unit testing of configuration boundaries.</td>
</tr>
<tr>
<td>4. How easy is it to create mock or stub implementations of dependent modules?</td>
<td>Mock or stub implementations are easy to create. At link-time, a module can be linked to one or more stubs of dependent modules. This capability supports incremental integration too.</td>
</tr>
<tr>
<td>5. Can modules be unit tested without access to special hardware and/or OS?</td>
<td>Yes. Programmers/testers do not need to go to the test lab for unit testing, and can test in their desktop environment. The UTF framework provides simulators with the same API as the original code.</td>
</tr>
<tr>
<td>6. How easy it is to set-up the unit tests of a module?</td>
<td>Just a couple of instructions are needed to set-up a test program.</td>
</tr>
<tr>
<td>7. Are there dedicated test programs for each public function of a module?</td>
<td>Yes, all interfaces have one or more dedicated unit test programs.</td>
</tr>
<tr>
<td>8. How lengthy and complex is each test program?</td>
<td>Some are lengthy (~100 lines including comments and blank lines) because they test more than one scenario and could be split into smaller ones. Some are complex because the function under test returns the same return code from multiple paths requiring extra test code.</td>
</tr>
<tr>
<td>9. How are the tests results collected and reported for further analysis?</td>
<td>Currently, they use the gcov (GNU coverage) tool. It reports line coverage. All test failures are reported in a text file that is manually reviewed by the tester.</td>
</tr>
<tr>
<td>10. Is there a standard principle or pattern for unit testing?</td>
<td>Yes. All core modules consistently use the concept of stubs to do unit testing. Also, all test makefiles for test suites share the same structure.</td>
</tr>
</tbody>
</table>
3.6.4 Issues Beyond the Scope of Unit Testing

While having a good set of unit test helps to validate the correctness of individual modules, there are several issues beyond the scope of unit testing in the context of SPLs. For example:

1. All unit tests in the CFS have been designed to run in a single thread. Thus, detecting multithreading issues are not covered by these unit tests and thus must be complemented by other means of testing, e.g., in integration testing.

2. In the publish-subscribe architectural style, the behavior of a publisher with one or more subscribers should be tested. However, in the unit tests at any given point of time only either a publisher or a subscriber is running, and not both of them. Thus, lost messages, duplicated messages, timing problems etc. are difficult to detect during unit testing. We are currently applying the architectural analysis method proposed in [103] to detect such issues related to the publish-subscribe style.

3. The behavior of the software bus (or other modules) during a heavy traffic situation is not simulated during unit tests. Thus, it is difficult to find timing problems and other issues related to the characteristics and behavior of the message buffer. In fact, the CFS team occasionally encounters such issues despite conducting rigorous performance, load, and stress testing.

4. Only the upper and lower limits for each configuration parameter are unit tested independently of other configuration parameters. Thus it is difficult to predict how the system will behave for different combinations of values. For example, what is the implication of the maximum number of messages in the software bus to the number of application modules in use? This is a much deeper question prompting the research community to understand configuration spaces spanned by the configuration parameters from the source code point of view.

5. Runtime dynamic reconfiguration is one of the core features of the CFS. From unit testing alone, it is not easy to understand how the whole system behaves if a module is substituted by a new module at run time.

Despite following a systematic approach for unit testing using the software architecture of the CFS and stubs, some of the team members feel that maintenance of unit tests is costly in part because of the level of involvement required by a software maintenance person. For example, if the implementation changes then typically unit tests and stubs will need to change. At the time of finalizing this thesis, research was under way to automatically derive unit tests from a state machine model of API specifications. Initial results of automated model-based test generation for the OSAL layer of the CFS are available in [104] and [90].

3.7 SPL Architecture and Testability

In this section, we discuss characteristics of the SPL architecture that facilitate or impede unit testing using the analysis of the CFS unit tests.
3.7.1 Some Design decisions that Facilitate Unit Testing

This review of unit tests has resulted in some insights related to the influence of SPL architecture on unit testing. Here, SPL specific examples from the CFS analysis are presented.

The key to flexible unit testing is programming to abstract interfaces. For example, in the case of CFS the core layer is designed and implemented in such a way that it is completely agnostic to the OS, hardware, and board support packages. More concretely, consider this simple case of creating a queue, and sending and receiving messages using the queue. Naturally, different OSes offer different queue APIs. If the system is programmed with a hard binding to the OS-specific APIs then it is, of course, very difficult to unit test such a system because we need access to special OS features, which may not be present on all developers’ machines. In addition, a hard binding to the OS-specific APIs would require different sets of unit tests to be developed for each OS type. Otherwise, the correctness of the modules cannot be demonstrated for all OS types. In order to avoid these issues, the CFS team introduced abstract interfaces with diversified implementations (see Figure 3-18).

The software bus (SB) module is programmed to these abstract interfaces, and the actual binding to a specific implementation occurs only at link time. One positive side effect of this architectural feature is that the test cases of the SB module are agnostic to the underlying OS type. Furthermore, the SB module can be tested with a lightweight stub implementation of OS APIs. This capability reduces the time taken to run the unit test cases of the SB module because of the lightweight nature of the stubs. Note also that this principle of programming to abstract interfaces should be applied to other resources, such as hardware sensors, COTS components, databases, and servers. If such resources are not abstracted, unit testing cannot be performed unless all of these resources are up and running.

A standardized interaction protocol between application and core modules facilitates testing. For example, in the CFS, all application modules use the core modules in the same fashion meaning that all application modules have common architectural behaviors such as all of them a) get started and initialized by a core module, b) create zero or more pipes, c) subscribe and/or publish messages using the core software bus module, and d) send and/or receive messages using the software bus. Because of such common architectural behaviors, it is possible to develop a unit testing...
strategy to test modules in isolation. If application modules interact with core modules in different styles, then devising a common testing strategy would be difficult.

### 3.7.2 Unit Testing needs Internal Details of Modules

In this subsection, we present two example scenarios to demonstrate that unit tests often require access to internal details of the MUT. The first scenario is about testing of boundary-values constraints of configuration parameters. The second scenario is about testing of certain off-nominal behaviors.

While hiding modules’ secrets is one of the fundamental principles of software engineering [130] this principle has to be weakened during development in order to write good unit tests. For example, consider the load library function snippet (see Figure 3-19), which loads the given shared library (LibName) and calls the function with the given name (EntryPoint). The CFE_ES_LoadLibrary is defined in the Executive Service module. The CFE_ES_MAX_LIBRARIES is a configuration parameter defined in a public header file that must be set to a particular value. This function should return an error code if it is called more than the number of times set during configuration. Note that it uses the CFE_ES_Global data structure for keeping track of the number of libraries that are already loaded. This data structure is hidden inside the ES module meaning that no other module is allowed to access this data structure. However, in order to test that this function will return an error if it is called more than the configured number of times, the unit test benefits from manipulating the CFE_ES_Global data structure, see Figure 3-20.

```c
int32 CFE_ES_LoadLibrary(char *EntryPoint, char *LibName, …) {
boolean LibSlotFound = FALSE;
for ( i = 0; i < CFE_ES_MAX_LIBRARIES; i++) {
  if ( CFE_ES_Global.LibTable[i].RecordUsed == FALSE ) {
    LibSlotFound = TRUE;
    break;
  }
}
if(LibSlotFound == FALSE) return CFE_ES_ERR_LOAD_LIB;
}
```

Figure 3-19: Code fragment for loading a library at runtime.

In case the tester has no knowledge of this private global data structure, then one strategy would be to repeatedly invoke the CFE_ES_LoadLibrary function one more time than the maximum number of libraries allowed. And then the tester could assert whether the expected error code is returned for the last call or not. One drawback of this strategy is that the tester should create necessary number of libraries to load, but testing could be performed strictly using public APIs. Another disadvantage is that it takes more time to run all tests using public APIs to simulate such a scenario.
Figure 3-20: Testing the upper bound of a configuration parameter.

However, we found some scenarios where the tester must have access to internal data structures and cannot unit test strictly through the public APIs only. Let us consider the delete pipe function, which takes the pipe id as the argument, defined in the software bus module. In order to delete a pipe, the caller of this function must be the owner of the pipe. If an invalid pipe owner tries to delete a pipe owned by someone else, then an error code should be returned. To test this scenario, the tester must manipulate the internal pipe-owner table, which is part of the MUT, in such a way that the caller id of the delete pipe function is different from the actual owner id of the pipe, see Figure 3-21. By default, the owner id of each test function is initialized to zero. Therefore, the manipulated pipe-owner table is now inconsistent with the real owner id. As a consequence, the delete pipe function would know that the caller is not the owner of the pipe and will return an error code. This example also demonstrates that if the tester has no knowledge of the design of the MUT, testing such scenarios would be very difficult or even impossible.

Figure 3-21: Test uses the internal data structure CFE_SB.PipeTbl.
The observation from these two examples is that unit testing sometimes requires access to internal data structures to transform the system into a desired state. Thus, many of the internal global variables have public visibility, but are declared in files with conventions such as \texttt{sb\_private.h} to warn developers not to access private data. Architectural rules were defined by the CFS team to make sure such publicly visible secret variables were not referenced by other modules using the architectural analysis approach presented in Chapter 2. This is an example of how the risk of relaxing some engineering principles can be mitigated by adding architecture/design rules. This example also shows that the C language does not have explicit language constructs neither to facilitate testing nor to avoid misusage of internal variables that were made public.

### 3.7.3 Some Design Issues that Make Unit Testing Difficult

Consider the code snippet used for sending a message on a software bus (see Figure 3-22). The first condition checks whether or not the input pointer is null. If the input is null, then this behavior is logged with an error id called \texttt{CFE\_SB\_SEND\_BAD\_ARG\_EID} using the event service module. The second condition checks whether or not the input is a valid message. If it is not a valid message, then this behavior is logged with an error id called \texttt{CFE\_SB\_SEND\_INV\_MSG\_EID} using the event service module. We see that this send message function returns the same return code \texttt{CFE\_SB\_BAD\_ARGUMENT} from two different conditional blocks for different error types. Because of this design, the unit testing code of this function becomes more complex than if unique return codes were used because it needs to determine exactly which one of the two code paths returned that value for path coverage.

```c
int32  CFE\_SB\_SendMsg(CFE\_SB\_Msg\_t *MsgPtr) {
    /* check input parameter */
    if(MsgPtr == NULL) {
        CFE\_EVS\_SendEventWithAppID(CFE\_SB\_SEND\_BAD\_ARG\_EID,...);
        return CFE\_SB\_BAD\_ARGUMENT;
    }
    MsgId = CFE\_SB\_GetMsgId(MsgPtr);
    /* validate the msgid in the message */
    if(CFE\_SB\_ValidateMsgId(MsgId) != CFE\_SUCCESS) {
        CFE\_EVS\_SendEventWithAppID(CFE\_SB\_SEND\_INV\_MSG\_EID,...);
        return CFE\_SB\_BAD\_ARGUMENT;
    }
    ... }  
```

Figure 3-22: The same return code is used for different error types.

In addition, the test code has to check whether or not the intended error event type was sent if something is wrong in the input parameter list. Thus, it calls the stub implementation of the send event function to make sure the right event type was sent. The stub implementation of the \texttt{CFE\_EVS\_SendEventWithAppID} function keeps a
history of events that were sent by the FUT. The unit test checks with the history for the existence of the particular event type, see Figure 3-23. Using this test strategy, the test code can be certain that the intended path was taken by the particular test case. The drawback of this strategy is that when a new logging event is added or deleted to the FUT, the existing tests could fail because the number and type of events are different than expected by the test.

```c
void Test_SendMsg_NullPtr(void){
    ...
    ActRtn = CFE_SB_SendMsg(NULL);
    ExpRtn = CFE_SB_BAD_ARGUMENT;
    if(ActRtn != ExpRtn) {
        TestStat = CFE_FAIL;
    }
    ExpRtn = 1;
    ActRtn = UT_GetNumEventsSent();
    if(ActRtn != ExpRtn) {
        TestStat = CFE_FAIL;
    }
    if(UT_EventsInHistory(CFE_SB_SEND_BAD_ARG_EID) == false) {
        TestStat = CFE_FAIL;
    }
}
```

Figure 3-23: Test code for function shown in Figure 3-22.

The review identified a few functions that suffer from this design limitation with respect to return values. These issues are currently being addressed by the CFS team. The recommended fix is to change such functions so that they all return a unique return value from each of its path, and thus make the unit testing code clearer.

To sum up, this section has proposed the idea of stubs and how they can be systematically designed for unit testing purposes. An in-depth discussion was offered to better understand how architectural design decisions facilitate or impede unit testing. Implications of data hiding on unit testing challenges are also highlighted.

### 3.8 Good Practices for Unit Testing and Analysis of SPLs

In this section, we discuss some good practices that were derived from the CFS analysis and which might be worth considering during unit testing and architecting in an SPL context.

- **Unit tests are an integral part of the product:** An important premise is that the unit tests are considered an integral part of the product, and versions of unit tests are managed just like any other source code in the configuration management system.
• **Test artifacts are also deliverables:** The unit tests as well as the stubs, the unit testing framework, and the test results are also deliverable to customers (i.e., the application developers) of the SPL. The main benefit is that the application developers are able to validate the core modules in their environment. Further, they need not wait for the core modules to be ready for unit testing of their modules in isolation.

• **Create many small tests instead of a few large tests:** Small tests will make it easier 1) to locate problems, 2) to maintain and evolve test cases, and 3) to characterize the purpose of test.

• **Test code needs naming conventions:** Similar to the source code, the test code should also follow naming conventions. For example, the file name of tests should reflect the file name of the MUT, and the file name of the stub should reflect the file name of the module being stubbed. Further, the purpose of each test should be clear from the name of the test itself. Test function names should be verbose, for example, the function name `test_sendMsg_NullMsgPtr` speaks for itself and anyone reading the test code will understand the purpose of the test. Non-descriptive test function names such as test_1, test_2, and so on should be avoided.

• **Test execution order should not matter:** Unit tests for functions of the MUT should be independent of each other. Further, it should be possible to change the order of test execution and get the same test results. To achieve this capability, each test should run set-up and tear-down functions to initialize and clean-up resources used by the test, respectively.

• **Tests should be self-verifying:** When the tests are self-verifying, the outcome of the unit test is immediately known, without the need to manually inspect and compare log files, or step through the code in the debugger. This allows regression tests to be run at any time, as part of the build delivery process or more frequently as determined necessary by the development team.

• **Structuring unit tests hierarchically facilitates comprehension of tests:** For better comprehension of unit tests, it is useful to structure them in a top-down fashion. For example, at the top level of the unit tests of the software bus module, the main function of the test runner calls a test function for each API of the MUT. On the next level, each unit test function calls the good and off-nominal behaviors for the corresponding FUT. We found that hierarchical structuring helps us to quickly comprehend what is tested (and not tested) by simply visualizing the call-graph structure of unit tests. Such structuring also helps to easily trace between unit tests and the actual behavior being tested.

• **Strategies for unit testing of internal functions:** In the C language, internal functions are either static or not declared in a public header file. If a function is not static then it has a global visibility even if internal usage only was intended. One strategy to test such internal functions is to test them indirectly (i.e., transitivity) through the publicly visible APIs of the MUT. If such internal functions are complex, then separate test functions have to be developed to test them directly. To facilitate unit testing, internal functions should not be declared static at development time. Some developers prefer bottom-up testing, meaning that they test the leave-level functions of the call graph first, whereas others prefer testing through the top-level APIs. In the CFS, there is a mix of both preferences.
• **Stubs should be simple and small**: Stubs should be as simple and small as possible in comparison to the real implementation. However, there are scenarios where the stubs need to resemble the production code, for example, when simulating a software bus with meaningful messages, as discussed in Section 3.6.1. If the stubs are complex, we have the challenge of debugging them.

• **Stubs should not depend on the stubs of other modules and the production code**: If stubs are independent of stubs of other modules and the production code, developers will be able to incrementally perform integration testing by plug-in and plug-out of dependent modules with stubs, recall Figure 3-6.

• **Assign a unique error code for each error type**: Reusing the same error code for different error types makes the unit tests complex because additional test code has to be developed to make sure the intended path was executed, recall Section 3.7.3.

• **The software architecture should support stubbing**: To facilitate reusing of stubs as well as controlling the number of stubs needed for unit testing, the architecture of the system under test should be designed in such a way that each module is used by other modules via a standardized set of APIs. Otherwise, “too many” stubs have to be developed for unit testing. For example, all application modules of the CFS follow a standard protocol for interactions with core modules, making it possible to define a common unit testing strategy for all modules.

• **The software architecture should abstract OS and Hardware**: Embedded systems typically require features of real-time OS as well as special embedded processors. To facilitate unit testing, modules should be abstracted away from APIs of special OS and hardware. Otherwise, developers cannot run tests on their own development machine using standard OS types such as Windows, Cygwin, and Linux, recall Figure 3-18.

• **The software architecture should separate computation, coordination, and communication**: For example, the application modules of the CFS do not know the fact that the software bus is implemented using OS queues. Due to the abstraction of the actual communication mechanism, it is possible to unit test each module by reading/writing messages to files, instead of requiring the software bus implementation to be ready and running.

• **Only the developers of the MUT can test certain (off-nominal) behaviors**: As discussed in Section 3.7.3, internal workings of the MUT are necessary to test certain behaviors and configurations. Not all behaviors can be tested strictly through the APIs of the MUT. For example, to test the behavior of the MUT for certain off-nominal behavior such as deleting a pipe by a non-owner requires sound understanding of how the relationship between a pipe and its owner is stored internally. Based on such internal details, the test program will be able to bring the module to the desired state.

• **Only the developers of the MUT can test certain configuration options**: To unit test a module that has several configuration parameters, the tester should know the internal details of how the configuration parameters are stored internally as well as the knowledge of dependency among configuration parameters. As discussed in Section 3.7.3, there is often no possibility to set values to configuration parameters through APIs.
• **Code coverage tools help deriving new tests:** For mission-critical systems, unit tests should at least cover all lines although code coverage does not necessarily mean that all cases have been tested. Code coverage tools help the tester by pinpointing those lines that are not covered by current tests. This capability enables the tester to investigate and construct additional tests. Note that if the test objective is to achieve the Modified Condition/Decision Coverage (MC/DC) criterion [125], a requirement of the aviation standards for safety-critical software, then line coverage is not enough because conditional statements such as an “if” or a “while” statement could be made of several decisions and tests should be constructed to make each decision independently pass or fail.

• **Call graphs and Metrics are useful to analyze quality of tests.** The analysis shows that by visualizing the call graphs of tests and using straight-forward metrics, such as the number of tests, size of the tests, the number of stubs, the size of stubs, and the size of the production code, we are able to understand the testing strategy as well as the modularity of tests and the production code.

• **Relational query languages are useful for analysis of tests.** During the analysis of unit tests, analysts need answers to questions such as which tests invoke a given function? And which functions are transitively tested during the execution of other functions? We found it useful to employ the RPA toolkit for automatically obtaining answers to these questions.

### 3.9 Related Work

In this section, we compare the tools and techniques in use to test CFS with existing work on stubbing frameworks, testing of SPLs, and SPL architecture-support for testing.

**Unit testing with stubbing frameworks for the C language:** Inspired by the success of the JUnit framework, an array of unit testing frameworks for C has been developed [285]. These frameworks offer built-in assertion capabilities and execute test programs and report test statistics such as the number of failures. Among these frameworks, we found a few that support stubbing for the C language such as CMock [58], CMockery [59], CppUTest [62], and LCUT [179]. These frameworks generate stub implementations of dependent modules based on header files of modules used by the MUT. Using these frameworks, unit tests can verify whether a dependent function used by the FUT was invoked with certain parameters or not. Also, unit tests can force the dependent function to return certain return code of interest. It is worth noting that these frameworks were developed and released in the period of 2005-2010. In contrast, the CFS’ unit testing framework (UTF) and the counter-value pattern have a long heritage, meaning that several NASA missions were tested using the stubbing technique introduced in this chapter. Nevertheless, we acknowledge these open source frameworks have more automation capability than the CFS testing infrastructure. As part of the on-going work, we might investigate the benefits and challenges of integrating existing open source frameworks with the CFS’ testing frameworks.

The dependency injection features of the Spring [267] and Google’s Juice [145] frameworks facilitate unit testing because binding to the actual implementation is
configurable. Dependency injection is popular in the Object-oriented community, but is rather uncommon for procedural programming (e.g., the C language). In the CFS case, dependency injection is simulated using the concepts of stubs.

Sellink and Verhoef noted in [256] that a very common concept in the development of systems is the use of code, data, or entire programs that are built for debugging or tracing purposes, but never intended to be in the final product. Stubs of the CFS are indeed used for testing and debugging purposes. This technology is also known as scaffolding. Knuth ([162], p. 189) mentions that the "most effective debugging techniques seem to be those which are designed and built into the program itself--many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other hand; the first half, which usually consists of fairly straightforward routines, will eventually be thrown away, but the net result is a surprising gain in productivity." Brooks ([40], p. 148) wrote that it is "not unreasonable for there to be half as much code in scaffolding as there is in the [final] product. As shown in this chapter, unit testing of modules of different layers of the CFS would not be practically possible without developing stubs.

**Testing of SPLs:** In [176] and [280], surveys of testing methods and processes for a SPL are presented. They acknowledge that there are only a few papers on testing of a SPL. In contrast to our work, techniques covered in these surveys typically do not cover white-box, unit testing of modules in isolation. McGregor’s [196] conceptual framework for testing a SPL discusses, for example, test plans, test assets, and test reports development. Combinatorial testing concepts based on orthogonal arrays are explained to test configuration parameters. It would be interesting to investigate the relationship between line or path coverage with respect to n-way coverage of orthogonal arrays. Reisner et al. [240] used symbolic evaluation to discover how the settings of configuration parameters affect line, basic block, edge, and condition coverage. In the CFS case, configuration parameters are tested by partitioning the possible values into different categories as needed to cover different paths and conditions. Also, internal design details are used to bring the system to a certain state for testing certain configurations. Pohl and Metzger [232] present a scenario-based test generation technique for testing a SPL. They derive test cases from use case diagrams. Uzuncaovo et al. [288] present a formal, specification-based test generation technique. They use constraint solvers to derive test data from formal models of features. These two techniques are for black-box testing in contrast to white-box, unit-testing of a SPL discussed in this chapter.

**SPL Architecture and Testing:** Trew presents an approach for how to facilitate testing by introducing architectural rules [282]. This observation is similar to the CFS in the sense that rules are added to the architecture. Linden and Müller present the building-block architecting method [183], which shares architectural similarities with the CFS architectural principles. The concepts of *generics* and *specifics* in their method can be mapped to the core layer and the application layer of the CFS, respectively. In their method, no two specifics are allowed to communicate directly, but all communication should only through generics. This is similar to the ideas behind the CFS where an application is not allowed to directly communicate with other applications. The idea of using abstract interfaces to manage variability is supported in the building-block method as well as in the CFS architecture. In an academic SPL study, Heineman [127]
explains that when abstract dependencies are present, one has greater flexibility in whether to use actual modules or stub modules. The results of the CFS analysis support that argument. The CFS analysis provides concrete examples for how to make unit testing easier by using abstract interfaces and alternative implementations. Michlmayr et al. [202] presents a preliminary framework (RAY) for unit testing of publish/subscribe applications against linear temporal logic (LTL) specifications. Similar to the CFS, the RAY framework also includes a mock component of the publish/subscribe infrastructure to facilitate unit testing of publish/subscribe applications. In contrast to the RAY framework, the CFS unit testing framework has more power, for example, it is possible to define a function hook of a publish function that can return success for the first invocation and can return failure for the second invocation. RAY does not support such sophisticated function hooks.

3.10 Closing Remarks

In this chapter, we described the analysis of the CFS product lines’ unit testing strategy and accompanying unit test cases and testing environment. The CFS has been refined over more than 10 years and has gone through rigorous inspections and improvement initiatives. In addition, the CFS captures knowledge from implementing dependable flight software for more than 20 years of specifying, developing, testing and flying with such software. Thus, we are grateful that we can analyze and use CFS as an example of good software engineering that we can all learn from, even though there are always some issues that can be optimized. One example of possible optimization would be to separate nominal and off-nominal behaviors that are currently mixed in several of the unit tests.

The CFS has successfully tackled the practical unit testing problem that modules often depend on other modules, making them hard to separate and unit test in an independent fashion. They have also addressed the issue that modules often depend on unique features and functions provided by the operating systems and may require the hardware in-the-loop for the software to function properly. Such dependencies impede setting up a controlled unit test environment. The CFS has addressed this problem by introducing stubs and abstraction layers for handling variability in SPLs.

The CFS’ approach to unit testing also allows to use the same set-up for incremental integration testing because stubs for testing can be swapped in or out depending on the situation, thus limiting the risks that are associated with big bang integration testing. However, unit testing and the type of incremental integration testing described above are only two aspects of testing, and other forms of testing need to be conducted in order to detect those types of defects that such testing cannot detect. Supported by the NASA IV&V center, Fraunhofer, and VU in collaboration with the CFS team, is researching ways to develop new testing techniques that address such issues.
Chapter 4
Architecture Compliance Checking at Runtime

4.1 Abstract

In this chapter, we report on our experiences with architecture compliance checking - the process of checking whether the planned or specified software architecture is obeyed by the running system - of an OSGi-based, dynamically evolving application in the office domain. To that end, we first show how to dynamically instrument a running system in the context of OSGi in order to collect runtime traces. Second, we explain how to bridge the abstraction gap between runtime traces and software architectures, through the construction of hierarchical colored Petri nets (CP-nets). In addition, we demonstrate how to design reusable hierarchical CP-nets. In an industry example, we were able to extract views that helped us to identify a number of architecturally relevant issues (e.g., architectural style violations, behavior violations) that would not have been detected otherwise, and could have caused serious problems like system malfunctioning or unauthorized access to sensitive data. Finally, we package valuable experiences and lessons learned from this endeavor.

Keywords: Runtime Monitoring, Architecture Compliance Checking, and Hierarchical Colored Petri nets.

4.2 Introduction

The Japanese office equipment manufacturer Ricoh Co. Ltd. develops Multi-Function Peripherals (MFPs) that have several functions, such as copy, scan, print, facsimile, etc. In the future, subsidiaries of Ricoh should be able to customize and extend the functionality by plugging in components, and thus enriching the feature set provided by the MFP. Moreover, these MFPs are not going to be stand-alone devices: they will be more and more integrated into highly dynamic enterprise systems. All these business requirements drive Ricoh to invest into software architectures.

Fraunhofer supports Ricoh in exploring architectural challenges for MFPs, which are long-lived products and have to undergo many releases for many customers with hard deadlines. Thus, in order to avoid architecture degeneration during evolution, Ricoh decided to introduce a systematic architecture compliance checking approach. Architecture compliance checking has been proven to be a usable and useful approach

---

4 Based on the version appeared in the International Journal on Information and Software Technology (IST), 2009 [95]. This invited article was an extended version of the best papers of the IEEE International Conference on Software Quality (QSIC), 2008.
for avoiding architecture degeneration (e.g., see [46], [210], [168], [233], [272], and [96]).

Manually inspecting a running system for its architectural compliance is nearly an impossible task, for many practical reasons. Given the size and complexity of typical industrial systems, it is almost impossible to inspect the control and dataflow of a running system for architectural compliance. Furthermore, manual inspection is not possible if the system changes in the field (e.g., on-the-fly component updates). Thus, a systematic, automated, built-in approach is necessary for monitoring the running system for architectural compliance.

Existing approaches address the problem by extracting the implemented architecture using static code analysis and then comparing it to the planned architecture. If relationships among code elements are known before runtime, these methods work well. In general, this scenario is not necessarily true for systems that are based on frameworks (e.g., OSGi [220]), or that adapt dynamically at runtime (e.g., on-the-fly component updates [258], [183], and [218]). The OSGi framework, which is used by the MFP, is a module system and service platform for the Java programming language that implements a complete and dynamic component model. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. In the office domain, runtime adaptation is becoming more and more important. For instance, software services might be downloaded and installed on customer sites or office devices are integrated at runtime with other devices in order to support intelligent workflow management.

Runtime analysis approaches published in the literature usually do architectural analysis in an offline mode, that is, after the system’s execution. For systems that never terminate, adapt or evolve dynamically (e.g., [128], [192]) however, we need to be able to monitor the running system in a non-intrusive manner, and perform architectural compliance checking at runtime itself. This capability would help in earlier detection of significant problems, such as illegal access to sensitive data, the system malfunctioning due to inappropriate usages of MFP components by the software extensions plugged-in by the subsidiaries of Ricoh, etc. Thus, we believe it is crucial to do appropriate system monitoring and architectural compliance checking in order to avoid such kind of problems. To that end, we evaluated a prototype version of the MFP using the DiscoTect method [251].

In this chapter, we present our experiences and adaptations of the DiscoTect method, which supports architectural view extraction at runtime. More specifically, we present an approach for collecting runtime traces from systems based on the OSGi framework. We then demonstrate our approach for monitoring the running system and discovering the actual software architecture using runtime traces for the goal of architectural compliance checking. We will show how to systematically design hierarchical Colored Petri nets for that goal.

The results offer evidence that the proposed approach is capable of discovering the actual architecture at runtime as well as violations of specified architectural rules or constraints in a systematic way. We demonstrate, using the pipe-and-filter architectural style [260], that the proposed approach can discover the actual configuration of the
style as well as potential violations of the style constraints at runtime. Examples of violations include a) unused filter, b) unused pipes, c) direct function calls between filters, and d) even the possibility of halting the whole system by misusing the interfaces of adjacent filters.

Structure of the remaining chapter: In Section 4.3, preliminary concepts and our adaptations of the DiscoTect method are presented. In Section 4.4, the proposed approach is explained. In Section 4.5, the approach is applied on Ricoh’s MFP prototype. In Section 4.6, experiences and lessons learned, effort data, limitations of the approach, and feedback of developers and architects are presented. In Section 4.7, the existing closely related research literature is discussed and compared. In Section 4.8, conclusions are drawn.

4.3 Motivation

The concepts of Colored Petri nets (CP-nets) are the main building blocks of this chapter. Readers who are not familiar with the syntax and semantics of CP-nets are referred to Appendix C for an overview. First, we present the motivation for choosing CP-nets for architecture discovery. Second, we present the motivation for adaptations of the DiscoTect method.

4.3.1 Why CP-nets for Architecture Discovery

We experimented with CP-nets and found them suitable for purpose of analyzing runtime events to discover the actual architecture of a running system. We also note that other mathematical frameworks, such as, process algebra [14], [84], Algebra of Communicating Processes (ACP) [14], Calculus of Communicating Systems (CCS) [203], micro Common Representation Language (μCRL) [208], [85] and its successor μCRL2 [197], Communicating Sequential Processes (CSP) [129] or Structural Operational Semantics (SOS) [3], might also be used for the same purpose. In principle it is possible to convert petri nets into SOS scheme's but that the ensuing graphical notation is not as intuitive as for CP-nets. As noted in [208], [85], and [197], basic process algebras such as CCS, CSP, and ACP are well suited for the study of elementary behavioral properties of systems. One main problem is that process algebras tend to lack the ability to handle data. μCRL has the capability to handle data. Data are defined by means of equational abstract data types. Data of this kind are integrated in the process algebra ACP, by allowing process variables and actions to be parameterized with data. Readers interested in discussions related to the comparison of Petri nets with other mathematical frameworks are referred to [16], [2], and [67].

We used CP-nets because a) CP-nets have the capability to handle not only events but also data, time, and hierarchy which are not necessarily well supported in process algebra based notations [2], b) CP-nets have intuitive visualization, c) CP-nets are supported by several industrial strength tools [227], d) CP-nets support notations for synchronization of sequential as well as interleaved events that can be used to track the progress of a running system and reconstruct architectural abstractions based on primitive runtime events (e.g., method call, instance creation, etc.), and e) CP-nets have
preconditions associated with transitions that can be used to analyze runtime events and choose events of interest that satisfy preconditions.

Let us consider a system based on the pipe-and-filter architectural style [260]. During execution of the system, its architectural elements, such as filters, read ports, write ports, and pipes may be created at different points over its lifetime. Basically, we have fragments of runtime events that contribute to the architectural style. If we want to discover the actual configuration of the pipe-and-filter style, we need a suitable way to keep track of fragmented events and synchronize and stitch them together. The places of CP-nets are used to hold runtime events as well as intermediate results. The preconditions of transitions are used to encode the pattern matching as follows. For example, when an instance of the class "Pipe" is created then produce a pipe connector as an output token. Basically, CP-nets are used to keep track of the progress of a running system and build the actual architecture over the period of execution.

4.3.2 Adaptations of the DiscoTect Method

Figure 4-1 shows the architecture of the DiscoTect method. The focus of the DiscoTect method is to extract a Component-Connector view (C-C) [55] from a running system. A pipe-and-filter architectural style is an example of a C-C view, where Filters are Components and Pipes are Connectors. The essence behind the DiscoTect method is to define a mapping between the planned architecture style (e.g., pipe-and-filter) and implementation style (i.e., coding conventions). For example, a mapping was stated as follows. Every instance of a class inherited from the “Filter” class is considered a “Filter” component. By formalizing such mappings using CP-nets, the sequence of runtime events (e.g., call events, object creation events) is interpreted in terms of architecturally significant events (e.g., creation of components, connectors). The mapping language, with the semantics of CP-nets, used in DiscoTect is called DiscoSTEP in Figure 4-1.

Figure 4-1: The DiscoTect Architecture (figure from [251]).

Adaptations. We attempted to apply the DiscoTect method to MFP. However, due to the following issues that appeared during our evaluation of the DiscoTect, we had to adapt various components of the DiscoTect method in the following ways.
First, we had to enable runtime data collection in order to handle unknown components that will enter the system at runtime. Since the DiscoTect method does not provide any facilities to do that, we developed a load-time weaving approach for runtime data collection. This adaptation corresponds to “Probes” in Figure 4-1. An approach for runtime data collection is presented in Section 4.4.3.

Second, we had to decouple the DiscoTect method from its mapping language called DiscoSTEP, used for specifying a mapping between the planned architectural style and implementation style. The DiscoSTEP specification is automatically converted into CP-nets in order to extract the runtime architecture. In our evaluation, we found bugs in this automatic conversion step. Due to the prototypical state of the tool at the time, we decided not to use the DiscoSTEP language. Thus, we use existing general purpose CP-net languages. That is, we define a mapping between the planned architectural style and its implementation using CP-nets which are modeled using a commercial-off-the-shelf component called Exspect [1]. Through this adaptation, we take advantage of easy-to-use and freely available tools that support CP-net construction, execution, testing, and debugging. To summarize, we used the Exspect functional programming language instead of the DiscoSTEP language for modeling the CP-nets.

Third, we had to extract sequence diagrams in order to check behavioral compliance at runtime. This capability is not addressed in the DiscoTect method. We adopted our approach such that we are not only able to check architecture-level structural compliance but also check for architecture-level behavior compliance such as the ordering of messages exchanged between components.

Fourth, we had to adapt the architecture visualization tool (called “Architecture Builder” in Figure 4-1) to our context. For integrity reasons - since we already had a well-established tool for architecture visualization - we used the Fraunhofer SAVE (Software Architecture Visualization and Evaluation) tool [96], instead of the ACMEStudio tool used by the DiscoTect method. Per se, the SAVE tool alone cannot automatically extract and visualize C-C views (e.g., Pipes-and-Filters). This tool only considers low-level code relations, such as function/method calls, variable access, and include/import as connectors. Hence, the runtime interactions among components through connectors (e.g., pipes, RPC) are not shown. However, the SAVE tool supports the import feature of external data that allow users to visualize any dependency model extracted from other tools and methods. We used that import capability to visualize the architecture extracted from our approach presented in this chapter.

4.4 Approach

The problem that our approach addresses is to check whether a running system is in compliance with the planned architecture. To solve this problem, we need to resolve a number of technical challenges. In the subsequent sections, we first present these challenges, and then we give a brief overview of the proposed approach. Finally, we go the technical details of the approach.
4.4.1 Challenges in Architectural Compliance Checking

1. Monitoring the system, even in the field, to collect runtime events (e.g., call events) in a non-intrusive way. This should also include the capability to handle such systems where components can enter at runtime.

2. Bridging the abstraction gap between low-level runtime events and high-level architectural elements.

3. Handling of interleaved runtime events. While the system is running, different events partially contribute to architectural elements, and thus, the traces are interleaved with respect to the architecture. We need to recognize fragments of traces.

4. Managing different interests. Not all runtime events are of relevance to all architectural views. For example, a thread start event is of interest to a process or execution view, but not to a module view, where implementation relations are of relevance.

5. Storing the extracted architectural views in an appropriate format for further analysis and visualization by external tools.

4.4.2 Overview of the approach

In order to tackle the above challenges, the proposed approach is decomposed into two major phases (see Figure 4-2). In the following, we explain how different steps address the challenges.

Phase 1

- Define Goal
- Define Mapping
- Define Probes
- Construct CP-nets

Phase 2

- Execute System
- Evaluate Extracted Views

Figure 4-2: Steps of the approach.

Phase 1 starts with the definition of goals. That is, at an architectural-level goals define the concerns that need to be monitored. For instance, a goal definition was stated as “check whether the running system follows the pipe-and-filter architectural style”. Since there is an abstraction gap between software architectures and the system implementation, we define a mapping between them to tackle challenge 2. This mapping is used as input to define probes (i.e., monitoring code) and to construct CP-nets. Probes are used to collect traces from the running system. The idea is to do selective instrumentation of the target system in order to avoid unnecessary runtime
traces. Our approach for defining probes is illustrated in detail in Section 4.4.3 in order to address the challenge 1.

As mentioned before, the DiscoTect method exploits CP-nets for bridging the abstraction gap between runtime events and architectures. CP-nets are constructed in order to extract architectural views from runtime traces. In other words, CP-nets encode meta-models of the planned architectural views, and produce instances of the meta-models as output. In Section 4.4.4, we present our meta-models to demonstrate how runtime traces, software architectures, and CP-nets are put to work together.

In Phase 2, the system under analysis is executed, in a monitoring mode, using different use-cases or scenarios. This will generate the runtime events as an input for the CP-nets that emit the actual architecture when the preplanned patterns are recognized. After that, the extracted architecture is analyzed either at runtime or offline.

### 4.4.3 Runtime Trace Collection with the OSGi Framework

For collecting runtime events (a.k.a. traces), instrumentation has been used traditionally. However, the instrumentation of a system usually has to be done before the system is compiled. Consequently, instrumentation done before running the system is not capable of reflecting dynamic system changes at runtime. Since MFPs are utilizing the OSGi framework, there were some other challenges we faced due to the fact that components may enter the system whose implementation is not known a priori.

In order to address this issue, we followed an aspect-oriented approach [154]. The motivation for applying aspect-oriented programming was mainly driven by the fact that the code for instrumentation and the implementation of unknown components is separated per definition. Conceptually, aspects are considered a perfect fit since aspects modularize crosscutting concerns. Modularity is achieved by separating crosscutting code from the base code using so-called pointcuts, which are used for specifying the places in the base code where the crosscutting code (advice) should be placed. In our case, the advice was the instrumentation code that we wanted to place at particular places in the base code. For this reason, we implemented an AspectJ weaver that realizes load-time weaving in the OSGi framework [153].

Load-time weaving in the context of OSGi, however, turned out to be an unexpected challenge. On the one hand, we had to overcome conceptual issues in the realm of selective instrumentation. On the other hand, we needed to realize our solution concepts with a given technology. The technological solutions turned out to be on a very detailed level of granularity since we needed to solve OSGi issues on the platform level. In the following we elaborate on the concrete solutions that arose in the endeavor of runtime instrumentation within the OSGi framework.

With OSGi applications or components (coming in the form of bundles) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot (see Figure 4-3). A service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. With respect to aspect-orientation, a couple of approaches supporting load-time or runtime weaving already exist. When it comes to integrating aspect-oriented approaches with the OSGi platform,
however, two issues arise. The first issue concerns the adaptation of aspect weaving to the restricted class visibility among bundles within OSGi. Since each OSGi bundle has its own class loader defined, there is no standard way for accessing classes that are located in different bundles. However, this capability is essential in order to put the instrumentation code together with the bundle code. If a bundle wants to access services of another bundle, the bundle has to request the service registry. However, the bundle to be accessed at runtime has to be determined before compile-time. In that case, the OSGi registry returns a reference to the bundle that provides the appropriate service. After that, the requestor can submit a request to the bundle that was looked up in the registry. The bundle internal classes, however, are not visible to other bundles per se (e.g., if the required bundle packages are not explicitly exported and imported respectively). This raises problems when aspects have to be woven into a new bundle entering the system at runtime.

Due to these restrictions imposed by the platform, the connection between incoming bundles and the respective aspect bundles can be established during load-time the earliest. That implies that we need to completely decouple aspect code from component code. Therefore, the second issue arising can be regarded as a consequence of the solution to the first issue. A mechanism needs to be provided for separating the weaving information from components that can be utilized during class loading.

To support a clear separation of advice and its referred pointcuts, we partitioned the aspect into two parts. The first part is the advice logic (containing the instrumentation logic) and the second part is the pointcut implementation (coming with each component individually). To that end, we created a bundle that is in charge of holding the advices (the so-called AspectProvider). All other bundles that are determined for weaving bring

Figure 4-3: OSGi framework - Bundle concept.
their specific pointcut implementation. At that point we stress the fact that the advice logic and the pointcut implementations are decoupled completely.

### 4.4.3.1 Class Loading

As stated earlier, the class visibility is restricted to the scope of the respective bundle, and therefore, the separation of advice and pointcut causes some problems. For that reason we developed a customized class loading concept for OSGi. Assuming the AspectProvider bundle running, we dynamically create a special class loader each time a new bundle is being started. This customized class loader ("Intermediate Class loader") is initialized with a reference to the class loader of the AspectProvider bundle, and at the same time, as an extension of the newly entered bundle's class loader (see Figure 4-4). Using that construct, we exploit the java class loading mechanism by overriding specific methods, so the Intermediate Class Loader always finds the required classes.

![Figure 4-4: Creating an “Intermediate class loader”](image)

The resulting class loading process is then working as follows.

1. Apply the standard way the framework deals with class loading - `loadClass()`.
2. In case the class could not be found, the request is forwarded to the Intermediate class loader (ICL) - `loadClass()`.
3. ICL uses - `findLoadedClass()`.
4. Finally, ICL delegates to the class loader of the Aspect Provider bundle - `loadClass()`.
4.4.3.2 The Weaving Process

To successfully cope with load-time aspect weaving two important challenges had to be solved:

1. Determination of the classes and the advices that need to be woven

2. Support of loading dynamically constructed classes

Extension mechanisms were used for elegant coping with both of these challenges within the OSGi environment. The OSGi framework supports a mechanism to attach one bundle (called fragment) to another (called host). A fragment attached to the system bundle is called extension bundle. Extensions are used to insert additional functionality into the framework that is required for all running bundles. The Equinox implementation of OSGi provides a convenient way to implement and attach extension bundles. At that point, our concept exploited the same mechanisms as they are used in the AOSGi project [9]: hookable adaptor. In particular, we used ClassLoadingHook as well as BundleWatcherHook for performing our weaving strategy. ClassLoadingHook provides functionalities to manipulate class loading via the method processClass (see step 2 in Figure 4-5). Hook objects are initialized by the adaptor during the launch of the framework.

Figure 4-5: Overview of the weaving process.
The extensions-based aspect weaving approach comprises two detailed phases: An initialization phase and a weaving phase. Here, we describe the nitty-gritty of runtime data collection for systems based on the OSGi framework. Basically, this phase is responsible to inject monitoring code for collecting runtime data during execution of the system under analysis. The collected runtime data will be passed to CP-nets for architecture discovery and analysis of rules.

4.4.3.2.1 Initialization Phase

The initialization phase was performed by an extension bundle called org.weaver.hookextension. It contains a class called Hook that implements ClassLoadingHook and BundleWatcher. We manipulated class loading by replacing the base class loader of all bundles by a class loader that is capable of resolving all classes within the AspectProvider bundle’s class space. To that end, Hook creates and sets an instance of ExtendedDefaultClassLoader (EDCL) as base class loader for all the bundles that are starting. EDCL enhances DefaultClassLoader and provides the class loading strategy described before. That is, EDCL possesses an instance of an ICL that is used to load classes that are placed in the AspectProvider bundle. So we were able to intercept class loading of all bundles in order to launch class weaving before a class is defined by the EDCL. The EDCL performs standard class loading (step 1 in Figure 4-5) if the RuntimeWeaver bundle is not started.

4.4.3.2.2 Weaving Phase

Considering the RuntimeWeaver bundle running, the actual weaving phase was initialized in case a particular application bundle that has an aop.xml file defined tries to load a class. In this example, we advise all calls to the constructor of the class CopyFilter. However, here we do not specify how the calls should be advised. The advice logic is defined in the abstract aspect (see Figure 4-6).

```
<aspects>
  <concrete-aspect name="FilterConstruction" extends="">
    <pointcut name="filterConstructors"
      expression="call(CopyFilter.new(…))"/>
  </concrete-aspect>
</aspects>
```

Figure 4-6: Aspect definition with a concrete pointcut.

Please note that initially the extends-attribute of the aspect is empty. That is, the bundle coming with this description is not aware of the aspects that potentially use its pointcut definitions. By replacing the empty string with the name of the aspect that is supposed to be woven during load-time, we were able to instantiate the abstract aspect as it is used by the aspect bundle. The bundle holds a reference to the ICL that is provided by Hook to EDCL invoking it to resolve classes in step 2 in Figure 4-5. The ICL accomplishes steps 3 and 4 in the presented class loading schema. The decision
whether to start weaving a particular class are made by looking at only those bundles that contain a "META-INF/aop.xml" representation of concrete aspects. This mechanism also enables a dynamic match between concrete and abstract pointcuts (see step 5 in Figure 4-5). In the example given by Figure 4-7, we refer to a Java method (generateEvent()) that prints log data of a certain format: [name of the class, created object address]. Note that the pointcut filterConstructors() does not have an implementation at that point.

```java
public abstract aspect Probe_Aspect {
    // Track constructors of Filter
    public pointcut filterConstructors();

    after() returning (Object result): filterConstructors() {
        generateEvent (thisJoinPoint, result);
    }
}
```

Figure 4-7: Aspect definition using an abstract pointcut.

The RuntimeWeaver is responsible for the actual weaving. The bundle has a factory that is used by Hook to instantiate AJWeaver classes for all bundles having aop.xml files defined. AJWeaver matches all concrete and abstract pointcuts using their names and extends the concrete pointcut definitions of the respective bundles at load-time. That is, the extends-attribute as shown in the aop.xml would be changed to extends="Probe_Aspect". After this step, the aspect advices are eventually “connected” to the concrete bundle pointcuts. Then, the AspectJ API was used to perform the weaving (see step 6 in Figure 4-5). The woven class was then passed back to EDCL for defining the class in the bundle’s scope.

In the context of architecture compliance checking we were able to successfully instrument incoming OSGi bundles by using this approach. The load-time weaving in the OSGi context enabled us to conduct selective instrumentation of bundles to keep the profiling as well as the instrumentation code within acceptable limits.

### 4.4.4 Modeling Runtime Traces and Software Architectures using CP-nets

This section elaborates on the role of CP-nets in the context of encoding architectural abstractions. In other words, we describe how CP-nets connect the world of runtime traces with the world of architectural abstractions. In the world of CP-nets, tokens are used to model control- and dataflow. Therefore, the inputs and outputs of CP-nets are always tokens that are stored in places.
Every place in a CP-net has a data type. Hence, every token has a data type (see “CP-Net” in Figure 4-8). We treat both the input (runtime traces) and output (architectural views) as tokens. Equivalently, we define data types for both of them. Therefore, data types are viewed as the “interfaces” that connect the three different worlds (see Figure 4-8). An architectural view is defined as a set of architectural elements that are arranged according to a topology. A topology defines the meta-model of a view.

Table 4-1 contains some sample data types and definitions of architectural elements and runtime events. In Table 4-1, “*” refers to architecture-level tokens, and “+” refers to low-level runtime tokens. Note that the inheritance relation is a static relation. We extracted this relation at load time of classes, using reflection mechanisms which offer methods for identifying the parent of a class. Here, the main message is that we modeled architectural elements and runtime events as tokens with certain data types.
4.4 Approach

Table 4-1: Type definitions for CP-nets

<table>
<thead>
<tr>
<th>DataType</th>
<th>Data Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>archId:String, parentId:String, implId:String</td>
</tr>
<tr>
<td>Relation</td>
<td>source:Entity, target:Entity</td>
</tr>
<tr>
<td>Message</td>
<td>source:Entity, target:Entity, message:String,</td>
</tr>
<tr>
<td>Init</td>
<td>constructor:String, instance_id:String, args:List,</td>
</tr>
<tr>
<td>Call</td>
<td>Method:String, caller_obj:String, callee_obj:String, args:List</td>
</tr>
<tr>
<td>Inherit</td>
<td>derived_class:String, base_class:String</td>
</tr>
</tbody>
</table>

Thus, the purpose of CP-nets is to emit “high-level” tokens that represent architectural elements by using “low-level” tokens that represent runtime events. Our CP-nets extracted the set of all architectural elements collected from the planned architectural views, and leave the task of visualizing the views to other external programs. Table 4-2 presents how the elements of CP-nets were interpreted in the context of architectural compliance checking.
Table 4-2: Interpretation of CP-net Elements

<table>
<thead>
<tr>
<th>CP-net Element</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place</td>
<td>A place either refers to a type of runtime event or it a type of an architectural element.</td>
<td>We model every runtime event type as a place in CP-nets. For example, a Call event type has a CP-net place in our CP-nets. We also store the extracted architectural elements in the places of the CP-nets.</td>
</tr>
<tr>
<td>Token</td>
<td>A token either refers to a single runtime event of a certain type or an instance of an architectural element of certain type.</td>
<td>For example, every call event is a token in the place of type Call.</td>
</tr>
<tr>
<td>Transition</td>
<td>The precondition that formalizes the mapping between the architectural elements and the corresponding coding conventions or implementation style.</td>
<td>The mapping specifies how to interpret low-level events in terms of architecture. This specification is formalized using preconditions of transitions.</td>
</tr>
</tbody>
</table>

The execution of CP-nets follows the path shown in Figure 4-9. When a runtime event occurs (step 1), a token is generated that is put into all appropriate places in the CP-net. Places are appropriate in case that they match the respective runtime event type. After that, the CP-net preconditions will be evaluated (step 2) and transitions will be enabled according to the semantics of the CP-net (step 3). Finally, the extracted architectural element will be placed in the places that match the architectural element type (step 5). That is, all components and connectors will be placed in the respective places of the CP-net.
4.5 Architecture Compliance Checking in Action

Figure 4-10 shows an overview how the different parts are supposed to work together in practice.

The running system produces traces, and the traces are fed into CP-nets that emit architectural elements when preplanned architectural patterns are recognized. The architecture compliance checker, which is also a collection of CP-nets, inspects the state of CP-nets for architectural violations. Eventually, the architecture builder reads architecturally relevant tokens from CP-nets and visualizes the extracted architecture. In a sense, the system is “compiled” together with the CP-nets that are responsible for architecture extraction and compliance checking at runtime. JMS (Java Messaging Service) was used as an event bus to pass the runtime events collected by probes to CP-nets. The reasons for using JMS were as follows. (1) JMS supports both synchronous and asynchronous messaging services. Hence, a running program can put traces to a queue and keep running. (2) The running program and the CP-nets can run anywhere. This set-up is interesting for embedded systems like MFP because we can also do reverse engineering activities on a computer and the actual system can run on the actual
device. The CP-net feeder program reads the runtime data from the JMS bus and passes them to the CP-nets in the Exspect tool.

To apply our approach on the MFP example, we followed the process depicted in Figure 4-2; now, we instantiate each of the steps.

4.5.1 Step 1 – Define Goal

The goal was to check whether the MFP implementation is complaint to the planned pipe-and-filter (P-F) architectural style. There are many filters and pipes in the implementation, and one of the goals was to extract actual dependencies among filters via pipes in order to check whether style properties or constraints are obeyed by the implementation. In addition, the MFP design rule specifies that every filter component should register to the connected filters for error handling reasons, before exchanging any data between them. Traditionally, the P-F style does not allow such dependencies among filters, but the MFP needed them. Thus, the goal was also to extract a message sequence view to show that filters indeed register with connected filters before passing any data.

4.5.2 Step 2 – Define Mapping

Since there are many ways the P-F style could be implemented, the architects of MFP defined a mapping. In the mapping, the implementation style (also called coding conventions) of architectural elements is specified. It states how filters, ports, and pipes are expected to be implemented in the system. In the MFP case, the following fragment of the implementation style is followed.

**Filter:** Every instance of a class that inherits from the root class “Filter” is a filter.

**Pipe:** Every instance of the class “Pipe” is a pipe.

**Write Port:** A write port should only be created on a filter the first time it writes some data to a pipe instance. In addition, whenever the filter writes to a different “pipe” instance, a corresponding write port should also be created on the filter.

**Read Port:** A read port should only be created on a filter the first time it reads some data to a pipe instance. The implementation of MFP follows the observer pattern to realize the P-F style. That is, pipes are observable and filters are observers. When a filter wants to read from a pipe, it first has to register to the pipe. The pipe then calls the registered filter whenever there is some data in the pipe.

**Filter Connection:** If a filter writes to a pipe object and another filter reads from the same pipe object, then we conclude that the filters are architecturally connected.

4.5.3 Step 3 - Define Probes

We developed a collection of aspects to collect low-level events and used our load-time weaver to instrument the OSGi bundles of MFP. It is important to note that we do not instrument the complete system. We used the mapping definition of the previous step to instrument only those portions of the code which are relevant for extracting
architectural views. This particular step might be iterative if data analyses raise new questions.

4.5.4 Step 4 - Construct CP-nets

We will now show how CP-nets are used to formalize the mapping defined in Step 2 and in order to extract architectural views. We explain how to design CP-nets in such way that they become reusable across a set of systems that follow the same implementation style for realizing the intended architectural views.

Defining Token Data Type

Before constructing CP-nets, we need to define data types for architectural elements and low-level events. According to the goal definition, we need to extract the component-connector view of the pipe-and-filter (P-F) architecture. The architectural elements and data types relevant to these views are listed in Table 4-3. Using the mapping specification, we identified the list of low-level events, such as Init and Call (see Table 4-1), that are relevant to extracting the architectural elements.

Table 4-3: Data types of P-F Elements

<table>
<thead>
<tr>
<th>Architectural Element</th>
<th>Data Type</th>
<th>Data Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Entity</td>
<td>( archId ) - name of the filter ( parentId ) – not used here ( implId ) - instance id of the filter class</td>
</tr>
<tr>
<td>Read Port</td>
<td>Entity</td>
<td>( archId ) – name of the port ( parentId ) – name of the port that reads from the port ( implId ) – instance id of the pipe used by the filter</td>
</tr>
<tr>
<td>Write Port</td>
<td>Entity</td>
<td>Similar to Read Port (i.e., Write instead of Read)</td>
</tr>
<tr>
<td>Port Connector</td>
<td>Relation</td>
<td>( source ) - Write Port ( target ) - Read Port</td>
</tr>
<tr>
<td>Message</td>
<td>Message</td>
<td>( source ) – Filter ( target ) – Filter ( message ) – method name</td>
</tr>
</tbody>
</table>

A Sample CP-net

Here, we introduce a simple CP-net that emits a pipe id whenever an instance of pipe is created. The intention is to give a feeling of CP-nets to those readers who are not familiar with this formal method.

In Figure 4-11, the places \( init\_event \) and \( pipe\_id \) are of the type \( Init \) and \( String \), respectively (see Table 4-1). The transition \( InitPipe \) waits for an init event and fires if appropriate preconditions are satisfied. Recall that the mapping specification states that every instance of a class “Pipe” is a pipe.
We can formalize this specification in the Exspect functional language as follows:

**Precondition**: \( \text{init\_event}@\text{constructor} = 'Pipe' \)

**Action**: \( \text{pipe\_id} \leftarrow \text{init\_event}@\text{instance\_id} \)

Every time the InitPipe transition fires, a single token (a dot within the circle) that satisfies the precondition from init_event is consumed and a new token is produced in pipe_id.

### 4.5.5 Constructing a hierarchy of CP-nets

A hierarchical CP-net is a CP-net that contains one or more CP-nets. The main motivations for using hierarchical CP-nets for our context are:

- Sub-CP-nets can be tested independently.
- Sub-CP-nets are reusable for extracting architectural elements that are also present in other systems that are implemented in the same style.
- Sub-CP-nets are understandable by humans, as they are typically small and conceptually isolated modules.

The core idea is that each sub-CP-net emits a part of the architecture to be extracted; composition of these sub-CP-nets into a single CP-net emits the complete architecture. A fragment of a hierarchical decomposition of CP-nets for the P-F style is shown in Figure 4-12.

---

\( x @ y \) refers to field \( y \) of data type \( x \). \( x = y \) means “if \( x \) is equal to \( y \)”. \( x \leftarrow y \) means “\( y \) is assigned to place \( x \)”. 
The root level CP-net is composed of sub-CP-nets, namely, “pipe-and-filter” and “Filter to Filter Call”. The former CP-net emits architecture models for the P-F style, while the latter emits any direct dependency from one filter to another filter using method calls. At the next level of decomposition, the “pipe-and-filter” sub-CP-net is composed of sub-CP-nets for filter creation, port creation, initialization of pipes, and attachment of filters.

In this way, every CP-net is decomposed further until it is a separate module on its own. To improve the reusability of CP-nets, we follow a layered design of CP-nets. We hide implementation-specific details at lower layers; thereby, we can reuse the top-level structure.

### 4.5.6 Execution of CP-nets

Here, we show a brief collection from our library of CP-nets based on the above decomposition. Each CP-net emits a certain type of architectural element of the P-F style; composition of these CP-nets into a single CP-net produces the whole P-F architecture style.

**Common Conventions of all CP-nets:** The black boxes are also CP-nets. The circles with “i” (input pin) and “o” (output pin) in all the CP-nets denote the input/output from/to other CP-nets.

The data types for the places are shown in Table 4-4.

**Table 4-4:** Data types for the Places of CP-nets

<table>
<thead>
<tr>
<th>Place name(s)</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter*, write_port*, read_port*, _filter</td>
<td>Entity</td>
</tr>
<tr>
<td>port_connection_holder</td>
<td>Relation</td>
</tr>
<tr>
<td>Message_holder</td>
<td>Message</td>
</tr>
<tr>
<td>init_event</td>
<td>Init</td>
</tr>
</tbody>
</table>

---

6 x* and *y refer to the places with a starting and ending name x and y, respectively.
The Exspect tool supports a simulation mode where the flow of tokens among CP-nets is easy to visualize. In fact, the user can open each CP-net in a separate window, and visualize follow of tokens at runtime in each window. Here, we summarize the execution or simulation of CP-nets for our CP-nets that recognize the pipe-and-filter architectural style.

Creating Filters: The CP-net that creates filters (see Figure 4-13) emits filter tokens in the output pin “filter_holder” by waiting for an instantiation of classes that inherit from the base class “Filter” (as per the mapping definition).

![Figure 4-13: The CP-net for filter creation (Rule_CreateFilter).](image)

Every token of the place “child_of_filter” denotes a class derived from the base class “Filter”. The “CreateFilter” transition will fire if there is a token in the place “init_event” whose constructor or class name is the same as a token in the place “child_of_filter”. Note that it also gives back the token to “child_of_filter” because many instances of the class derived from the base class “Filter” can be created during program execution. If the transition did not give back the token, the “CreateFilter” would not be able to create more than one filter.

The precondition and action for creating filters are:

**Precondition:**  \( \text{init\_event}@\text{constructor}=\text{child\_of\_filter} \)

**Action:**  
\[
\text{filter\_holder} \leftarrow \text{init\_event}@\text{constructor} + \text{init\_event}@\text{instance\_id}
\]
\[
\text{child\_of\_filter} \leftarrow \text{init\_event}@\text{constructor}
\]

The precondition checks whether the created instance of a class is an instance of a child class of the “Filter” base class. If the precondition is true, then a filter component is created with name as the name of the constructor with the unique object instance id.

The CP-net that emits all children of the base class (see Figure 4-14) is composed with the filter creation transition “CreateFilter” using the output pin.
We can formalize the CP-net for recognizing children of the base class “Filter”:

**Precondition:** \( \text{inherit\_event}@\text{base\_class} = 'Filter' \)

**Action:** \( \text{child\_of\_filter} \leftarrow \text{inherit\_event}@\text{derived\_class} \)

**Creating Ports:** The CP-net for creating read ports (see Figure 4-15) waits for filter tokens (“filter\_holder” pin) and pipe tokens (“pipe\_id” pin) from other CP-nets. If the filters read data from the pipes, using an appropriate method call (placed in “call\_event”), then read ports are emitted in the place “read\_port\_holder”. Note that this CP-net gives back the filter tokens (“filter\_holder” pin acts both as an input and output pin). Otherwise, the filter token would be lost, and we could not create more than one read port in a filter. Similarly, the CP-net for write port creates write ports on those filters that write to pipes.

**Attaching Filters:** The CP-net for attaching filters (see Figure 4-16) waits for read and write port tokens, and emits tokens that contain a pair of attached write and read ports in the place “port\_connection\_holder”. A write port is attached to a read port if a filter writes data to a pipe object and another filter reads from the same pipe object. The “message\_holder” place contains the abstract logical message “process\_Data” when two filters are attached. The “message\_holder” is used for drawing sequence diagrams to show the ordering of data flow from one filter to another.

---

Figure 4-14: The CP-net for Rule_GetFilterChildren.

Figure 4-15: The CP-net for Rule_CreateReadPort.

Figure 4-16: The CP-net for Rule_AttachFilter.
Putting the Building Blocks Together: Once the building blocks are constructed they have to be composed in order to emit architectural elements. To compose the building blocks, the Exspect tool provides a GUI where the user has to connect the input/output pins of sub-CP-nets to appropriate places in the above CP-nets in the decomposition hierarchy.

The Composition of Sub-CP-nets follows a convention: If a token is needed as an input to more than one sub-CP-net, then token multiplexing is performed.

For instance, both of the CP-nets that create read and write ports require pipe tokens as inputs (see Figure 4-17). To pass the pipe tokens to those CP-nets, we create a transition “Multiplex_pipe_id” that does nothing but just copying the pipe token into two places, namely, “pipe_id_1” and “pipe_id_2”. All transitions with a name starting with “Multiplex” are token multiplexers.

Creating a Message Sequence View from runtime traces involves extracting components and messages exchanged among them. For the MFP case, every filter is treated as a component. Thus, we do not create separate CP-nets for the extraction of components; instead, we reuse the output filters of the “CreateFilter” CP-net, shown in Figure 4-13. In case a filter makes a direct method call to another filter, the invoked method name is used as the message name. This scenario is handled by the CP-net “Rule_Filter_2_Filter_Call” (see Figure 4-18). This CP-net monitors all calls and selects those calls whose source and target objects have already been identified as filters. Instead of a direct method call, if two filters are attached through a pipe, we emit an abstract logical message “processData” from the source filter to the target filter. In
Figure 4-18, the CP-net “AttachFilters” emits such a token in the place “message_holder”.

![Figure 4-18](image1)

**Figure 4-18: The CP-net for Rule_Filter_2_Filter_Call.**

**Storing the Extracted Views:** We stored the extracted views as a collection of architectural elements in the form of tokens. To this end, the architectural elements of the views, namely the component-connector view and the message sequence views, were stored in the root level CP-net (see Figure 4-19).

![Figure 4-19](image2)

**Figure 4-19: The root level CP-net.**

This CP-net is the result of composing the two sub-CP-nets discussed before. The architectural elements, such as filters, read ports, write ports, pipes, and messages are stored in the places “filter_holder_copy”, “read_port_holder”, ”write_port_holder”, ”port_connection_holder”, and “message_holder”, respectively.

To summarize, the whole process of developing CP-nets followed fundamental principles of software engineering [222], such as abstraction, modularity, encapsulation, and composition. The hierarchical decomposition of CP-nets into building blocks resulted in a modular design of CP-nets, where each module encapsulates its design decision. For instance, the CP-net for creating a read port (see Figure 4-15) on a filter does not know how the filter was created; it just has filter tokens as an input to the transition.

**Step 5 - Execute System (MFP)**

As mentioned earlier, huge portions of the functionality of MFPs are based on the OSGi platform. The MFP filters are implemented as OSGi bundles. We start the so-called OSGi bundles and trigger many scenarios (e.g., scan, copy, resize papers, etc). The emitted traces eventually reach our CP-nets, which extract architectural elements and store them as tokens.
Step 6 – Evaluate Extracted Views

In this step, the goal was visualize and evaluate the architecture that is stored as tokens in CP-nets places.

From the CP-nets places, an external program reads the tokens and prepares a visualization of architectural views. In our case, we automatically converted the architecture stored as tokens into the data format followed in our SAVE tool. Figure 4-20 is visualized using our SAVE tool, where boxes are filters and arrows are pipes. The SAVE tool is not yet capable of visualizing ports. Thus, the read and write ports of filters are not shown in this figure.

![Diagram of a sample extracted C-C view](image)

Figure 4-20: A sample extracted C-C view.

Apart from visualizing the actual architecture at runtime, we also evaluated whether the extracted architecture complies with the intended architectural properties. That is, we checked whether the architectural style constraints are not violated. In the MFP example, we check the P-F style properties, such as: a) Every filter should have a read or write port, b) Every write port should be connected to a read port, c) A filter either reads or writes to a single pipe, but not both, d) No two filters should make a direct method call, and e) Every filter is connected to at least one other filter through a pipe.

Since the extracted architecture is stored as tokens in the CP-nets, we developed a library of reusable CP-nets to automatically verify such style-specific properties. Note that the CP-nets that check for style constraints violations are reusable due to their independence from implementation details. Thus, the same CP-nets could be used to check architecture compliance across systems with the same planned architectural style, independent of the way the style is implemented.

In the following, we summarize the major results of compliance checking.

**Unused filters** were detected. Our analyses revealed that instances of filters had been created that were never used during the system execution.

**Unused pipes** were detected. In our CP-nets, the place that holds pipe tokens should ideally be empty after the MFP execution, but this was not always the case. Some pipes were created, and there was no reading or writing to them. Discussions with the architect of the MFP exposed the reasons: Every filter is responsible for creating its input pipes. Thus, input filters have input pipes, but no other filter writes to them. Hence, their input pipes were never used.
4.6 Discussion

**Direct method calls from filters to filters were detected.** However, this turned out not to be an architecture violation in the MFP case: At runtime, for error handling reasons, filters are allowed to make certain direct calls to other filters. Even though filters can directly communicate in MFP, there are some protocols they must follow. For example, a filter is not allowed to send the ‘abort’ signal to other filters; users have to explicitly activate it from the GUI. It was expected that the design itself would enforce that rule, but we identified a violation: A filter could send an ‘abort’ signal to other filters. Such violations were detected at runtime, more interestingly even before the system crashes.

**Application-specific P-F style constraints were validated:** The planned architecture of the MFP defines three types of filters: input, process, and output. There are some protocols with respect to the data flow from one type of filter to another. For example, a process filter is not allowed to pass data to input filters. These constraints were validated with the help of the CP-nets.

![Figure 4-21: A sample extracted message sequence view.](image)

The extracted message sequence views capture behavioral aspects of the software over time. In MFPs, every filter must directly register itself to its connected filters for error handling reasons before passing any data (see Figure 4-21). The underlying idea was to enable an efficient backward propagation of failure events that should cause the stalling of the current data pipeline in order to avoid situations that are hard to recover from. Using the extracted message sequence view, we confirmed that the running system does follow that design rule.

### 4.6 Discussion

This subsection presents a) experiences and lessons learned, b) effort data, c) benefits and limitations, and d) feedback from developers or architects of MFP.

#### 4.6.1 Experiences and Lessons Learned

In this section, we present the experiences that we made throughout the application of our approach for architecture compliance checking at runtime. We align our experiences along the different phases that the approach is comprised of.
Phase 1:

Step 1: Define Goal

Goal definition is the most important step in the compliance checking process. It is worthwhile to spend time on clarifying the goals of runtime monitoring, that is, what kind of architectural views are to be extracted for validation. The goals help a) to define probes in a disciplined way (selective instrumentation) and, b) to construct appropriate CP-nets for the views of interest.

Step 2: Define Mapping

Human expert availability is important for constructing CP-nets and Probes. The approach is best applied in those contexts where there is an architect who can define a mapping between architectural elements and traces. If no such architect is available, we think it is probably very hard to do compliance checking.

Step 3: Define Probes

It is important to separate the monitoring code and source code. As we know, the source code of non-trivial systems is already complex. Adding the monitoring code for quality assurance purposes within the source code further increases the complexity. Moreover, it will be very hard to a) switch-on and switch-off the monitoring code and b) to reuse the monitoring code in similar contexts. Thus, we recommend clearly separating the monitoring concern from the source code, using technologies such as AspectJ.

Runtime trace collection requires significant effort. Given that our system is implemented in Java, we thought we could quickly write some aspects to obtain traces. However, this was not as easy as it sounds, as all our components are OSGi bundles that have their own class loaders, which makes aspect weaving a challenging task. Moreover, we needed methods in order to collect trace data while the system is running in a dynamic context: bundles come and go. Fortunately, we could work out load-time weaving concepts to overcome this problem. Although these are low-level technical issues related to data collection, we have to keep in mind that data collection is an important and critical task for runtime monitoring and architecture compliance checking.

AspectJ is non-intrusive. Although we have not quantified the overhead due to monitoring, we did not observe any notable difference in the response time before and after the instrumentation. Thus, we are of the view that AspectJ is non-intrusive. However, it is also worth mentioning that we have done selected instrumentation using the goal definition, as explained earlier.

Load-time weaving approach is necessary and useful for automatic instrumentation of OSGi applications. In this case study, the user of the system is not at all aware that the running system is monitored. This was achieved using our load-time weaving approach, which seamlessly injects monitoring code whenever a new OSGi bundle enters the system. The emitted runtime data is then used to extract the actual architecture, which can be used by engineers as a starting point for their detailed diagnosis and debugging of errors, if any. Using our approach, we were able to run the
system, and simultaneously visualize its actual architecture in more or less real-time. The architectural views get updated if architecturally significant events happen at runtime.

**Step 4: Construct CP-nets**

*Constructing CP-nets is an iterative process.* Although it might appear that our CP-nets are constructed in one step, this is not true. Given that CP-nets are used to formalize the implementation style based on the knowledge shared by architects, it is hard to gather that knowledge very precisely in the first step. In our case, we had three iterations to precisely model the implementation style. Nevertheless, the benefits of constructing CP-nets were worth the investment.

*Layered design improves the reusability of CP-nets.* To improve the reusability of CP-nets, we recommend a layered design of CP-nets. That is, hide implementation-specific details at the lower layers of CP-nets. In this way, the top-level structure can be reused in other systems. Also, by following the modularity principle in the design of CP-nets, we can reuse parts of the CP-nets to extract architectural elements that are implemented in the same style in other systems. On the other hand, the CP-nets that check the completeness (e.g., architectural style violations) of the extracted views can be reused as is, because they are free from any implementation style.

*Hierarchically decomposition of CP-nets controls design complexity.* Like source code, CP-nets can also quickly become complex as more and more places and transitions are introduced. We believe by carefully decomposing the design of CP-nets into hierarchies we would be able to better understand and maintain them.

*CP-nets should be tested.* Some problems such as deadlocks and live-locks occurred in our initial CP-nets. After some manual verification effort, we were able to fix them. If there are design errors in CP-nets, then the extracted software architecture may not be the actual architecture of the running system. Thus, it is crucial to spend effort on testing and verification of CP-nets. We found that the actual effort may depend on the modularity of CP-nets. In the future, we intend to apply automated model checking [133] of CP-nets using tools that looks for potential design errors in the CP-nets. One possibility is to automatically convert Petri nets into PROMELA models and apply SPIN’s verification capability for analyzing design errors such as deadlocks and live-locks, see [266] for further details.

**Phase 2:**

*Step 1: Execute the system*

*The running system should cover a lot of code:* The extracted architectural facts are as good as the code coverage. This is a general problem with dynamic or runtime analysis. In the MFP case, we executed many scenarios specified in the use cases. In addition, we used code coverage tools to measure coverage at class levels in order to ensure that we covered all the classes related to the P-F style.
4.6.2 General Experiences

Uniform coding convention helps architectural compliance checking. As explained earlier, the proposed approach relies on coding conventions to develop probes and CP-nets. If architectural elements are implemented with different coding conventions within the system, then both the probes (i.e., aspects) and also the preconditions of the CP-nets should be made aware of all coding conventions. Otherwise, the extracted architecture will not be a complete and correct representation of the running system. Multiple coding conventions increase the design complexity of probes and CP-nets, making architectural compliance checking more difficult.

Scalability of the approach: Although our study was done on a prototype of an MFP, we are optimistic about the scalability of the approach because of the modularity of CP-nets. Our CP-nets encode meta-models of architectural views to be extracted. For example, our CP-nets for the P-F style can be used on systems with any number of filters and pipes, as long as the same implementation style of the P-F is followed. Of course, with the assumption that a good tool like Exspect is used to model and run the CP-nets.

Applicability to other architectural styles: Although we demonstrated the approach on the pipe-filter architectural style, the approach itself is not limited to this style. In another experiment, we applied the approach on another system based on the JAVA’s Remote Method Invocation (RMI) architectural style. While it was not possible to reuse any of the CP-nets developed for the pipe-filter style to the architecture and implementation based on RMI, experiences with our first endeavor helped us to come with a good design of hierarchical CP-nets for RMI. In this case, it was more the reuse of processes than the reuse of product artifacts such as CP-nets or probes. However, the CP-nets designed for RMI would be reusable for all systems based on RMI.

Effort Data: The approximate effort we spent on various steps of architectural compliance checking is given in Table 4-5. To relate the effort to the system under analysis, we list some size metrics in Table 4-6.

Table 4-5: Effort Data for architecture compliance checking

<table>
<thead>
<tr>
<th>Activity</th>
<th>~ Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detailed study of the DiscoTect method and its prototype</td>
<td>60 hrs</td>
</tr>
<tr>
<td>Learning about CP-nets and the <em>Exspect</em> tool</td>
<td>40 hrs</td>
</tr>
<tr>
<td>Interview with a developer/architect</td>
<td>10 hrs</td>
</tr>
<tr>
<td>Development of aspects for trace collection</td>
<td>50 hrs</td>
</tr>
<tr>
<td>Construction of CP-nets for MFP</td>
<td>90 hrs</td>
</tr>
</tbody>
</table>

In order to apply an existing method developed by other researchers, we first needed to learn all details of the method. Some unplanned effort was also spent on debugging the DiscoTect prototype. As a consequence, we decided to adapt the method to our own tool chain. Since we had no practical experience with CP-nets, we also spent effort on learning about CP-nets and related tools. In order to construct CP-nets, the
implementation conventions had to be collected. For that purpose, we interviewed a developer/architect, who explained the coding conventions/style corresponding to the architectural elements.

Table 4-6: Size of the MFP Prototype under Study

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Statements</td>
<td>22651</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>603</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>3375</td>
</tr>
</tbody>
</table>

Note that the time spent on aspects development for the trace collection does not include the development time of our general-purpose framework for load-time weaving into the OSGi platform [220]. It is not a trivial effort to break down CP-nets into a library of reusable CP-nets. Thus, our major effort was spent on constructing CP-nets. Since the planned software architecture does not necessarily change frequently, this effort is spent only once per system. We anticipate that this effort will go down if we construct CP-nets for other architectural styles, because of the potential to reuse knowledge and experience from this endeavor.

**Benefits:** By using the OSGi bundle technology, Ricoh subsidiaries provide new components on demand to extend the MFP software functionality of their customers. Anticipating the way Ricoh’s subsidiaries customize and use Ricoh components is difficult. It might happen that they violate the usage protocols of Ricoh’s components, which would pose the risk of system crashes or malfunctioning. To avoid such problems, the system needs to be monitored at runtime for architecture compliance so that constructive steps (e.g., notify service engineers) can be taken if a violation occurs. Our experience with the approach indicates that CP-nets can also be deployed within MFPs in order to detect such kinds of problems even in the field.

**Limitations:** Besides the listed benefits, we also identified a few limitations. Although a running system might comply with its architecture, it might not obey timing properties. For example, there are requirements like p pages should be copied in m minutes. There should be some way (e.g., using timed CP-nets) to monitor and notify timing violations of components or connectors. One limitation is related to probes: the probes and CP-nets are constructed manually, requiring significant time and effort. While it might not be possible to completely eliminate this effort, in the future, we would investigate some ways to automatically generate probes and CP-nets from the planned architectural style and the mapping definition. This capability would make the method more appealing in practice.

**Developer/Architect Feedback:** When the approach was presented to Ricoh’s developers and architects, there were many questions at the beginning, in particular referring to CP-nets. Also, many people were not that familiar with aspect-oriented programming. Once the fundamentals of CP-nets and aspects were clarified, they showed quite a lot of interest in the approach. During the execution of CP-nets in the Exspect tool, the visualization of the flow of tokens among places attracted their attention. Moreover, they liked the fact that the approach does not change the source code at all. They would prefer having more automation support during the design of
CP-nets and probes. From the development team’s point of view, one problem in practice is that maintaining the overall architecture is hard, as systems are getting bigger and bigger with a lot of requirements and variants offered. The approach presented in this chapter takes a step forward towards resolving this particular practical issue. In the future, we plan to explore techniques that can construct CP-nets from the planned behavioral specifications formalized using sequence diagrams [221], which are relatively more familiar to developers.

4.7 Related Work

The DiscoTect method was the basis of our work. The adaptations made were already discussed in the previous sections. This section places the DiscoTect method and our adaptations in the context of other research work.

In the conference paper [94], we presented the experiences of extracting architectures from runtime traces. In this chapter, we addressed the following, which were not covered in there:

1. We explained our approach for runtime trace collection on the OSGi framework.
2. We elaborated the experiences and lessons learned.
3. We introduced the related work section to position this work in the context of existing approaches.

Our work is related to a) techniques that apply CP-nets for architecture modeling and evaluation, b) aspect-oriented runtime monitoring, and c) architecture extraction using runtime traces.

4.7.1 Architecture Modeling and Evaluation using CP-nets

CP-nets have already been applied to modeling and evaluation of software architectures. The following three references are related to evaluating software architectures before the implementation.

Gomaa and Pettit [118] validate the dynamic behavior of concurrent systems that are modeled in UML by automatically converting UML models into CP-nets. Using the simulation capability of Design/CPN toolset, the authors check for concurrency issues, such as deadlock, livelock, and timing constraints.

Xu and Kuusela [312] model an execution architecture based on Petri nets. A module view is converted into a Petri net, and, using a simulation of the net, the authors identify quality problems, such as performance and throughput.

Fukuzawa and Saeki [87] apply CP-nets to evaluate quality properties of software architectures. Quality models for reliability, security, and performance using probabilistic CP-nets with timed transitions are developed. Using simulation of CP-nets, the authors perform trade-off analysis among architectural alternatives.

The simulation results are as good as the actual implementation of the planned architecture. Our case study complements their approach by reverse engineering the
implemented architectural views, which could then be used as an input to build and refine the simulation models for forward engineering purposes.

4.7.2 Aspect-Oriented Monitoring of Running Systems

In fact, monitoring the behavior of a running system is a concern similar to logging. One of the main issues in monitoring is overhead in terms of memory and response time due to additional monitoring code. Aspect-oriented programming has shown to be a promising concept for separating concerns [154] in an efficient way. Furthermore, aspects are developed as separate modules and weaved into the functional code, thereby clearly separating the system monitoring code and real code, and can also be switched off in case monitoring is not necessary.

Richters and Gogolla proposed a method [242] for checking whether behavioral constraints, specified in the OCL language of UML models, are followed by the implementation. Using the AspectJ language, the authors validate the running system against the expected behavior. Their work focuses on method-level preconditions, post-conditions, and invariants checking, in contrast to our focus on software architecture-level conformance checking.

Kiviluoma et al. [156] proposed a runtime monitoring method to validate a running system against its expected behavior, which is specified using UML profiles. From the profiles, the authors automatically derive and weave aspects into code to monitor expected behaviors. In our case, aspects are not automatically generated. The patterns of pointcuts and advice are programmed manually to collect runtime traces. Our primary focus was on preparing incoming components for being instrumented before load-time, which was not addressed in [156].

Briand et al. [39] proposed a comprehensive methodology for reverse engineering of sequence diagrams from distributed multi-threaded programs. The unique aspect of their work lies in the sound definitions of meta-models for runtime traces and sequence diagrams, which are often missing or unclear in other reports. Their meta-models include control statements and loops, usually ignored by other tools. So far we have not focused on detailed code-level analysis, since we work at an architectural-level currently. Hence, we do not yet collect information regarding control statements and loops.

4.7.3 Architecture Extraction using Runtime Traces

Riva and Rodriguez [243] combine static and dynamic analysis for architecture reconstruction. If the system structure is fixed and there are no dynamic updates of software components, then their approach works well. They do not distinguish connectors from components. Consequently, dependencies among components communicating only through connectors (e.g., pipe) cannot be recovered.

The SAVE tool [96], developed at Fraunhofer, extracts both the structural and behavioral views from source code and runtime traces, respectively. To build abstraction from low-level raw data, it makes use of the directory structure decomposition. Thus, the resulting views contain components (basically directories or files) and their interactions (e.g., function/method call, variable access). In [186], a
system of systems is analyzed by extracting the actual sequence diagram by snooping
the data passed in the network. That approach is not applicable for systems whose
components communicate only through connectors like pipes.

The pattern-lint tool [241] uses a hybrid approach (i.e., both static and dynamic
analysis) to detect deviations between the planned architecture and the actual
architecture implemented in the code. Pattern-lint uses prolog rules to express the
planned architecture. By using static analysis, source code relations are extracted and
stored as prolog facts. After that, the extracted facts are verified using the prolog rules.
In addition to static analysis, the actual system is executed and a number of views
based on the collected runtime data are visualized to mainly detect architectural
violations.

The Rigi tool [273] is an environment for visualizing the static structure of a system. In
general, the Rigi tool can be used to visualize any hierarchical directed graph. The user
can build abstract views from low-level dependencies either through automated
clustering or manually through Rigi’s graph editing features, such as collapsing, hiding,
etc. The tool has a simple import format allowing the user to import code-level
relations, such as call, include, use, etc. Although Rigi has been mainly used for
visualizing the static structure of a system, it can still be used to visualize dynamic
structure as long as the user obeys its import format. In fact, the Dali tool [148] from
the SEI uses Rigi to visualize the structure of the system after merging the static and
dynamic relations into a single database of relations. However, Rigi itself cannot
discover architectural styles from a sequence of runtime events. In addition, there is no
notion of software connectors, such as Pipes and Sockets, in the Rigi tool.

Arias et al. [11] identify dependencies among execution entities using dynamic
analysis. In addition to program traces emitted by the running system, they also monitor
operating system activities such inter-process communication [6] and execution of
tasks. Their approach extracts dependencies among tasks. In our approach, we have not
yet focused on the OS-level behavior analysis, which could give a complementary view
on the running system.

In all the above approaches, structural and behavioral views were extracted after
execution of the system using the complete traces, unlike our approach which does
architecture compliance checking at runtime using on-line traces.

4.8 Closing Remarks

We presented a method for verifying a running system against its planned architecture.
This method is especially applicable for systems whose components can enter and
leave at runtime, and communicate using connectors like pipes. For such classes of
systems, statically analyzing the implemented architecture is not sufficient. Using
dynamic analysis the actual architecture was extracted at runtime, and was evaluated
for violations of the intended style constraints. We have proposed a new method for
systematically instrumenting a running system using ideas from aspect-oriented
programming. Furthermore, we have shown how to bridge the abstraction gap between
the collected runtime traces and the planned software architecture using hierarchical
Colored Petri nets (CP-nets).
We successfully applied architecture compliance checking to Ricoh’s MFP prototype. To that end, we customized the DiscoTect method to our purposes: a) by directly modeling the mapping between architectural elements and runtime traces using hierarchical CP-nets, which are defined using general-purpose functional programming languages, b) by developing a tool chain for practically applying the method to overcome the technical barriers of the DiscoTect prototype. We have shown in detail how to systematically design reusable CP-nets.

Our experiences with this approach indicate that there is definitely an up-front investment to construct probes, design, develop, and test CP-nets. But, in our opinion it is worth applying this approach because we can show that both the architecture and the running system is consistent, otherwise software architecture is just a hypothesis. For Ricoh’s MFP prototype, architectural views and a number of issues, such as unused filters, pipes, and design rule violations in component usages, have been extracted. Moreover, the major benefit of the approach is that it has been used to check compliance between the running system and the planned architecture at runtime. Ricoh considers this approach to be of strong practical relevance for quality assurance. Defining an appropriate architecture and then checking its compliance at runtime are emerging as a relevant quality assurance strategy.
Chapter 5
An Analysis of the Publish-Subscribe Style

5.1 Abstract

Architectural styles impose constraints on both the topology and the interaction behavior of involved parties. In this chapter, we propose an approach for analyzing implemented systems based on the publisher-subscriber architectural style. From the style definition, we derive a set of reusable questions and show that some of them are answered statically whereas others are best answered using dynamic analysis. This chapter explains how the results of static analysis were used to orchestrate dynamic analysis. The proposed method was successfully applied on the NASA’s Goddard Mission Services Evolution Center (GMSEC) software product line. The results show that the GMSEC has a novel reusable vendor-independent middleware abstraction layer that allows the NASA’s missions to configure the middleware of interest without changing the publishers’ or subscribers’ source code, and b) a high-priority bug due to behavioral discrepancies, which were eluded during testing and code reviews, among different implementations of the same APIs for different vendors.

Keywords: Architectural Styles, Middleware, Vendors, Static and Dynamic Analysis, Component-Connector Views, and Colored Petri Nets.

5.2 Introduction

Architectural styles are abstract “high-level” concepts offering reusable solutions to recurring design problems. Equivalently, architectural styles define the roles, the topology, and the interaction behavior of involved components [259]. The publisher-subscriber architectural style is one of the most prominent styles, in which different components communicate in an indirect fashion by publishing and subscribing to messages managed by an intermediate communication bus (or broker) [72]. This indirect communication makes the architecture flexible because it facilitates adding and removing components [21]. This style is thus an attractive option, for example, when one wants to develop a family of similar products in a disciplined way. For example, in a previous case study [97], we reported in Chapter 2 that the NASA’s flight software product line was developed in a flexible way by adding or removing publisher/subscribers, based on the needs of missions.

---

However, this increased flexibility of the publisher-subscriber architectural style is also a curse because it makes it difficult to predict the emerging behavior and to prove the correctness of the integrated system even though each individual component passed testing and was demonstrated to be correct on its own. Thus, while flexibility increased, analyzability decreased. One example of problems that cannot be detected by analyzing individual components is inter-component oriented timing issues. Such issues emerge only when several components are using the bus.

Several researchers have analyzed the publisher-subscriber architectural style at an early stage (i.e., before the implementation) (e.g., [111], and [113]). They construct rigorous formal models of the publisher-subscriber style and prove correctness using model-checking techniques. Unfortunately, many existing systems were developed without such a rigorous architectural phase. Also, even if there was such a phase, experience reminds us that the implementation could deviate from the specified architecture (e.g., [233], [95], and [272]). Therefore, it is instructive to reverse engineer the implemented architecture and analyze the behavioral properties and constraints of the publisher-subscriber style.

We developed a practical approach for analyzing implementations based on the publisher-subscriber style. Our key activity is to derive a set of reusable questions from the definition of the style. These questions drive the analysis and the answers to them constitute evidence regarding the compliance of the implementation to the style constraints and overall design quality. We will show that some questions are possible to answer statically, whereas others are better answered by monitoring the running system. In the static analysis phase, we semi-automatically bridge the gap between “high-level” concepts such as publish, subscribe, unsubscribe, and communication bus, and the source code concepts. That is, we locate key interfaces, methods, and data structures used for implementing the publisher-subscriber style. Our static analysis is guided by analyzing dependencies to external entities (e.g., Middleware vendors’ APIs and programming language libraries) which are stored in our experience repository, based on more than 10 years of analysis of several commercial systems at Fraunhofer.

In order to conduct complementary dynamic analysis, we use the results of the static analysis to a) define and automatically insert probes at the right location for collecting runtime events (e.g., call events), and b) store and/or forward runtime events to various tools that we use to analyze data. The probes are defined using a minimally invasive instrumentation technique based on the aspect oriented technology that was introduced in Chapter 4. Our technical set-up for dynamic analysis takes advantage of the publisher-subscriber style by introducing a runtime event collector component (RECO) into the software architecture. Our probes emit runtime events as messages into the communication bus, which deliver them to the RECO. In a sense, dynamic analysis is seamlessly integrated into the publisher-subscriber style. By monitoring the running system, we interpret the runtime events for discovering a) component-connector (C-C) views of the publisher-subscriber style including components, ports and connections between ports as well as b) sequence diagrams including messages exchanged between different components indirectly using the intermediate bus or directly between components with the bus hidden. Our dynamic analysis is facilitated by the construction of Colored Petri nets (CP-nets), as used in Chapter 4. CP-nets are useful for recognizing pre-planned patterns in interleaved events. We used CP-nets because
runtime events of a publisher-subscriber architectural style are highly interleaved. For example, when one component is emitting subscribe events the other component could be in the middle of creating a connection to the communication bus.

The system under analysis is NASA’s GMSEC system, whose team has developed a software product line based on the publisher-subscriber architectural style. The proposed approach was followed for an independent analysis of the GMSEC implementation. We discovered a) a middleware abstraction layer that offers vendors’ independent abstract interfaces to interact with the communication bus of different vendors, b) C-C views, c) sequence diagrams, and d) some behavioral violations resulting in a high-priority bug due to inconsistencies between the implementations of the same interface for different vendors.

**Contributions of the chapter:** We have found little discussion on the topic of analyzing structure and behavioral constraints of architectural styles in the reverse engineering community. To this end, we hope this chapter contributes the following:

1. A practical approach for analyzing the publisher-subscriber style using a combination of static and dynamic analysis. The technical set-up and analysis questions are also reusable on other systems based on the same style.
2. The reverse engineered GMSEC architecture is also reusable, offering novel insights on how to design and implement a middleware abstraction layer, which frees organizations from being vendor-locked.

### 5.3 Approach

In this section, we will introduce the “high-level” concepts of the publisher-subscriber style, which will be used to derive a set of questions for the architecture analysis. In order to answer the questions, we will present the approach for discovering a set of views, using both the static and dynamic analysis.

#### 5.3.1 Concepts of the Publisher-Subscriber Style

In the publisher-subscriber style, each participant can play the role of a publisher of message, a subscriber of messages, or both. Messages are key entities in this style. Typically, each message has a subject (a.k.a. topic) as well as a structure containing a number of fields and values that carry the data to be sent [72]. The central artifact is the broker or the bus (a.k.a. software bus). A typical bus has elements for connection management, subscription management, message buffer and routing management, as well as interfaces for publishing and subscribing, as shown in Figure 5-1. The connection management is used by the publishers/subscribers for connecting to and disconnecting from the bus. Subscribers use the interfaces of the bus to subscribe to messages of interest. Similarly, publishers use the interfaces of the bus to publish messages, which are stored in an internal buffer, often created and managed by the software bus [72]. The subscription management is used by the bus to manage the list of subscribers. One important job of the bus is to route the messages to appropriate subscribers, thus it needs to have message routing management concepts. Note that the bus can also play a role of publisher and/or subscribers by using its own interfaces.
There are several implementations of software buses on the market, typically based on middleware. Systems that make use of a middleware may want to hide/wrap and even generalize the interfaces of the middleware in order to avoid being dependent on a particular middleware from a particular vendor. To provide true flexibility, the programming language used to implement the bus should not dictate the programming language to be used by the subscribers and publishers.

5.3.2 Deriving Analysis Questions from the Style

In order to understand and assess the implementation of systems based on the publisher-subscriber style, we now derive a set of typical questions based on the high-level concepts of this style, introduced above.

1. Are there any middleware used for implementing a software bus?
2. If middleware is used, is there any middleware abstraction layer that hides the knowledge of a particular middleware API?
3. Can publishers and subscribers be implemented in different programming languages? If yes, how are the variants in the languages managed?
4. Can publishers and subscribers come-and-go dynamically at runtime?
5. Can publishers, subscribers, and the software bus run on different machines?
6. Which subscribers receive messages from which publishers and in what order?
7. Do subscribers receive messages that are not subscribed by them?
8. Can subscribers subscribe to the same message more than once without an intermediate unsubscribe?
9. Can subscribers unsubscribe to messages that are not subscribed by them?
10. Are there timing delays in delivering messages to subscribers?

The above set of questions is concerned with both the structural and the behavioral aspects of implementations. For instance, questions 1-5 deal with the module-structure of the system, whereas questions 6-10 deal with behaviors. Therefore, our approach has a static and a dynamic analysis phase. In the static analysis phase, we discover key header files, classes, interfaces, and routines related to the software bus concepts. In addition, we create “box-and-lines” views of modules related to the software bus concepts that we have discovered. In the dynamic analysis phase, we use the results from the static analysis in order to a) create probes that monitor the running system,
and b) for pattern recognition of runtime events. The data from the probes is used to
discover components and connectors and are documented as component-connector
views and sequence diagrams. The discovered relationships depicted in component-
connector views offer insights related to the runtime structure of the software, which
are difficult to obtain using static analysis techniques. The views are used to understand
the implemented architecture and to draw conclusions on the systems’ implementation
quality. Now, we explain how we perform static and dynamic analysis to answer these
questions.

5.3.3 Static Analysis Strategies

Our goal of the static analysis step is to locate “high-level” concepts of the publish-
subscriber style in the source code. We automatically extract dependency models (e.g.,
Include and Call relations) from the source code and analyze them semi-
automatically. Our static analysis strategy is based on observations that systems
typically do not implement the publisher-subscriber style from scratch; instead they
build on top of external entities such as middleware frameworks offered by commercial
or open source vendors.

For the analysis, we use a set of tools and other resources that have been developed
during the past 10 years at Fraunhofer. For example, we stored the names of header
files, classes, and methods/functions of frameworks and programming language
libraries that are architecturally-significant for each architectural style. For example,
Table 5-1 shows a small snippet of the stored data for two middleware vendors, namely
the Tibco and the Apache Active MQ. We store this data in a relation database model,
and use it in conjunction with dependency models in order to locate the files that are
involved in the implementation of the software bus concept. For querying the
dependency model, we use relation partition algebra (RPA) [78], defined in Appendix
B. The search result is used to a) discover the presence of any abstraction layers or
wrappers to such vendor libraries, and b) detect potential architecture issues. For
example, if the system under analysis uses vendor libraries directly without any
intermediate wrappers then it indicates a potential design issue, because the system is
now vendor-locked, impeding switching to another vendor’s solution.
5.3 Approach

Table 5-1: An excerpt of middleware vendors’ APIs

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Method Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tibco Smart Sockets (SS)</td>
<td>SSConnection::SSConnection</td>
<td>Initializes connection to the middleware</td>
</tr>
<tr>
<td>Tibco SS</td>
<td>SmartSockets::TipcSrv::send</td>
<td>Publishes the given message</td>
</tr>
<tr>
<td>Tibco SS</td>
<td>SmartSockets::TipcSrv::setSubscribe</td>
<td>Subscribes to the given message</td>
</tr>
<tr>
<td>Apache Active MQ</td>
<td>CMSConnection::CMSConnection</td>
<td>Initialize connection to the middleware</td>
</tr>
<tr>
<td>Apache Active MQ</td>
<td>cms::MessageConsumer::setMessageListener</td>
<td>Subscribes to the given message</td>
</tr>
</tbody>
</table>

In order to reason about how the system handles programming language variants, we locate dependencies to common header files and trace backwards. For example, JNI [142] and XS [311] are respectively used for communication between Java to C++, and Perl to C++. Using this knowledge, we can locate files that are dealing with more than one programming language. We attempt to understand how the language-to-language translation is separated from other concerns.

As part of our reverse engineering activities, we also keep track of typical variable and routine names developers use for implementing different architectural styles of systems. For example, in a previous case study, programmers used “publish”, “subscribe”, “sendMsg”, “rcvMsg”, and “unsubscribe” to implement the publisher subscriber style in the C language [97]. We use these sets of keywords to search the code base and/or the extracted dependency models to recognize the presence of potential architectural styles in the source code. To facilitate searching, we developed a robust implementation of the vector space model, based on [249] and [8], which allows us to search the source code base and outputs a ranked list of files similar to a given list of such keywords. This search technology also facilitates the static analysis of systems that support different programming languages for implementing the publisher and subscriber style. For example, we can select the Java implementation of the software bus and request for files “similar” to the given set of files, which might implement the software bus for other programming languages.
We stop the static analysis after getting answers to questions 1-5. Since discovery is an iterative process, we often conduct more static analysis as dynamic analysis raises new questions. The output of the static analysis is the discovery of the presence (or absence) of significant components as well as key header files and/or interfaces corresponding to the “high-level” concepts shown in Figure 5-1. This output feeds into the dynamic analysis activities.

5.3.4 Dynamic Analysis Strategies

Our goal of the dynamic analysis is twofold. First, for discovering components and connectors in the implementation, and create corresponding component-connector views of the publisher-subscriber style. Second, to create sequence diagrams, showing messages exchanged among the publishers and subscribers, including the support for hiding the intermediate software bus. We need dynamic analysis because a) there is an inherent dynamism in the publisher-subscriber style, meaning that publishers/subscribers can connect and disconnect as each individual component feels necessary. Thus, the actual architectural configuration of the system is often only known at runtime, and b) questions related to ordering of messages and timing are not easy to answer statically, if not impossible. The correctness of systems based on the publisher-subscriber style depends on the correct functioning of the software bus, and also depends on the publishers and subscribers using the software bus in the right way. A misbehaving component could easily prevent it from functioning properly.

Now, we will enumerate a few key challenges related to such analysis and how we address them in dynamic analysis of the publisher-subscriber style.

First, in order to minimize both the overhead of injected monitoring code and the amount of runtime data, we need to locate the right spots where the necessary data can be collected. We address this challenge by inserting probes that monitor the usages of the interfaces of the software bus that were discovered using static analysis. Depending on the context factors (e.g., software architecture, programming languages, and organizational boundaries), we choose the most appropriate instrumentation strategy for defining and inserting probes. In Chapter 4, we used runtime weaving using Aspect-J for Ricoh’s photocopy machine. In the context of the publisher-subscriber style, we can design probes to emit runtime events, as messages with the special subject “trace”, into the software bus itself.

Second, we need to collect the emitted runtime events and forward them to analyzers. In our approach, we introduce and integrate a special component called the Runtime Event Collector (RECO) into the publisher-subscriber architecture, which subscribes to all messages with “trace” as the subject.

Third, we need to systematically handle interleaved runtime events of the publishers and subscribers for discovering component-connector views and sequence diagrams. For instance, when one publisher is in the process of creating a connection to the software bus, another subscriber might already have subscribed to another or the same message, and at the same moment, another subscriber might be waiting for other set of messages to come in. Note that publishers and subscribers could run on different machines, processes and threads. As a consequence, the emitted runtime events are highly interleaved and difficult to analyze.
We tackle the challenge of analyzing interleaved events by using Colored Petri nets (CP-nets), which are shown to be a useful executable formal language for handling asynchronous systems behavior [95]. CP-nets are well-elaborated in [95]. Basically, a CP-net is a graph with two types of nodes called places and transitions. Tokens are entities with data attributes/fields, associated with places. Each transition can have one or more input places and one or more output places. Every transition has a precondition which is a Boolean expression associated with input places. A transition fires if there is at least one token in each of the input places that satisfy the precondition. If a transition fires, it removes one input token from each of the input places. Every transition has an action, which is a sequence of assignment statements that assign tokens to output places when transition fires. A prototype implementation of CP-nets was developed at Fraunhofer [180] and [103].

Based on the publisher-subscriber style and its implementation concepts, which was discovered during static analysis, we constructed CP-nets that process the incoming runtime events and recognize pre-planned patterns, and produce the data necessary for constructing C-C views and sequence diagrams. By pre-planned patterns, we mean the implementation constructs corresponding to architectural constructs. For example, calls to the “publish” method of the Connection class correspond to the abstract publishing concept in the style. The output of CP-nets is used to visualize both C-C views and sequence diagrams that hide the software bus. If we do not hide the software bus, all communication will be between a component and the software bus. We developed Dyn-SAVE to automatically create and visualize sequence diagrams based on output from such CP-nets.

It is worth noting that CP-nets run in parallel to the system under analysis and interprets the runtime events. Thus, it is a novel formalism for building architectural monitoring and compliance checking at runtime for dynamically reconfigurable systems. In our approach, we also use CP-nets for checking behavioral constraints at runtime. For example, we can check whether a subscriber receives any message other than what was subscribed as follows: One transition of the CP-net can wait for subscribed events and output, to one of its output places, the list of subscribed messages by subscribers. The second transition of the CP-net can wait for events that read messages from the software bus to occur, and output, to one of its output places, the list of messages read by each subscriber. The third transition of the CP-net, designed to consume the output of the above two transitions, can output to its one of the output places, the list of unwanted messages wrongly routed by the software bus. Figure 5-2 summarizes the conceptual elements of our dynamic analysis environment for analyzing systems based on the publisher-subscriber style. It is worth noting that the RECO component is in fact plugged into the running system – just like other components – probes can also be injected into the RECO component in order to make sure this trace collection component uses the software bus in the right way.
5.4 Analysis of the NASA’s GMSEC

5.4.1 Objectives of the Case Study

The NASA’s GMSEC branch has developed the GMSEC software architectures as a reusable framework for missions inside and outside the NASA. In addition to their rigorous reviews and testing, they also prefer an independent organization to review the implementation quality, and report to them architecture/design issues as well as behavioral problems that could lead to failures.

5.4.2 General Process for the Analysis

In the first part of the analysis, the NASA’s GMSEC team provided the GMSEC framework 2.6 as well as some example applications that illustrate the publisher-subscriber architectural style. The Fraunhofer team then analyzed this version statically from the point of view of product lines and software architectures. This analysis led us to understand the implemented software architecture of the GMSEC framework. After the analysis, the Fraunhofer team presented the discovered architectural issues to the GMSEC team, which addressed some of the high-priority issues in versions 3.0 and 3.1. The GMSEC team then provided the 3.1 version and a set of real applications for analysis. The Fraunhofer team set up a test-bed for running and performing dynamic analysis of the GMSEC. This fruitful process, which is supported by NASA IV&V’s Software Assurance Research Program (SARP), was started in year 2009 and was active at the time of finalizing the thesis.
5.4 Analysis of the NASA’s GMSEC

5.4.3 Static Analysis of the GMSEC

The GMSEC source code contains several programming languages (C, C++, Java, and some Perl). We extracted code-level dependency models (e.g., Include, Import, and Call relations) and stored them as binary relations for querying using the RPA. We briefly explain how the dependency models were used to discover the software bus and the middleware abstraction layer in GMSEC.

Our approach is a combination of bottom-up and top-down strategies. Recall that we stored a list of middleware frameworks and the names of header files, classes, function/methods that deal with concepts such as connecting to the middleware, sending, and receiving messages, etc., (see Table 5-1). In the bottom-up strategy, we queried the models, using RPA, for all usages of commercial middleware frameworks and sockets. This led us to locate the directories and files of the GMSEC framework that implement the concepts of the publisher-subscriber style. We lifted the file-level dependencies to directory-level dependencies using the RPA’s lift operator. Our conclusion is that the GMSEC framework has a clean implementation that separates the concerns of using and supporting several commercial middleware from providing software bus services to subscribers and publishers.

The separation of concerns is implemented using a set of wrappers, one for each external middleware framework, as well as for the standard socket library. Each wrapper provides the same set of services to publishers and subscribers, thereby hiding the differences between different middleware. Thus, usages of external vendors’ libraries are only allowed through a wrapper. For example, the ICE (Internet Communication Engine) is a commercial middleware that offers APIs for developing systems based on the publisher-subscriber style. The GMSEC framework offers a wrapper called ice that accesses the ICE APIs. All vendor libraries supported by the GMSEC framework are wrapped in the same way, see Figure 5-3, where boxes are directories (except socket.h and stdsoap2.h and Connection). Arrows denote the direction of module dependencies. The filled arrow denotes that each module within the wrapper folder inherits from the Connection class, which offers interfaces for publishing, subscribing, etc.

All wrappers are fully implemented in C++. Each wrapper inherits from the abstract base class called Connection which contains interfaces for connecting to the middleware, publishing, subscribing, etc. In addition to commercial middleware, the GMSEC team also implemented their own middleware, which is called the software message bus (mb), based on standard sockets. We queried the dependency models to understand who uses each wrapper of the middleware vendors, which showed that there are no static dependencies to the wrapper folder.
In order to understand this architectural design, we also used a top-down strategy using the simple example applications offered as part of the GMSEC distribution. The examples clarified that there is a class called ConnectionFactory, which is responsible for dynamically loading the wrapper of a vendor at runtime based on configuration settings. The build process of the wrapper showed that there is a dynamically loaded library (dll) for each vendor. For example, the wrapper implementation of the ActiveMq middleware is compiled into gmsec_activemq.dll for Windows, and gmsec_activemq.so for Linux. The source code of the ConnectionFactory class revealed that each wrapper implements standardized interfaces (e.g., CreateConnection), which are called by the ConnectionFactory at runtime to initialize the wrapper by loading the corresponding dll. The CreateConnection method of the loaded dll creates an instance of the corresponding middleware’s connection class.

The core of the GMSEC is implemented in C++. However, the GMSEC also supports other publishers and subscribers being implemented in languages such as C, Java, and Perl. We analyzed the implementation of the core in order to understand how it handles variability due to programming languages. We used our text-based similarity tool for this purpose, and it revealed that there is an equivalent of Connection and ConnectionFactory class for each programming language. This was possible to discover automatically because the GMSEC team used the same method, variable names, and signatures in all programming languages. Since the GMSEC uses the
5.4 Analysis of the NASA’s GMSEC

“jni.h” file, which supports communication between Java and C++, it became clear that all calls to the Java implementation of the Connection, Connection.Factory are redirected to respective C++ method calls using JNI [142]. Similarly, all calls to the Perl version of the Connection, Connection.Factory are redirected to C++ method calls using the XS Perl to C++ interface [311].

To sum up the results from the static analysis, an attractive aspect of the GMSEC framework is that flexibility is built into the architecture, meaning that a) missions can easily add new middleware vendor of their interest by inheriting and implementing the abstract base class (Connection), b) missions can switch between different middleware using configuration settings and without changing the source code; the Connection.Factory class will take care of loading and binding to the selected middleware wrapper, c) applications (i.e., publishers/subscribers) are agnostic to middleware vendor’s API because they program to the vendor independent abstract base class, d) applications can be programmed in different languages, and e) applications can freely enter at runtime by connecting to the running software bus. From the static analysis, we understood the structure of the GMSEC, key interfaces, and classes involved in the publisher-subscriber style. We will now use this knowledge to analyze behavioral aspects of the style using dynamic analysis.

5.4.4 Dynamic Analysis of the GMSEC

During the static analysis, we observed that dependencies among applications of the GMSEC are impossible to extract statically because all communication is indirect using the intermediate software bus using middleware. Thus, it is difficult to determine exactly which application sends and receives messages. Therefore, we also conducted dynamic analysis, and customize Figure 5-2 to the GMSEC.

5.4.4.1 Defining Probes of the GMSEC Application

All “real” applications given to us are implemented in Java. Therefore, we have chosen AspectJ as the language for inserting probes [13]. Because the static analysis showed us that the Connection class is the core class for connecting to the middleware, publishing, subscribing messages, etc., we injected probes before and after the invocation of the methods of the Connection class in each application. We inserted probes “before” and “after” so that a) we could calculate the execution time of each method, and b) we could capture parameter values, which might be updated due to call-by-reference. We weaved our probes into the compiled binary class files of the GMSEC applications, which use the GMSEC framework. Our probes emit runtime events as messages (with subjects “trace.before” or “trace.after”) using the APIs of the Connection class, resulting in the publication of runtime events using the software bus itself. It is worth noting that we can use any middleware, for example a middleware, different from the one used for publishing/subscribing “real” messages, to send out trace messages. We use different middleware for “trace” and “real” messages in order to avoid any communication conflicts, see Figure 5-4.
5.4.4.2 Developing the RECO component

We developed the RECO component using the GMSEC APIs, and thus it was plugged into the GMSEC runtime environment like any other GMSEC compliant application. The RECO plays the role of a subscriber by subscribing to all messages with the subject “trace.before” or “trace.after”. One precondition for the RECO component is that it should use the same middleware that was used by the probes; otherwise, the trace messages will not be delivered to it by the GMSEC software bus (see Figure 5-4). We run the RECO component in monitoring mode in order to verify that it follows the behavioral constraints of the publisher-subscriber style and that it works well with all configurations of the middleware type. This is why there is a bidirectional arrow between the RECO component and the software bus for trace message. The RECO component reads traces of other applications and publishes its own traces as shown in Figure 5-4, where arrows denote data-flow, and each filled bubble is a GMSEC application. Two software buses are respectively used for publishing “trace” and “real” messages. RECO is the component for collecting traces emitted by other applications, including its own traces. Note also that the RECO component was configured and deployed to run on a different machine, similar to other GMSEC applications.

![Diagram](image_url)

Figure 5-4: The setup for dynamic analysis of the NASA GMSEC.

5.4.4.3 Discovering C-C views and Sequence Diagrams

We recall from Chapter 4 some main aspects of CP-Nets, which offers an in-depth discussion on how CP-nets were used for analyzing the pipe-and-filter architectural style of Ricoh’s photocopy machine software. We used the same concept to perform analyses of the publisher-subscriber architectural style below. Here, we informally explain the design of CP-nets for discovering the C-C view of the publisher-subscriber style. We designed our CP-nets in a modular fashion, meaning that it was a composition of several CP-nets such as a) One CP-net for recognizing the creation of a connection to the software bus by monitoring call events to the Create method of the GMSEC API (i.e., to the ConnectionFactory class explained in static analysis), b) One CP-net for creating the connection port used for publishing messages on the software bus by monitoring call events to the ‘publish’ method of the Connection class, c) One CP-net for creating the connection port used for subscribing messages on the software bus by monitoring call events to the subscribe method of the
Connection class, and d) One CP-net for attaching the ports of publishers with subscribers if one party consumes the messages published by the other party. Matching the subject of the published message with the subscribed message is the key activity of this CP-net.

In order to discover the C-C view, we ran the constructed CP-nets on events emitted by the running system. We used the RECO component, which places each runtime event into the different places that are responsible for holding call-events. Different parts of the CP-net processed the call-events, as explained above, and produced the C-C view as a collection of tokens. Here, we have manually drawn the C-C view by using the discovered high level events; see Figure 5-5 for an example, where each box is a runtime process and the names of applications are placed in brackets. In addition, we can see all connections to the software bus that are created by each application and can determine how many they are, for example.

Figure 5-5: An example C-C view discovered using runtime events.

All applications communicate with the RECO component because it consumes the runtime events that are published by the other applications. When we showed Figure 5-5 to the GMSEC team, they mentioned that this view is very useful as it nicely captures inter-application communication at a high level of abstraction, a view that is normally difficult to create. The good news is that there are no surprising dependencies between the applications. However, one of the developers mentioned that he did not know the fact that the CAT has 3 connections to the GMSEC bus for publishing messages to other GMSEC applications. Thus, this view was used by developers to understand exactly the dynamic architecture of a complex system. In order to understand how messages are exchanged among different parties of the publisher-subscriber style, our CP-nets used the subjects of the messages that were subscribed by the subscribers and the subjects of the messages published by the publishers. If the subjects of the messages match, then our CP-nets store the connection between
publishers and subscribers, the messages, and sending and receiving time of the messages. We visualized that output using the Dyn-SAVE tool. We can visualize messages and their data fields, too, as shown Figure 5-6.

![Diagram showing sequence and timing information](image)

**Figure 5-6**: A sequence diagram with timing information.

### 5.4.4.4 Detection of a High-Priority Bug

Here we briefly explain one of the bugs in the GMSEC framework, which is due to inconsistencies in the implementation of the same abstract interface (Connection) by different wrappers of middleware vendors, see Figure 5-3. We developed three CP-nets that check constraints of the publisher-subscriber style. One CP-net keeps track of all calls to the “subscribe” method of the Connection class, another CP-net keeps track of all calls to the “unsubscribe” method of the Connection class. A third CP-net detects multiple calls to the subscribe event, without an intermediate unsubscribe.

The CP-nets reported that the RECO component subscribed to the same message more than once. The GMSEC API has a feature that allows applications to have a call-back capability when a message arrives from other applications. We used that feature and subscribed to the same message three times, because we wanted to print “trace” messages in three different formats using three different call-backs. When we used the GMSEC’s software bus (i.e., mb in Figure 5-3) to send and receive “trace” messages,
the return code of all three calls to the subscribe method was NO_ERROR. We analyzed how activemq behaved for the scenario and switched from the mb to the activemq middleware wrapper. Its subscribe method returned MIDDLEWARE_ERROR reporting that the RECO component makes multiple subscriptions to the same message.

We showed the RECO implementation to the GMSEC team, and discussed the behavioral inconsistency between the two middleware wrapper implementations of the same abstract interface, which caused the RECO to fail when we switched from one middleware wrapper to another wrapper. They agreed that this is an important bug and fixed it so that all middleware wrappers will behave in an equivalent way with respect to their return code. Otherwise, applications cannot reliably choose and/or switch between different middleware wrappers. We would not have detected this important bug unless we modeled and verified the runtime behavioral properties of the publisher-subscriber style by collecting runtime events.

5.4.5 Answering the Questions

Table 5-2: Answers to the analysis questions

<table>
<thead>
<tr>
<th>Question</th>
<th>Answers/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Are there any middleware used for implementing a software bus?</td>
<td>Yes. The GMSEC software bus is implemented using middleware technology.</td>
</tr>
<tr>
<td>2. If middleware is used, is there any middleware abstraction layer that hides the knowledge of a particular middleware API?</td>
<td>Yes. There is an abstraction layer that “hides” vendor-specific APIs.</td>
</tr>
<tr>
<td>3. Can publishers and subscribers be implemented in different programming languages? If yes, how are the variants in the languages managed?</td>
<td>Yes. The core of the GMSEC is implemented in C++. However, there are interfaces for C, Java, and Perl. Language variants are managed using JNI [142] and XS [311].</td>
</tr>
<tr>
<td>4. Can publishers and subscribers come-and-go dynamically at runtime?</td>
<td>Yes. Publishers and/or subscribers can freely enter/leave the running system.</td>
</tr>
<tr>
<td>5. Can publishers, subscribers, and the software bus run on different machines?</td>
<td>Yes. Publishers and subscribers just need the IP address and the port number of the software bus.</td>
</tr>
<tr>
<td>Question</td>
<td>Answer</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>6. Which subscribers receive messages from which publishers and in what order?</td>
<td>The discovered sequence diagrams answer this question for the scenarios we have executed.</td>
</tr>
<tr>
<td>7. Do subscribers receive messages from connections that are not subscribed by them?</td>
<td>No. However, we cannot extrapolate because dynamic analysis results are not generalizable.</td>
</tr>
<tr>
<td>8. Can subscribers subscribe to the same message more than once without an intermediate unsubscribe?</td>
<td>This is a bug the GMSEC team fixed it based on our analysis.</td>
</tr>
<tr>
<td>9. Can subscribers unsubscribe to messages that are not subscribed by them?</td>
<td>No, not for the execution traces we analyzed.</td>
</tr>
<tr>
<td>10. Are there timing delays in delivering messages to subscribers?</td>
<td>At the time of finalizing the thesis, we were analyzing timing aspect for different middleware configurations using the discovered sequence diagrams.</td>
</tr>
</tbody>
</table>

### 5.4.6 Connecting the GMSEC’s Business Goals with Architecture

In order to understand the relationship between the GMSEC’s business goals and the implemented high-level architectural decisions, we discussed with the GMSEC project manager, the product leader, and senior engineers. Based on this discussion, we were able to explicitly link the business goals and software architectural decisions (see Figure 5-7).
5.5 Brief Comparison to Existing Work

It is interesting to note from this study that important architectural decisions were primarily influenced by business goals. Therefore, it is advisable that the implemented architecture needs to consist with the specified architecture, as shown to be true for the GMSEC with some exceptions mentioned above.

**5.5 Brief Comparison to Existing Work**

We will first list a few related articles and then highlight common differences to our work. Riva and Rodriguez [243] extract both the module-structure and sequence diagrams by respectively using static and dynamic analysis. Wendehals and Orso [305] extract automata using execution traces. Giannakopoulou and Havelund [114] use linear temporal logic (LTL) to verify behavioral properties of execution traces. Schmerl et al. [251] use the pair of architecture and implementation styles to discover C-C views of a running system. Strouli and Systa [276], and Cornelissen et al. [61] provide an in-depth survey on other dynamic analysis techniques. Dong et al. [68] review methods and research tools for recognition of design patterns from the source code. Briand et al. [39] extract present an elaborated approach for extracting sequence diagrams of distributed systems.

Key differences between our work and the existing work are: we extracted component-connector views, which showed the runtime structure of the software. We used CP-nets to systematically tackle the interleaving of runtime events with respect to the software architecture. In contrast to DiscoTect, we used static analysis to locate the implementation style of the specified architectural style [251]. Our sequence diagrams can a) hide the software bus for analyzing the publisher-subscriber style, and b) show
attributes/parameters of messages exchanged between parties. If we excluded parameters, like in many existing work, we would not have been able to distinguish calls to the subscribe method on two different message subjects, for example. Cornelissen et al. mentioned that there is a short-coming of research, in reverse engineering, on systems that evolve at runtime like the GMSEC. Regarding design pattern discovery, our focus was on bridging the abstraction gap between runtime events and the publisher-subscriber style.

5.6 Closing Remarks

We presented a practical approach for analyzing systems based on the publisher-subscriber architectural style. We derived a set of analysis questions, which focused on both the structural and behavioral constraints of the style. First, we performed the static analysis to answer the questions related to the structural constraints. Second, we used the results of the static analysis to organize the dynamic analysis for answering behavioral constraints. We discovered component-connector views and sequence diagrams using execution traces, which were fed into our Colored Petri nets, for tackling the challenge of interleaving of runtime events. Using this approach on the NASA’s GMSEC, we discovered that the GMSEC has a) a good middleware abstraction layer, which helps in avoiding vendor lock-in, and b) a high-priority bug due to behavioral discrepancies among different middleware wrapper implementations. At the time of writing the thesis, we were working on analyzing timing aspects of different middleware wrappers. In addition, we were exploring advanced parser technologies [173] and [174] to overcome some of the limitations of our current parsers.
Chapter 6
Architecture Discovery and Analysis Method (ADAM)\textsuperscript{8}

6.1 Abstract

In this chapter, we propose the Architecture Discovery and Analysis Method (ADAM) for analysis of quality properties, such as testability, performance, and maintainability. The premise of the ADAM is that architecture decisions and quality properties are inspired and influenced by the external entities that the software system uses. Examples of such external entities are COTS components and the programming language libraries. Traces of these architecture decisions can thus be found in the implemented software and manifest in how the software system uses such external entities. The ADAM is demonstrated using the NASA’s Space Network Access System (SNAS). The results show that the method offers reusable and repeatable guidelines for discovering the architecture and locating potential risks (e.g., low testability, decreased performance) that are hidden deep in the implementation. The analysis is conducted by using external dependencies to identify, classify and review a minimal set of key source code files supported by a knowledge base of external entities and analysis questions with strategies for obtaining answers. Given the benefits of analyzing external dependencies as a way to discover architectures and risks, it is argued that external dependencies deserve to be treated as first-class citizens during reverse engineering.

6.2 Introduction

At Fraunhofer CESE, we analyze customers’ existing software systems, for example in order to detect testability, performance, and maintenance risks. The customers of CESE want an “independent eye” to look into their implemented software systems, evaluate the implemented architecture, identify high risk areas, and propose practical suggestions for improvements and risk mitigations. Naturally, the customers expect the analysis results to be delivered “as soon as possible” so that they can effectively make use of the findings in their decision making process, incorporate improvements into their products and processes, remove issues, and meet their goal to produce a high quality software product on time. With this pressure to deliver critical and accurate architecture insights regarding previously unfamiliar software systems in relatively short time, CESE is continuously seeking ways to improve and make their analysis methods more efficient. The ADAM, introduced in this chapter, is the result of more

\textsuperscript{8} At the time of writing the thesis, this chapter was under review by the editorial board of the ACM Transactions on Software Engineering and Methodology (TOSEM).
than 10 years of such analysis and accompanying improvements of two dozen industrial systems.

One of the more fundamental insights that we have gained through our work is the importance of being able to zoom in on and reason about individual software concerns and how they are implemented in the source code. Industrial software systems are no-doubt large and inherently complex. Apart from offering user-oriented features, software systems manage multi-dimensional concerns that are somewhat hidden from the user. Examples of such concerns are that the software must run on several operating systems (OS), that it must handle data in a persistent way, that it must provide for dynamic loading and unloading of components, that it must allow for restricting the accessible and enabled features based on license validation etc. [175].

In the software systems we have analyzed, any given source file typically addresses more than one concern and each concern is typically distributed across more than one source file. For example, a source file may contain code that executes a database SQL query as well as code that writes each database interaction, as well as other events, to a log file. Thus, the code in this file addresses several concerns (i.e., database management and logging). In addition, several source files may contain code that is involved in these database interactions. Thus, code that addresses the database interaction concern is spread across several source files.

We can conceptually imagine every source code file as being one point in a multi-dimensional space, where each dimension refers to a concern. Most readers will agree that it is beyond our capability to comprehend and visualize shapes in more than two or three dimensions. Hence, we need to build abstractions of the software under analysis that emphasize only the concerns we are interested in and suppress (for a moment) everything else. Since the software under study is typically represented by source code only (documentation is often lacking), we need to create these abstractions using entities found in that source code. Thus, we can say that we need to identify selected implementation concepts in order to recognize the implementation of architectural abstractions such as layers, styles, components, and connectors that are typically used to express the high level software architecture and which are often “hidden” in the multi-dimensional space of concerns in the source code.

Naturally, several concerns are often implemented using the support of external entities (e.g., programming language libraries and COTS software), instead of developing homegrown source code. Thus, the premise of the ADAM is that dependencies to the very same external entities that were used to build the system can not only be used to efficiently discover its implemented architecture but also to reason about quality properties, in particular, testability, performance, and maintenance risks. The opposite of external entities are internal entities, which are the software items that belong to a particular system. Whether or not to consider a particular software item as external or internal often depends on whether the source code is maintained by the project team. External entities are typically not maintained by the project team.

---

9 [http://www.few.vu.nl/~rkrikhaa/adam/](http://www.few.vu.nl/~rkrikhaa/adam/) has the major list of systems that were analyzed by us.
To our knowledge, the ADAM is the first method that makes use of external entities to reason about testability, performance, and maintainability (see Related work is in Section 6.3 and Section 6.8). We analyze such quality properties using a multi-dimensional framework because testability, for example, is influenced by several factors, such as how the GUI, hardware elements, OS, and communication are abstracted. We perform reverse architecting in an incremental fashion, in that each increment is detailed and precise with respect to the selected concern.

In contrast to the existing literature, our method produces a suite of architectural diagrams that precisely capture each concern so that we can pinpoint quality problems that are deep in the code. Thus, separation of concerns is built into our architecture discovery method because we analyze one concern at a time.

In order to demonstrate that most software is heavily dependent on external entities, we measured the ratio between the number of dependencies to external entities to that of the total number of dependencies, see Table 6-1. We can infer that around 40% to 60% of all dependencies in typical software systems are to external entities.

Table 6-1: Distribution of internal and external dependencies

<table>
<thead>
<tr>
<th>System name</th>
<th>File Dependencies</th>
<th>Internal Dependencies</th>
<th>External Dependencies</th>
<th>Ratio of External vs Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA SNAS</td>
<td>28388</td>
<td>12419</td>
<td>15969</td>
<td>0.56</td>
</tr>
<tr>
<td>NASA GMSECAPI</td>
<td>2388</td>
<td>1441</td>
<td>947</td>
<td>0.40</td>
</tr>
<tr>
<td>NASA SPSR</td>
<td>53682</td>
<td>22305</td>
<td>31377</td>
<td>0.58</td>
</tr>
<tr>
<td>DRLM</td>
<td>5205</td>
<td>2498</td>
<td>2707</td>
<td>0.52</td>
</tr>
<tr>
<td>CCIS</td>
<td>6436</td>
<td>2690</td>
<td>3746</td>
<td>0.58</td>
</tr>
</tbody>
</table>

The proposed ADAM, presented in Section 6.4, is based on the fact that much of the critical information about an existing software system is stored in source files, and thus an analyst has to review such files in order to understand critical parts. For a small system, it is not a problem to analyze each and every file of the system. However, for larger systems there are typically too many source files (10,000 files is not unusual) for the analyst to review. Thus, we need a way to identify the most important parts of the source code that influence the architecture for review.

We have discovered that dependencies to external entities are useful to identify the parts of the source code that are most important – for the task at hand. Often an external dependency is based on a file name as well as a function name of the external entity. We have classified many of the commonly occurring external files and functions in such a way that we can select a perspective or category and can trace back to the files and functions in the software under analysis that use them. Thus, by reviewing only those specific parts of the source code, we can understand how a specific concern is handled by the system. This technique also allows us to reason about other parts of the system that have similar dependencies and we can draw conclusions about large portions of the source code without having to review it all. The acquired knowledge and insights of analyzing software systems are stored in a knowledge base, which on the one hand is used to analyze new systems more efficiently. On the other hand, it is used to improve our understanding of various real-world solutions to architectural challenges (e.g., how to architect the system so that database interactions can be by-
passed for unit testing purposes). These arrays of solutions are also discussed with our customers as alternative solutions to their architectural problems, if any.

In Section 6.5, using the ADAM on the relatively large (~500 KLOC) NASA Space Network Access System (SNAS) system, the analyst discovered several architectural insights by reviewing less than 4% of the 1578 source files including the discovery of the modular and the runtime inter-component communication structure of the SNAS, and the identification of testability, performance, and maintenance risks. Examples of architectural insights are a) that the implemented architecture of the SNAS is based on a distributed client-server architectural style, b) that the distributed subsystems exchange data by sending and receiving objects using the transfer object design pattern [5], c) that each subsystem of the SNAS has a dedicated layer for handling the persistence concern, and d) that the GUI subsystem is based on an event-driven architecture.

In addition, with the help of external dependencies, several architecturally relevant performance related constructs were discovered including the usage of a) a database connection pool design pattern in order to overcome the performance overhead of frequently creating and deleting database connections [10], b) the reactor design pattern in order to reduce the overhead of frequently creating and deleting threads for each client connection in a client-server architectural style [252]. Some testability problems due to a weak separation of GUI concerns from core logic were also discovered. Naturally, many of the SNAS maintainers are aware of the architectural discoveries that are discussed in this chapter.

**Contributions:** We believe that this chapter makes the following novel contributions:

- A practically inspired and validated method to discover software architectures from implementations using external dependencies.
- An architecture-centric framework for evaluating testability, performance, and maintainability of implemented systems without reviewing each individual file.
- Several concrete real-world code snippets to demonstrate the true meaning of abstraction, separation of concerns, and architectural design for testability and performance.

### 6.3 Existing Knowledge-based Methods

Here we compare the ADAM to related reverse engineering methods that also give importance to external entities during the architecture discovery process. Finally, we position the ADAM using the taxonomy of Ducasse and Pollet [69]. Additional related work that has some relevance to this chapter is discussed in Section 6.8.

The *X-Ray method* is an architecture reconstruction method for distributed systems [200]. The first step of the X-Ray method focuses on the extraction of a module dependency graph from source code, where a module equals a `.c` file. The second step of the X-Ray method focuses on identifying runtime components by partitioning the module dependency graph based on entry modules (i.e., those modules with a main function/method), exclusive modules (i.e., those modules that are only used by entry modules), and shared modules (i.e., those modules that are used by more than one
runtime component). Each entry module is grouped with its associated exclusive modules to form a runtime component. We identified a few challenges related to this step. The X-Ray method assumes that every module with a main function/method is an entry module and will be part of a separate executable runtime component. This is not always a valid assumption. For example, in NASA’s SNAS system there are 220 files that each has a main method. However, the majority of those main methods were introduced and used for testing purposes only. Using the X-Ray method, the analyst would thus wrongly believe that there are 220 runtime components in the SNAS system; instead there are only 7 runtime components in the production delivery. A complicating factor is the fact that it is not possible to distinguish between main methods that are used to launch production components and main methods that are only used to facilitate testing, unless the build process is analyzed. However, the authors of the X-Ray method explicitly state that they do not analyze build or configuration files because they consider them to be error-prone, see page 314 of [200].

Another discomfort is rising from the very definition of distributed systems used by the X-Ray method because it assumes that each runtime component has a main method or otherwise clearly defined entry point stated in the source code. However, we have analyzed distributed systems with only one main function even though there are many runtime components. For example, in NASA’s CFS system (see Chapter 2 and Chapter 3), each runtime component is started independently as a separate task allocated to one of the available processors, by dynamically loading its compiled object code and starting its entry function, which is not called main, as specified in a text configuration file. These runtime components then communicate indirectly with each other using a software bus. In this case, the X-Ray method would wrongly treat all modules of the CFS as one runtime component because there is only one main function. In contrast, an analyst using the ADAM would first analyze the external dependencies, which would reveal that the CFS uses external library functions for dynamic loading and unloading of modules. This would then lead to the discovery of the architecturally significant dynamic reconfiguration capability. In addition, it would lead to the discovery of the runtime structure made of several distributed components that could run on different machines and communicate using the software bus, despite having only one main function.

The third step of the X-Ray method uses pre-defined prolog patterns related to inter-process communication for locating code elements contributing to connectors. The X-Ray method requires an abstract syntax tree (AST) of the system under analysis. In contrast, the ADAM does not build an AST for pattern matching, because that would require the system under analysis to compile, which for example, requires all necessary header files to be present. In our analysis of embedded systems, we have come across many types of C language dialects based on different C standards. This increases the difficulty of parsing the code and constructing a precise AST. In addition, in an independent analysis of large systems, obtaining access to all necessary header files of external entities is difficult because of the need to obtain licenses, which may be expensive. Because of such practical constraints, and the high-pressure to deliver reverse engineering results within a short time frame, the ADAM adopts a relatively light-weight extraction of code relations. In addition, ADAM’s pattern matching is somewhat primitive, yet powerful. The ADAM uses search scripts against source code as well as relational algebra queries on extracted code relations. It is also worth noting
that the X-Ray method was validated on a small system (29 C source files) while the ADAM has been validated on large systems consisting of many thousand files.

In [231], Pinzger and Hall use the knowledge of experts, design documents, and external entities for recognizing architectural abstractions (e.g., Clients, Servers) in the source code. They use an XML-based tool called Enhanced String Pattern Recognition tool (ESPaRT), developed by Knor et al. in [161], for codifying the patterns to match in the source code. ESPaRT appears to have the capability to perform both structural and regular expression based pattern matching of external entities in order to locate what the authors call architectural hot-spots in the source code. In contrast to their approach, the ADAM does not employ domain experts during architecture discovery because on the one hand experts may not be available and on the other hand independent analysis and review is expected to be performed independently.

The ART method of Fiutem et al. [82] analyses ASTs using a knowledge base of patterns, called architectural clichés, to identify connectors at different levels of abstraction, such as the system level, the program level, and the module level. On the system level, ART extracts IPC (Inter-Process Communication) connectors (e.g., pipes and sockets) between components, which are identified after an analysis of makefiles. On the program level, ART treats function calls, variable accesses, and forks as connectors between tasks and directories/files. On the module level, ART treats function calls, data accesses between functions and variables as connectors. In contrast to the ART method, connectors are discovered and analyzed with respect to one concern at a time, and there is no notion of levels of abstraction of connectors in the ADAM. For example, in the ADAM, classes that provide abstractions of database concepts are treated as connectors with respect to the persistence concern, because such classes act as a bridge between a database and the rest of the system.

The ManSART method of Harris et al. and Yeh et al. ([124] and [314]) utilizes a library of architectural styles and corresponding code patterns to detect software architectures in the source code, and similar to the X-Ray and the ART methods it requires an AST as an input as well as a list of architecture styles that are expected to occur in the system. The ManSART method utilizes a combination of top-down and bottom-up reverse engineering techniques. The analyst selects the expected architectural style, which triggers the ManSART method to locate code elements that match the selected style. In contrast, the ADAM works mostly bottom-up, because in many cases it is difficult to know the architectural style to search for. In addition, it is difficult to codify all the different ways the various architectural styles can be implemented in. For example, one of the subsystems of the SNAS has a database abstraction layer comprising just one class. This class contains all SQL queries for interacting with several database tables. If such a pattern is not codified and stored in the library of ManSART, then it would not detect the presence of such a database abstraction layer. In fact, this limitation applies to the methods discussed above as well, because the user has to codify the architectural clichés in order to be able to search for such patterns in the source code. In contrast, the ADAM stores the names of external entities that are known to be used to implement certain concerns. The names are limited in number and there is no need to encode certain patterns of how these names might be used to form certain architecture structures. In addition, the names are stable and typically do not change over time.
In comparison to the methods discussed above, the knowledge base of the ADAM generalizes beyond communication concerns. It is interesting to note that parts of the knowledge base for the C/UNIX domain of those existing methods match the knowledge base of the ADAM. In addition, the knowledge base of the ADAM contains patterns for several architecturally significant concerns such as GUI, persistence, OS variants, error handling, and several COTS. The X-Ray, the ART, and the MANSART methods perform data and control flow analysis to make the matching of architectural recognition plans accurate. However, it is worth noting that if the source code slightly deviates from the knowledge base of their architectural recognition plans, then there is a risk that those architectural features are not detected in the code. In the ADAM, the architectural recognition plan is relatively lightweight, meaning that a certain predefined set of keywords (e.g., class names, header files) are used for searching and locating architecturally relevant features.

In the X-Ray, the ART, and the ManSART methods, layers are discovered by partitioning the module dependencies graph extracted from source code relations (e.g., using the Include relation among files). In these methods, modules involved in cyclic dependencies are grouped together in one layer. However, in many of the industrial systems we analyzed, cyclic dependencies commonly occur also between modules in different layers. Thus, the discovered layered structure may not match the conceptual layered structure. This is one reason why it is very difficult to automatically recognize layers from the module dependency graph. In our opinion, it is the semantics of the nodes of a module dependency graph that contribute to the layered structure and not necessarily the connectivity or the topology of the graph. Automatically recognizing the differences on the levels of abstraction among the nodes of a module dependency graph is difficult. This is also one reason why the ADAM lets the analyst detect layers using external dependencies as the driver. For example, in the CFS system, the knowledge base helped the analyst locate files that make use of the APIs of different operating systems, such as VxWorks, RTEMS, and UNIX. Using the ADAM, the analyst easily recognized the OS abstraction layer because different OS types were wrapped with a common OS independent abstract API with alternative implementations for different OS types. For another example, see Chapter 5, which explains how the knowledge base helped in detecting the presence of a middleware abstraction layer in the NASA’s GMSEC system.

It appears many of the existing methods do not leverage artifacts such as build scripts, batch scripts, and configuration files. In contrast, the ADAM leverages these artifacts because of architectural knowledge stored in them. In fact, some of the analysis questions, for example, what are the runtime components are difficult to answer if we exclude these artifacts, as discussed later.

Another major difference between the ADAM and other existing methods stems from the fact that the ADAM not only supports architecture discovery but also supports the evaluation of quality properties. For example, to analyze testability, the ADAM analyzes the system from the communication perspective and evaluates a) how deep the knowledge of a particular connector type penetrates the system, b) how well the system’s architecture abstracts the vendors’ communication or middleware APIs, and c) whether or not individual runtime components can be tested independently of other components. In general, we found very little discussion on such analysis of the
discovered architecture in the existing literature on reverse engineering. One exception is the work of [269], which discusses a quality attribute driven architecture discovery approach. Their method focuses on architecture discovery for analyzing modifiability and portability. In contrast to the ADAM, their method does not include analysis of testability.

In [69], Ducasse and Pollet present a taxonomy that is used to classify existing architecture reconstruction approaches along five axes, namely: goals, processes, inputs, techniques, and outputs. The ADAM can be positioned with respect to their taxonomy as follows. Goals: A typical goal of the ADAM is discovering the implemented architecture for analyzing the quality of the implementation with respect to various concerns. Processes: The ADAM follows a bottom-up approach to discover the architecture. Architecture discovery is conducted incrementally by focusing on one concern at a time. Inputs: Source code and a knowledge base constitute the inputs of the ADAM. Techniques: The ADAM utilizes a mix of semi-automatic and quasi-manual techniques. Extraction of source code relations as well as querying such relations using RPA [169] is semi-automatic, whereas reasoning about the extracted data and analyses of quality properties are mostly manual. Outputs: The typical outputs of the ADAM include a collection of views, each view explaining the architecture with respect to a particular concern. Code snippets are additional outputs that are used to precisely explain the implemented architecture and quality issues.

6.4 The ADAM

6.4.1 Terminology and Background

The architecture of a software system is the set of structures needed to reason about the system. The structures comprise software elements, relations among them, and properties of both [55]. A view explains the structure of the system with respect to one particular concern, where a concern is an area of interest or focus in a system, typically expressed by a stakeholder. Views facilitate separation of concerns since each view emphasizes certain facets of the software architecture while deemphasizing and ignoring other facets [33]. It is generally accepted that software architectures are typically complex enough that they must be described using several such views (e.g., [170] and [248]). These definitions apply to reverse engineering as well as to forward engineering. There is, however, a huge difference that is important to keep in mind. In forward engineering, we create views early in the software development process (i.e., before any source code has been developed) to describe the architecture of a system that does not exist yet. Thus, there are many details that are unknown and that will not be described in any of the views, but will be left for the design or the implementation team to describe.

In reverse engineering, we create views late in the software development process (after all source code has been developed) to describe the architecture of a system that does exist. Thus, there are many details that are available that can be used in the creation of reverse engineered views. Actually, in any larger software system, the source code provides such an enormous amount of information, which is scattered across a large number of files and directories, that the challenge is to find a starting point for the
6.4 The ADAM

analysis and then to filter out as much information as possible without losing anything important to explain how the source code addresses a specific concern.

Thus, views created as part of reverse engineering are based on sets of and relations extracted from the source code that are filtered and abstracted until the necessary characteristics emerge. The information in views can be presented in different ways depending on the current needs, for example as text, tables or, graphs. In the case when no filters are applied to the sets and relations extracted from the source code, then the tables and graphs contain all details and become large and difficult to handle. In the case when the information is filtered so much that only the structures on the highest level of abstraction remain, then even the largest system can be expressed in terms of a few entities (e.g., subsystems) and their inter-relations. These are the extreme cases and the analyst needs to be able to create views that are useful for the analysis task at hand. There is also a need for different types of views. For our reverse engineering purposes, we have found that module dependency views and runtime views are especially useful. In addition, we make use of other types of relations such as degree of similarity between modules to detect software clones and reason about clones at the architecture level.

Module dependency views consist of modules and dependency relations between modules. In reverse engineering, the most natural and most commonly occurring module is the source code file, which often has a descriptive name. We also consider source code elements contained in files, such as classes, routines, procedures, and functions as modules, if necessary. Files are often organized in a folder hierarchy depending on how closely related the files are to each other. In addition, folders often have descriptive names. Thus, the file and folder structure provides valuable information for reverse engineering, which we use to create an initial module structure. Modules are dependent on each other. For example, module A (which in this case is a file) depends on module B (which also is a file) if A includes (or imports) B, or if A accesses global variables and other data items in B. Especially useful is the Call relation that occurs if routine X calls routine Y because the Call relation can be used to create call graphs that are used in reverse engineering [69].

However, module views do not offer full insight into how modules are partitioned into runtime entities. Thus, we need to extract runtime views, which consist of components and connectors and are characterized by the fact that they have some presence during runtime. Runtime components are the processes, tasks, threads, and objects that execute when executable files are launched. Connectors are mechanisms that allow software components to communicate [260] and [198]. Examples of connectors are communication channels (e.g., queues, sockets, pipes, data files, etc.), shared memory, event dispatchers, and database queries. Our runtime views also contain ports, which are the points at which connectors connect to components. Ports can be communication port numbers, pipe ids, file ids, and so forth. Communication and synchronization are essential runtime activities of a multi-tasking system [6]. Thus, we also extract how the elements of runtime views communicate and synchronize with each other.

It is important to note that, in general, it is difficult to distinctly label software entities as either components, connectors, or ports because there are situations when a software entity can have more than one label. For example, the developers of a software bus may consider it a component while the developers who are using the software bus may treat
it as a connector for exchanging messages between other components. Similarly, the developers using the Java’s `PipedReader` class could treat it as a read port of a filter type component to read messages from a pipe, whereas the developers of `PipedReader` might view it as a component. What is considered to be a component in one view can be considered as a connector in another view, for example.

In [260], the authors examine a number of architectural styles (e.g., layered style, pipe-and-filter). An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with topological constraints and execution semantics on architectural descriptions. In our approach we detect the presence of architectural styles using the extracted module and runtime views. For instance, if we have a module view where the flow of control goes from top to bottom and our analysis indicates that the bottom modules are more generic than the top modules, then we conclude that the software elements described by this view are organized in layers and have the characteristics of a layered style. Similarly, in a runtime view, if we find that the tasks communicate using one or more pipes as connectors, and tasks do not share state and are not aware of each other, then we conclude that the components and connectors described by this runtime view have the characteristics of pipes and filters and organized in accordance with the pipe-and-filter style.

It is worth to note that many architectural styles are driven by the type of connector used, including pipe and filter, real-time data feeds [247], event-driven architecture [63], message-based style (e.g., [279], Chapter 2, Chapter 3, and Chapter 5) and dataflow style [260]. Thus, we first need to discover components and connectors before attempting to detecting styles. In some scenarios, we have found that we can discover architectural styles by first discovering the connectors and then by locating the components that interact with the connectors. A design pattern is a reusable approach to solving a commonly-occurring software design problem [88]. Design patterns are at a lower-level of abstraction in comparison to architectural styles [206]. Of course, one can leverage design patterns to implement a style, in Chapter 4, an Observer pattern was used to implement the pipe-and-filter style in Ricoh’s office appliances, meaning that filters that registered with pipes will be notified when input data arrives to the pipes.

### 6.4.1.1 Equivalent Modules

Because large software systems are typically made of a large number of modules, there is a need to limit the number of modules that must be analyzed in detail, for example to reduce analysis effort. In order to do so, we introduce the notion of equivalent modules. The idea is that only a small sample of a large set of equivalent modules needs to be analyzed in order to characterize all modules in the set. For a given concern, a set of modules is considered equivalent if: a) the modules share the same pattern of dependencies as well data and control flow, and b) the modules are independent of each other. We learned these constraints from our experience and found them useful to generalize a large set of modules into one abstract module. Depending on the concern of analysis, we will also include additional constraints to define equivalent modules.
6.4.1.2 Software Architecture vs. Design Details

One of the basic insights from the existing literature is that in order to be useful, architectural views should not mix several concerns (e.g., [170] and [248]). From our experiences with more than two dozen real-world industrial systems, we have learned that systems often have to deal with concerns including Legal Standards, GUI, Persistence, OS variability, License Management, Logging, Monitoring, Security, Internalization, communication among multiple systems and/or multiple languages. Thus, the software architecture must also describe these concerns, which is naturally achieved as a collection of views, each one focusing on one concern. Thus the software architect’s job is to describe all necessarily details related to each concern otherwise otherwise the described architecture is most likely too vague and of limited use as a basis for design and implementation.

It is important to note that the question of some architectural views being too detailed does not make any sense unless the context and role of the person who is commenting on the views is explicitly defined. For example, the NASA SNAS system described in this chapter is in fact a subsystem of the NASA SN system. From the SNAS architect’s point of view, it is important to describe the architecture of the SNAS for each concern. However, from the SN architect’s point of view, such views may be considered detailed design because the SN architect is most likely only interested in how SNAS fits into the SN architecture, and not necessarily interested in the SNAS’ internal architecture.

We all agree that the software architecture of any system is an abstract artifact. In our opinion, we sometimes need to use code elements to describe a view. For example, in order to explain how the system interacts with a database, the view should capture principles for how and where a) the connection to the database is established, b) the queries are prepared and executed, c) the database interaction errors are managed, and d) the database is abstracted together with necessary APIs. In our opinion, this view is abstract despite the fact that “low-level” syntax is present in the view because the view is free from algorithmic details.

Having introduced our basic model of software architectures, components, connectors, and equivalent files, the next section introduces our reverse engineering method.

6.4.2 The Structure of the Knowledge Base of External Entities

We define the structure of the knowledge base using RPA-based mathematical relations as follows (also see Table 6-2 and Table 6-3). An external entity keyword is an expression that could be part of a source code statement and is typically language specific. An example keyword is the java.sql portion of the java statement import java.sql. The relation \texttt{KPL\textsubscript{part_of}} denotes the assignment of a keyword to a certain programming language. For example, \texttt{java.sql} is the name of a java package that provides the API for accessing a database and thus is part of the extended Java programming language. This relation is stored as one row in the \texttt{KPL\textsubscript{part_of}} table in the knowledge base as <“java.sql”, “Java”>. Similarly, the relation \texttt{KOS\textsubscript{part_of}} denotes the assignment of a keyword to a certain OS type such as \texttt{taskSpawn} which is part of the VxWorks OS vocabulary and thus can be found in source code that makes use of VxWorks.
Table 6-2: Some relations of the knowledge base

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>KPL_{part_of}</td>
<td>Keyword</td>
<td>PL</td>
<td>keyword x is part of language y</td>
</tr>
<tr>
<td>KOS_{part_of}</td>
<td>Keyword</td>
<td>OS</td>
<td>keyword x is part of operating system y</td>
</tr>
<tr>
<td>KCots_{part_of}</td>
<td>keyword</td>
<td>COTS</td>
<td>keyword x is part of COTS y</td>
</tr>
<tr>
<td>CC_{part_of}</td>
<td>Concern</td>
<td>Concern</td>
<td>concern x is part of concern y</td>
</tr>
<tr>
<td>KConcern_{handle}</td>
<td>Keyword</td>
<td>Concern</td>
<td>keyword x handles concern y</td>
</tr>
<tr>
<td>KType_{is_of}</td>
<td>Keyword</td>
<td>Type</td>
<td>keyword x is of type y</td>
</tr>
</tbody>
</table>

The relation $KCots_{part_of}$ denotes the assignment of a keyword to a certain COTS, such as the keyword $\text{org.apache.log4j}$ which is part of the apache COTS vocabulary for logging. The $KConcern_{handle}$ relation models the relation between a keyword and the concern it addresses, where concern is a name. For example, $\text{msQLib.h}$ addresses concerns related to communication using message queues. Further, concerns can be part of other concerns, which is denoted by the $CC_{part_of}$ relation. For example, if we consider persistence using a database as a concern, then initialization of database connections and preparing and running queries can be considered two sub-concerns that are part of the database concern. Finally, the $KType_{is_of}$ relation is used to assign a type for each keyword. For example, $\text{msQLib.h}$ is a file type, whereas $\text{msgQSend}$ is a function type. Additional sets and relations can be defined as needed because the structure is flexible and can easily be extended.

Table 6-3: Some sets of the knowledge base

<table>
<thead>
<tr>
<th>Set</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>List of programming languages</td>
</tr>
<tr>
<td>OS</td>
<td>List of operating systems</td>
</tr>
<tr>
<td>COTS</td>
<td>List of COTS</td>
</tr>
<tr>
<td>Concern</td>
<td>List of concerns</td>
</tr>
<tr>
<td>Keyword</td>
<td>List of keywords of external entities</td>
</tr>
<tr>
<td>Type</td>
<td>List of types of external entities</td>
</tr>
</tbody>
</table>

Summary Generation using the Knowledge Base: For an analyst who is not familiar with the system under analysis, the summary generator program offers help by generating a brief summary of key technologies and function areas used by the system under analysis. This is done by first matching tokens present in the files of the system under analysis with keywords in the knowledge base. For each matched token, all related keywords and descriptions are retrieved and used as a summary. It detects the supported programming languages (the KPL_{part_of} relation), the OS types (the KOS_{part_of} relation), the COTS keywords (the KCots_{part_of} relation) as well as the Concerns (the CC_{part_of} relation) of the system under analysis. The summary generator prints a list of matching keywords and descriptions.
6.4.3 Creation of the Module Dependency View

The module dependency view is created by detecting code elements of interest and their interrelationships in the source code [69] and [45]. The analyst needs to analyze the extracted dependencies on different levels of abstraction, including directory level dependencies and function level dependencies, for example call graphs. In order to lift the extracted code relations to different levels of abstraction, we use the RPA’s transitive closure operator, see Figure 6-1. In Appendix B, these RPA operators are defined.

\[ A U B \] – Union of two relations or sets
\[ A \cap B \] – Intersection of two relations or sets
\[ A \circ B \] – Composition of relations A and B
\[ A^{-1} \] – Converse of the relation A
\[ A^* \] – Transitive closure of the relation A
\[ A_{\text{dom}} S \] – Restrict the domain of the relation A to the set S
\[ A_{\text{ran}} S \] – Restrict the range of the relation A to the set S
\[ A_{\text{car}} S \] – Restrict the carrier of the relation A to the set S
\[ \Gamma(A) \] – Set of top or root elements of relation the A
\[ \gamma(A) \] – Set of bottom or leaf elements of relation the A
\[ A \cdot x \] – The left image of the element x
\[ x \cdot A \] – The right image of the element x

Figure 6-1: The RPA operators used in this chapter.

The transitive closure operator performs a reachability analysis on the extracted source code relations (see Table 6-5) and outputs a new relation that contains tuples describing all range values that can be reached from every domain value. This new relation provides the basis for creating abstract module dependency views on any level. In addition to the file and folder structures, we make use of structures provided by the programming language. For example, if the system under analysis is implemented in Java then directories are equivalent to packages and files are equivalent to classes. Similarly to directories, Java packages can contain other Java packages as well as Java classes, and each Java class is typically stored in one file. For systems implemented in Java, we extract module dependency views based on packages and classes as follows.

First, we parse the source code and extract a basic suite of sets and relations as defined in Table 6-4 and Table 6-5. Other sets and relations, such as relations describing dependencies between code elements and database tables, shared variables, and so forth might also be extracted depending on the analysis task.
Second, we run an RPA query that uses the hierarchy relationship between packages and sub-packages as well as dependencies among classes to deduct package dependencies on various levels, depending on the need. The top level packages and their dependencies is denoted by the $\text{MM}_{\text{use}}$ relation, which captures Module-to-Module dependencies, see Figure 6-2, and by modifying that relation, any package level dependencies can be determined.

$$\text{MM}_{\text{use}} = \left( \left( \left( \text{EE}_1 \circ \text{CC}_{\text{use}} \right) \circ \text{EE}' \right) \circ \text{car} \right)$$

Figure 6-2: RPA query for abstracting dependencies between packages.

**Extraction of Similar Modules:** In Table 6-4, we defined the Token set that contains all words present in the source code including comments as well as related sources such as configuration files and build scripts. In Table 6-5, we define the Index relation that contains tuples describing relationships between Token and File sets, where each tuple is a token and the name of the file where the token is present. We use the Index relation to compute the Similarity relation between each pairs of files. Our definition of similarity is based on the tokens present in each file. If two files share at
least 80% of common tokens, we consider the files as similar to each other. The Similarity relation can be lifted to different levels of abstraction depending on the need. For example, the RPA query shown in Figure 6-3 is used for abstracting the file level similarity relation to the top-most package level.

\[
\left(\left(\text{car} \ \left(\text{Similarity} \ \circ \ EE^*\text{part_of}\right)\right)^\circ \right)
\]

Figure 6-3: Query for lifting the similarity relation to the package level.

### 6.4.4 Creation of the Runtime View

Once the sets and relations allowing for creations of any module dependency view are established, the next step is to determine how modules are related to components and connectors at runtime. In general, the module dependency view may not reflect the runtime structure because there could be a many-to-many mapping between modules and components.

#### 6.4.4.1 Identifying runtime components

Beyond considering each executable a runtime component, our strategy to identify subcomponents is based on whether the system under analysis is of type SingExe or of type MultExe. If the system under analysis is a single executable that creates one or more tasks, processes, or threads, then we refer to it as type SingExe. If the system under analysis consists of several executables that run on one or more machines, then we refer to it as type MultExe. The component type is determined by analyzing the batch scripts that launch the system because they indicate whether one or several executables are used. For example, in the NASA SNAS case, each batch script contains the names of the executable jar file. For each executable, we first identify entry points to each task by locating all instances of the task creation function, since it contains the name of the task and the entry function as arguments, which we then extract. We have found that lexical pattern matching using regular expressions of such function signatures can locate the names of the tasks and the entry point functions for each task. In several of the systems we analyzed, names of tasks and entry points are either hardcoded or defined as global constants that are passed as arguments to functions. Thus, we are able extract those data items without performing rigorous data flow analysis, although we note that data flow analysis has the capability to make the data extraction phase more efficient, in particular if there are more than two levels of data indirection.

#### 6.4.4.2 Partition of modules into runtime components

We developed executable RPA queries to automatically partition modules into components. These queries identify a) parts of the source code that are shared among different components, b) parts of the source code that are unique to each component, and c) the source code needed to execute each component.
Table 6-6: Sample relations for identifying runtime components

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP\texttt{create}</td>
<td>Script</td>
<td>Process</td>
<td>script s creates process p</td>
</tr>
<tr>
<td>PC\texttt{enter}</td>
<td>Process</td>
<td>Class</td>
<td>process p starts execution from class c</td>
</tr>
</tbody>
</table>

Table 6-7: Sets for identifying runtime components

<table>
<thead>
<tr>
<th>Set</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>List of batch or shell scripts</td>
</tr>
<tr>
<td>Process</td>
<td>List of processes</td>
</tr>
</tbody>
</table>

In order to discover all classes and packages that are needed for execution of each process, we run the RPA queries shown in Figure 6-4 and Figure 6-5. The PC\texttt{use} and PM\texttt{use} relations denoted relations between processes and classes used by the processes, and processes and packages used by the processes, respectively.

\[
PC\texttt{use} = \left( PC\texttt{enter} \cup CC\texttt{use} \right)^* \bigcap_{dom} (Process)
\]

Figure 6-4: RPA query to identify all classes required for each process.

\[
PM\texttt{use} = \left( PC\texttt{use} \cup EE^*\texttt{part_of} \right) \bigcap_{ran} \left( \downarrow (EE\texttt{part_of}) \right)
\]

Figure 6-5: RPA query to identify packages required for each process.

By querying PC\texttt{use} and PM\texttt{use} relations we automatically identify classes and packages that are a) unique to each process and b) shared among multiple processes. Using the data defined in Table 6-8 and Table 6-9, we partition modules into tasks by running the three queries shown in Figure 6-6.

Table 6-8: Extracted sets from the source code

<table>
<thead>
<tr>
<th>Set</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>List of task names</td>
</tr>
<tr>
<td>Function</td>
<td>List of function names</td>
</tr>
<tr>
<td>Variable</td>
<td>List of global variables</td>
</tr>
</tbody>
</table>
6.4 The ADAM

Table 6-9: Extracted relations from the source code

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF_{enter}</td>
<td>Task</td>
<td>Function</td>
<td>function f is the entry point for task t</td>
</tr>
<tr>
<td>FG_{use}</td>
<td>Function</td>
<td>Variable</td>
<td>function f refers to global variable v</td>
</tr>
</tbody>
</table>

In Figure 6-6 the first query outputs the usage relation between tasks and functions (denoted by TF_{use}), the second query outputs the usage relation between tasks and global variables (denoted by TG_{use}), and the third query outputs the usage relation between tasks and modules (denoted by TM_{use}) by using the outputs of the first two relations as well as the containment relation between code elements.

\[
\begin{align*}
TF_{use} &= (TF_{enter} \cup \text{Call})^* \quad \text{Task} \quad (1) \\
TG_{use} &= TF_{use} \circ FG_{use} \quad (2) \\
TM_{use} &= (TF_{use} \cup TG_{use}) \circ EE_{partof} \quad (3)
\end{align*}
\]

Figure 6-6: Queries for partitioning modules into components.

6.4.4.3 Identifying connectors among runtime components

Once the modules are partitioned into components, the next step is to identify connectors that connect the components. For each component that was based on an executable, connectors “inside” the component can be message queues, pipes, data files, shared memory, and so forth. Components that write to and read from the same connector are connected. We identify connectors and connected components using the relations of the knowledge base as follows. We use message queues as an example, but the same approach applies to all types of connectors. Once we have detected that the system under analysis uses keywords related to message queues, then we extract data such as the names of queues and names of functions that read to or write from message queues. We use regular expressions-based lexical pattern matching to extract the names of queues that are passed as arguments to external functions responsible for read to or write from message queues, see Table 6-10.

Table 6-10: Extracted relations between function and queues

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FQ_{create}</td>
<td>Function</td>
<td>Queue</td>
<td>function f creates queue q</td>
</tr>
<tr>
<td>FQ_{read}</td>
<td>Function</td>
<td>Queue</td>
<td>function f reads from queue q</td>
</tr>
<tr>
<td>FQ_{write}</td>
<td>Function</td>
<td>Queue</td>
<td>function f writes to queue q</td>
</tr>
</tbody>
</table>

We lift the extracted relation between functions and queues to components (i.e., tasks in this case) and queues, as shown in Figure 6-7. The first query lists each task and the
queues it create (denoted by \( T_{Q_{\text{create}}} \)). The second query lists each task and the queues it reads from (denoted by \( T_{Q_{\text{read}}} \)). The third query lists each task and the queues it writes to (denoted by \( T_{Q_{\text{write}}} \)). Components that write to and read from the same queue are connected. Similarly, components that communicate using shared memory (e.g., global variables), files, and pipes are connected, see Chapter 7 which demonstrates how these formulas were used to reverse architect medical device software.

\[
T_{Q_{\text{create}}} = TF_{\text{use}} \circ F_{Q_{\text{create}}}
\]

(1)

\[
T_{Q_{\text{read}}} = TF_{\text{use}} \circ F_{Q_{\text{read}}}
\]

(2)

\[
T_{Q_{\text{write}}} = TF_{\text{use}} \circ F_{Q_{\text{write}}}
\]

(3)

Figure 6-7: Abstracted relations from components to queues.

If the system under analysis is of type \texttt{MultExe}, then connectors between components based on executables can be sockets, middleware, software buses, or similar constructs. Figure 6-8 shows the RPA queries to identify components (i.e., processes in this case) that play the server side socket role. The first query identifies all external keywords of the type “class” that are related to the “server socket” concern stored in knowledge base relations. The second and third queries identify all classes and components of the system under analysis that use classes related to the “server socket” concern, respectively. Analogous queries were developed that identify all classes and components of the system under analysis that use classes related to the “client socket” concern, respectively.

\[
KSSocket_{\text{handle}} = (K_{\text{Type is_of . "class"}}) \cap (K_{\text{Concern* handle . "server socket"}})
\]

(1)

\[
CSSocket_{\text{use}} = CC_{\text{use KSSocket_{\text{handle}}}}
\]

(2)

\[
PSSocket_{\text{use}} = PC_{\text{use o CSSocket_{\text{use}}}}
\]

(3)

Figure 6-8: RPA queries to identify server-side runtime components.

In order to connect server-side and client-side components, we need to identify the ports they use for communication. Ports are implemented in different ways for different purposes and in different programming languages. For example, socket connections use port numbers. One possible source to identify port numbers is the configuration files that contain IP address as well as port numbers for each process. Another possibility to identify port numbers is to use the source code by locating all calls to the external keywords related to “bind”, “listen”, or the constructors of socket classes. From the parameters of these method signatures, we extract port numbers to connect the servers and clients to form the runtime view.
6.4.5 Deriving Concern Specific Views

Once the basic module dependency and runtime views are established, the next step is to create other views that address concerns such as how the software under study handles OS variants, synchronization of tasks, how it handles persistence for interactions with database, and other views depending on the analysis task at hand. If the system under analysis uses external keywords related to different OSes, we can analyze how the OS variants are abstracted. For example, if the system under analysis uses the `CreateThread` C function for Windows and the corresponding C function `pthread_create` for UNIX, then these dependencies indicate that the implementation manages several OS variants. By querying the call relation for dependencies to OS functions, we can determine whether an Operating System Abstraction Layer (OSAL) is present or not. An OSAL offers an abstract interface and alternative implementations for different types of OS.

In a multitasking system, synchronization of tasks is an important architectural concern for analysis. Semaphores are commonly used for synchronization of tasks. We extract tasks synchronization view by first collecting the code relations defined in Table 6-11, and then lift these relations to the task level by using the queries shown in Figure 6-9, in that the first query extracts which tasks create which semaphores (denoted by \( TS_{create} \)), the second query extracts which tasks take which semaphores (denoted by \( TS_{take} \)), the third query extracts which tasks give which semaphores (denoted by \( TS_{give} \)), and finally the fourth query extracts which tasks delete which semaphores (denoted by \( TS_{delete} \)). In all these four queries, the \( TF_{use} \) relation defined in Figure 6-6 is used.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FScreate</td>
<td>Function</td>
<td>SemId</td>
<td>function f creates semaphore s</td>
</tr>
<tr>
<td>FStake</td>
<td>Function</td>
<td>SemId</td>
<td>function f takes semaphore s</td>
</tr>
<tr>
<td>Fsgive</td>
<td>Function</td>
<td>SemId</td>
<td>function f gives semaphore s</td>
</tr>
<tr>
<td>FSdelete</td>
<td>Function</td>
<td>SemId</td>
<td>function f deletes semaphore s</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
TS_{create} &= TF_{use} \circ FS_{create} \\
TS_{take} &= TF_{use} \circ FS_{take} \\
TS_{give} &= TF_{use} \circ FS_{give} \\
TS_{delete} &= TF_{use} \circ FS_{delete}
\end{align*}
\]

Figure 6-9: RPA queries to extract the tasks synchronization view.

If the system under analysis uses external keywords related to the database concern, then we analyze how the system abstracts the database. By querying the dependencies to classes dealing with the database concern, we detect the presence of the database
abstraction layer. In addition, we also detect which processes depend on a database by running the three queries shown in Figure 6-10, in that the first query uses the knowledge base to locate all external classes dealing with the database concern, and the second query restricts the class dependency relation $CC_{use}$ to database classes, and the third query restricts the process to class dependency relation $PC_{use}$ to database classes.

\[
K_{Database\_handle} = (K_{Type\_is\_of} \cdot \text{“class”}) \cap (K_{Concern\_handle} \cdot \text{“database”})
\]

\[
C_{Database\_use} = CC_{use} \rightharpoonup K_{Database\_handle}
\]

\[
P_{Database\_use} = PC_{use} \circ C_{Database\_use}
\]

Figure 6-10: RPA queries to extract processes to a database usage.

Similarly, we produce concern specific architectural views by using the relations of the knowledge base and by constructing RPA queries that manipulates base views using algebraic operators.

### 6.4.6 Evaluation of Quality Properties

The central idea is to evaluate quality using the discovered architecture by focusing on one concern at a time. One quality property that we typically evaluate for is testability. There are many different ways to evaluate testability so the task needs to divided into smaller tasks. For example, testability can be evaluated with respect to the persistence concern, which can be formulated as questions such as: Can the system’s core logic be tested without the database being up and running? The question can be answered by using one or more views to show that there are dependencies to the database that make it impossible to test the system without the database. These views can be used to explain to the members of the project team why the system is not possible to run and test without the database.

Similarly, the analyst can evaluate testability with respect to other concerns, for example the GUI. For example, to answer the question: Can the system’s core logic be tested without the GUI? Runtime views are also needed to reason about testability. For example, in a recent analysis of medical device software, which is a single process with multiple tasks, we highlighted the fact that entry point functions to several components (i.e., a task in this case) are defined in one module (i.e., a file in this case), see Chapter 7. As a consequence of this decision, whenever the module is changed, and thus recompiled, all components have to be retested even though some component specific code may not have been changed. This is because the compiled binary code of the module has code of other components too. Also, in order to demonstrate that each component has 100% code coverage, we would have to wait until all components are tested because coverage tools typically work at the file level and not at the task level.

Another quality property that we typically evaluate for is performance, which is also evaluated by focusing on one concern at a time. For example, the analyst can focus on the persistence concern and identify threats to the performance of the database due to
the style the implemented architecture uses for database connections. Similarly, the analyst can focus on the GUI concern and evaluate how event listeners and dispatchers might slow down the GUI’s response time. In the ADAM, performance evaluation is conducted at an architectural-level, meaning that the analyst focuses on high-level principles that influence the whole system. For example, the threading model used to read data from a socket and to dispatch data to data processors can be considered architecturally significant because if the thread reads and dispatches data in a synchronous manner then there is a risk that one low performing data processor affects the performance of the entire system.

A third evaluation perspective is uniformity or common look-and-feel. Common look-and-feel is an aesthetic property that helps programmers and new-comers understand different parts of the system. Evaluations of common look-and-feel compare how different parts of the system implement the same concern and whether component and system interfaces are standardized or not. For example, the analyst can focus on the persistence concern and evaluate whether or not all modules interacts with the database in the same way.

The purpose of code duplication or clone analysis is to understand how the architecture abstracts commonality and manages variability. To achieve this, the analyst interprets the collected similarity relation within a context of a concern using the discovered architecture.

The purpose of evaluating compliance to architectural rules is to determine whether or not the actual (i.e., the implementation) architecture is consistent with the specified architecture. For example, if the existing documentation specifies that the interactions with a hardware port only should using certain software interfaces of the hardware abstraction layer, the goal of the verification is to check whether there are deviations from this specification. We observed that by analyzing one concern at a time, verification of architectural rules becomes focused and detected deviations are clearly explainable to the development team, see Chapter 2. Figure 6-11 presents the conceptual dimensions of the ADAM.
6.4.7 Tools used in the ADAM

In order to efficiently perform architectural analysis of commercial systems using the knowledge base, analysts require several tools including the tools listed in Table 6-12.

Table 6-12: Tools used in the ADAM

<table>
<thead>
<tr>
<th>Tool</th>
<th>Purposes</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Understand tool</td>
<td>The ADAM uses this commercial tool to extract code-level dependency models from the source code (see <a href="http://www.scitools.com/">http://www.scitools.com/</a>)</td>
</tr>
<tr>
<td>The RPA tool</td>
<td>The ADAM uses this research tool to query the dependency models using relational algebraic operators [169].</td>
</tr>
<tr>
<td>The SAVE-LIGHT tool</td>
<td>The ADAM uses this research tool to import and visualize the dependency models provided by e.g., the Understand tool and the RPA tool.</td>
</tr>
<tr>
<td>The Prefuse tool</td>
<td>The ADAM uses this research tool to visualize the content of the knowledge base as directed and undirected graphs, tree views, and tree</td>
</tr>
</tbody>
</table>
6.5 The ADAM in Action

In this section, the applicability of the method is demonstrated using examples from NASA’s Space Network Access System (SNAS) system, developed by Honeywell. The SNAS is a customer interface to the Tracking and Data Relay Satellite System (TDRSS) and is used for planning, scheduling as well as real-time service monitoring and control of the Space Network (SN). The SNAS is implemented in Java and in SQL. The SNAS has 450KLOC Java code and 30KLOC SQL code, excluding test code, comments, and blank lines.

6.5.1 Why the NASA SNAS was chosen for the study?

We have applied the ADAM onto several commercial systems, implemented in various languages including Ada, FORTRAN, C/C++, and Java. In many cases, there were limited or no possibilities to evaluate the findings of the method because the developers (often contractors) who built the system were not accessible. In the SNAS case, fortunately we have access to several members of the team who are familiar with different parts of the system.

First, we performed an independent analysis of the source code of the SNAS. Once the analysis was completed, the analysis results were presented and feedback was collected from the SNAS team. Thus, there were no influences whatsoever from the SNAS team on the results described here, unless explicitly stated.

Structure of this section: First, we describe the development view that was exacted from the source code of the SNAS. Second, we describe the discovery and analysis of the SNAS runtime view. Third, we describe the discovery and analysis of the SNAS database interaction architecture. Fourth, we briefly summarize the SNAS GUI concern and performance risks detected statically using external dependencies. Fifth, we briefly summarize the SNAS OS interaction architecture.

---

10 We confirmed with the SNAS team that there is no generated source code in it. In addition, we also determined whether the code was generated by analyzing common words often present in comments of generated code [172], such as “generated”, “automatically generated”, “do not edit”, and so forth.
6.5.2 The Module View of the SNAS

The source code dependencies between modules, i.e., the top folders on the disk, of the SNAS are shown in Figure 6-12, where boxes represent folders and arrows are code relations (e.g., import, inherit, and call). The common and framework folders are used by all folders. The dependency from sve to sam is due to dead code. The dependency from dsdm to sdif is due to the sharing of a string utility class.

While this module view offers a useful overview of how the source code is organized on the disk, and which folder uses other folders, there are some limitations if one analyzes architectures only from this dependency diagram alone. For example, we also need to obtain answers to the following questions: a) Do the modules communicate at runtime using intermediate connectors? b) Do the modules run in the same machine or is the system distributed? c) If there is a database, what is the interaction style or pattern used by different modules? These questions are not straightforward to answer by looking at the module view alone.

![Figure 6-12: The module view of the SNAS.](image)

6.5.3 Summary of Build and Batch Scripts of the SNAS

The SNAS has a separate folder for build and batch scripts. The analyst reviewed those scripts and found that the SNAS has seven executable runtime components. These seven components are started by seven batch scripts. Each script runs a java command to start the respective entry point. It is interesting to note that none of the batch scripts start the shareclient, common, and framework modules shown in Figure 6-12. The analysis of build scripts revealed that these three modules contain shared source files that are used by other runtime components. That is, the executable jar file of each runtime component contains a physical copy of these three modules. Thus the analyst concluded that the SNAS consists of the following seven components: sve, sam, mocclient, oamclient, snif, dsdm, and sdif.

6.5.4 Some Facts about the SNAS using its External Dependencies

The analyst starts by generating a high-level summary using a collection of scripts that use the extracted source code relations of the SNAS and the knowledge base as inputs, see Figure 6-13. One of the advantages of this generated summary is that it shows the list of concerns (e.g., GUI, Database, Configuration, Security, and Logging) built inside
6.5 The ADAM in Action

the SNAS. It also shows some potential architectural connectors (e.g., Sockets) implemented in the SNAS.

A Snippet of summary produced using the knowledge of External Dependencies

- GUI because it uses java.awt and javax.swing packages
- Database because it uses the java.sql package
- Secured Socket Layer (SSL) because it uses the COTS crysec.ssl package
- UDP sockets because it uses the java.net.DatagramSocket class
- TCP sockets because it uses the java.net.ServerSocket and java.net.Socket classes
- Non-blocking Socket channels because it uses java.nio.ServerSocketChannel
- Configuration files because it uses java.util.Properties (load, getProperty) methods
- OS interaction because it uses java.lang.Runtime (exec) method
- Logging because it uses the org.apache.log4j package

Figure 6-13: (Snippet) Summary generated using the knowledge base.

6.5.5 Discovering the Inter-Process Communication Structure

The analyst concluded from the high-level summary that the SNAS uses sockets. Therefore, it was reasonable to hypothesize that the modules of the SNAS communicate using sockets as runtime connectors. Thus, the analyst’s natural next step was to identify server-side and client-side sockets.

Discovering Server-side sockets: Which modules create instances of java.net.ServerSocket, java.nio.ServerSocketChannel, and crysec.ssl.SSLServerSocket?

Discovering Client-side sockets: Which modules create instances of java.net.Socket, crysec.ssl.SSLServerSocket DatagramChannel.socket(), and SocketChannel.connect(…)?

Discovering Socket Wrappers: Also check whether there are wrapper classes to external socket libraries, because socket instances could be indirectly created by creating instances of wrappers.

Discovering Ports: Use dependencies to the java.util.Properties class and locate configuration files. Experience tells us that IP addresses and port numbers are often specified in configuration files.

Figure 6-14: Analysis questions for discovering the IPC structure.

6.5.5.1 Discovering Server-side Components using External Entities

To understand how the sockets were used, the analyst queried the extracted code relations of the SNAS in order to identify all components that create instances of the java.net.ServerSocket class. This was done based on the strategies stored in the knowledge base (see Figure 6-14). The analyst queried the extracted code relations of the SNAS in order to identify all components that create instances of the java.net.ServerSocket class. The results showed that the dsdm and sdif components create one instance of the ServerSocket class, see Figure 6-15 (a) and (b). Since the SNAS also uses Java’s non-blocking Input/Output class java.nio.ServerSocketChannel, the analyst also queried the code relations for dependencies on this class. The query detected that the ServerSocketAcceptor
within the framework folder create a socket instance using java.nio.ServerSocketChannel.

After a quick glance at the ServerSocketAcceptor class it became clear that this is a wrapper class for creating server side socket instances. Thus, the analyst queried the code relations in order to find all components that create instances of this wrapper class. The analyst found that snif creates two instances of this server socket wrapper class and that sam creates four instances; see the unfilled circles of Figure 6-15 (c), (d). However, the mocclient has a different strategy to create server socket instances. The analyst found that the mocclient has a base class that uses the wrapper class of the framework to create a server socket instance. In addition, there are six children of the base class that indirectly create their server-side socket ports using calls to the super method of their parent class, see Figure 6-15 (e).

Figure 6-15: Discovered Server-side sockets.

Since there are dependencies from the SNAS to the crysec.ssl.SSLServerSocket, which is a COTS component, the analyst also queried the dependency model to find all components that create instances of this class. It turned out that the sam subsystem is the only subsystem that creates and uses two secured server side instances of the SSLServerSocket, see the filled circles of Figure 6-15 (d).

6.5.5.2 Discovering Client-side Components using External Entities

The analyst repeated the process and discovered the client side socket instances, using dependencies to Java’s client-side socket class java.net.Socket and Crysec’s SSLSocket class.

Figure 6-16: Discovered Client-Side sockets.

6.5.5.3 Connecting Server and Client Side Ports using External Entities

In order to connect client-side and server-side ports, the analyst used Java’s properties file used for configuring each subsystem. The analyst located the right set of property files using dependencies to the java.util.Properties class. These property files contain the IP address of each subsystem together with the actual port values. From this
information, the analyst was able to map the server-side ports to the client-side ports. The names of the files involved in socket communication contain a good prefix (e.g., `sam2sveConnector.java`), offering additional valuable data to connect the ports. The external dependencies to Java’s Datagram socket class, which contains methods for implementing the UDP protocol, showed the analyst that the UDP protocol is used between the `sve` and `snif` components, see Figure 6-17, where filled boxes are back-end systems that interact with the SNAS. Objects are sent between components using the Transfer Object Design Pattern [5].

![Diagram](image.png)

Figure 6-17: Discovered Runtime View.

### 6.5.5.4 Discovery of Transfer Object Design Pattern for Transporting Data over Sockets using External Dependencies

The extracted code relations showed that the files that are involved in socket communication use the `java.io.ObjectStream.writeObject` and `java.io.ObjectStream.readObject` methods. The analyst reviewed those files and found that the components of the SNAS use the `writeObject` and the `readObject` method for sending and receiving serialized objects over the socket, as required by the Transfer Object Design Pattern [5]. The central idea of this pattern is to transfer objects across communication channels, instead of making remote procedure calls to overcome the inherent network performance overhead of RPC [300]. In addition, the extracted code relations had shown the analyst that all those serialized objects that are sent over sockets are located in the `common` folder explaining why all components depend on the `common` folder. Had the analyst excluded the external dependencies to Java’s `writeObject` and `readObject` methods, it would not have been straightforward to discover this design pattern in the implementation.
The analyst noticed that there are 384 files (or classes) inside the common folder. By reviewing a limited number of files the analyst concluded that all of them are equivalent bean classes (i.e., data containers) because a) all the classes of the common directly or indirectly (i.e., using inheritance) implement the Serializable interface, a vital condition for transferring objects on sockets, b) all methods of all common classes are simple setters and getters, i.e., they have the prefix get, set, and toString (in some cases), c) in addition, the collected code metrics showed that almost all methods of the common classes are one-line getters or setters, d) in most of the files there were no logging which matches the fact that in general, bean classes do not typically log their activities, and e) there were no outgoing dependencies from common to other components, except that some the classes of common use utilities of the framework. Based on this evidence, the analyst inferred that all 384 classes of the common folder are bean classes used to transfer data between components. This capability to analyze a small sample and generalize to findings to a collection of classes and summarize their role in one sentence is crucial in architecture discovery because otherwise it would be necessary to analyze all files.

For this chapter, it was not possible to analyze the back-end systems (colored boxes in Figure 6-17) without discussing with the SNAS team because the analyst did not have access to the source code. The SNAS team told the analyst after the analysis was completed that the reason for having 6 client-side socket ports at the snif subsystem is due to its counterpart back-end: the NCCDS system, which was developed many years before the SNAS was developed. Similarly, a new requirement drove them to introduce 6 server-side socket ports at the mocclient subsystem, in order to allow the EPS system to communicate with the NCCDS system.

Finally, the SSL is used for the connection between mocclient, oamclient and the sam because both clients are deployed in an open network and the connection must be secure. These are the kinds of design rationale we will not be able to discover from the source code alone, and definitely need to talk to the people (if available).

6.5.5.5 Analyzing the Discovered Runtime View

In this section, we summarize the analysis of the discovered runtime view using the following questions: 1) What are the performance influencing architectural decisions from the communication perspective? 2) Is there a common look-and-feel from the communication perspective? 3) Are the files involved in interactions with communication channels cloned from each other? and 4) Can the components of the system be tested independently?

6.5.5.5.1 Performance and Communication

The analyst concluded that the distributed components of the SNAS communicate using the Transfer Object Design pattern by sending and receiving serialized objects over the sockets. SUN’s book mentions that Remote Procedure Calls (RPC) using the Java’s Remote Method Invocation (RMI) can be slow due to communication overhead [5], despite the fact that RMI is simple and fairly easy to understand and program. The authors of [5] also note that by using the Transfer Object Design pattern, performance can be improved. However, according to the SNAS team the introduction of Transfer
Object Design pattern did not solve all performance problems, because if objects are sent over sockets, then there is an issue of managing the waiting time in the sockets before the receiver picks-up the objects for processing. To avoid potential delays and degraded performance, the SNAS team introduced additional ports to different components and transferred different types of objects through different ports. For example, the four socket connections between the `sam` and `sve` are used for exchanging four different types of objects, see Figure 6-17. By doing so, the SNAS team attempted to reduce the waiting time of objects on sockets.

Traditionally, socket programming uses one thread per client connection. However, frequently creating and destroying threads due to short-lived sessions would incur performance overhead. Also, valuable CPU time can be wasted just because of context switching due to threads. In order to overcome these performance issues, Java 1.4 introduced a new concept called non-blocking socket communication channels for client-server communication. The analyst found that the files that are involved in socket communication use the `java.nio.channels.SelectionKey` and the `java.nio.channels.Selector` classes. These two classes are the core for implementing the reactor design pattern in Java (see [252] and [212] for details). In this pattern, the event demultiplexer waits for events that indicate when a socket is ready for a read or write operation. The demultiplexer passes this event to the appropriate handler, which is responsible for performing the actual read or write. Based on this collected evidence, the analyst hypothesized that the SNAS inter-component communication is inspired by performance goals.

### 6.5.5.5.2 Performance and Threading Models

The analyst discovered another similar performance problem by using dependencies to Java socket classes. The discovery started by observing that the `DataManager` class of the `mocclient` is responsible for reading responses from the socket connected to the server, and delegating these messages to appropriate processing classes. The analyst tried to understand the threading strategy used for reading the data from the socket and dispatching it to the appropriate processors. To achieve this, the analyst reviewed the `run` method within the `DataManager` that reads data from the socket. The analyst revealed that the method uses the same thread for reading data from the socket as well as for dispatching data to the data processors, in a synchronous mode. The analyst then concluded that due to this synchronous threading model, slow performing methods of data processor classes may hurt the entire system since data cannot be read from the socket until control returns from the data processors to the `DataManager`. We discussed this potential issue with the SNAS team, and they acknowledged this problem and even mentioned that the socket timeout happens before processing all data in the socket due to synchronous method calls. We are discussing the possibility of either introducing a thread pool design pattern using the `java.util.Executor` class to resolve this performance problem, or to introduce additional queues so that the data from the socket can be just transferred to different queues, and thus socket timeout can be avoided. The SNAS team is evaluating these solutions for the next release.
6.5.5.3 Common look-and-feel and Communication

The analyst detected some common look-and-feel issues due to the by-pass of the socket wrapper defined in the shared framework folder. In particular, both the dsdm and sdif components create instances of server sockets by directly using the java.net.ServerSocket class. Similarly, the mocclient, oamclient, and sdif subsystems create instances of client sockets by directly using the java.net.Socket class instead of using the wrapper. Thus, the look-and-feel from a communication point of view is different among the components. The reasons for differences in look-and-feel will be discussed together with other architectural violations at the end of this chapter.

6.5.5.4 Code Cloning and Communication

The analyst used the similarity tool to compare all files of the SNAS and produced a similarity table that contains pairs of potential file clones. Since the list of files involved in socket communication was located during the discovery of the SNAS runtime view, the similarity table was filtered with respect to those files only. This filtering with respect to a concern reduced the number of clone pairs to review. The analyst concluded that all six files (see Table 6-13) which accept client connections and act as server-side socket ports of the mocclient are very similar to each other.

Table 6-13: Text similarity of the server-side of mocclient

<table>
<thead>
<tr>
<th>From File</th>
<th>To File</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NccPmDataAcceptor.java</td>
<td>NccReconfigAcceptor.java</td>
<td>0.974</td>
</tr>
<tr>
<td>NccSchReqAcceptor.java</td>
<td>NccTswStoreAcceptor.java</td>
<td>0.964</td>
</tr>
<tr>
<td>NccPmDataAcceptor.java</td>
<td>NccSchStatusAcceptor.java</td>
<td>0.962</td>
</tr>
<tr>
<td>NccReconfigAcceptor.java</td>
<td>NccSchStatusAcceptor.java</td>
<td>0.962</td>
</tr>
<tr>
<td>NccAcqStoreAcceptor.java</td>
<td>NccTswStoreAcceptor.java</td>
<td>0.961</td>
</tr>
<tr>
<td>NccAcqStoreAcceptor.java</td>
<td>NccSchReqAcceptor.java</td>
<td>0.958</td>
</tr>
<tr>
<td>NccSchReqAcceptor.java</td>
<td>NccSchStatusAcceptor.java</td>
<td>0.895</td>
</tr>
<tr>
<td>NccSchStatusAcceptor.java</td>
<td>NccTswStoreAcceptor.java</td>
<td>0.893</td>
</tr>
<tr>
<td>NccReconfigAcceptor.java</td>
<td>NccSchReqAcceptor.java</td>
<td>0.886</td>
</tr>
<tr>
<td>NccPmDataAcceptor.java</td>
<td>NccTswStoreAcceptor.java</td>
<td>0.884</td>
</tr>
<tr>
<td>NccReconfigAcceptor.java</td>
<td>NccTswStoreAcceptor.java</td>
<td>0.883</td>
</tr>
<tr>
<td>NccAcqStoreAcceptor.java</td>
<td>NccSchStatusAcceptor.java</td>
<td>0.881</td>
</tr>
<tr>
<td>NccPmDataAcceptor.java</td>
<td>NccSchReqAcceptor.java</td>
<td>0.881</td>
</tr>
<tr>
<td>NccAcqStoreAcceptor.java</td>
<td>NccReconfigAcceptor.java</td>
<td>0.877</td>
</tr>
<tr>
<td>NccAcqStoreAcceptor.java</td>
<td>NccPmDataAcceptor.java</td>
<td>0.873</td>
</tr>
</tbody>
</table>

It is interesting to note that even though there is a base class for each of those six classes in mocclient, there is a lot of code duplication between these six files. In addition, the six files that act as client-side socket ports of the snif are very similar despite the fact that they have the same base class. This means that shared behavior is not properly abstracted yet, see Table 6-14. The two server side socket files of the sam subsystem are also cloned. Similarly, the files that are involved in the client-side socket communication of the oamclient and mocclient are very similar. There are many methods in these two files which are exact copies of each other and could be moved to the shareclient, which is a shared infrastructure for both client types.
Table 6-14: Text similarity of the client-side of snif

<table>
<thead>
<tr>
<th>From File</th>
<th>To File</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccPmDataConnector.java</td>
<td>0.968</td>
</tr>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccReconfigConnector.java</td>
<td>0.968</td>
</tr>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccSchReqConnector.java</td>
<td>0.968</td>
</tr>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccSchStatusConnector.java</td>
<td>0.961</td>
</tr>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccSocketConnector.java</td>
<td>0.961</td>
</tr>
<tr>
<td>NccAcqStoreConnector.java</td>
<td>NccTswStoreConnector.java</td>
<td>0.961</td>
</tr>
<tr>
<td>NccPmDataConnector.java</td>
<td>NccReconfigConnector.java</td>
<td>0.795</td>
</tr>
<tr>
<td>NccPmDataConnector.java</td>
<td>NccSchReqConnector.java</td>
<td>0.782</td>
</tr>
<tr>
<td>NccPmDataConnector.java</td>
<td>NccSchStatusConnector.java</td>
<td>0.776</td>
</tr>
<tr>
<td>NccPmDataConnector.java</td>
<td>NccSocketConnector.java</td>
<td>0.766</td>
</tr>
<tr>
<td>NccPmDataConnector.java</td>
<td>NccTswStoreConnector.java</td>
<td>0.749</td>
</tr>
</tbody>
</table>

6.5.5.5 Testing and Communication

Because of the distributed client-server architectural style, clients can be tested with fake servers and vice-versa without changing any source code. However, some changes (e.g., IP address and ports) are needed in the configuration file. In fact, the SNAS team has also developed simulators for back-end systems so that the SNAS can be tested without the real back-end systems being up and running. These simulators can send data over the socket to components of the SNAS. Classes involved in the socket communication will read the incoming object types as in the real scenario. More details on testability issues due to databases and GUIs are discussed below.

6.5.6 Discovery of the Database Interaction Architecture

In this section, we discuss the architecture analysis of the SNAS from the view point of interaction with databases from Java code. We discuss how the analyst detected the presence of database abstraction layers, testability, error handling issues, and performance influencing architectural decisions. How to architect database interactions so that unit testing can be performed without the database being up-and-run based on the SNAS architecture is provided for developers to learn and reuse in their projects.

Our approach for analysis of persistence concerns is based on the following observations of several commercial systems: Many systems implement their need for persistence by using a RDBMS that is based on the SQL language, which is typically external to the software under analysis. Thus, the software needs to connect to and disconnect from the database, communicate with and transfer data to and from the database, as well as manage errors during interaction with the database. It is also desirable if the software is not directly dependent on the database so that it can be tested without the database and so that the database can be replaced if necessary. Many systems implement a DAO (Data Access Object) layer which contains classes that are responsible for interacting with the database. On the one hand, DAOs collect the results of database queries and convert them into data beans which are basically data containers with getters and setters. On the other hand, if we want to store the data of a bean in a database table, the bean object is passed as an argument to the methods of the responsible DAO [5].
Based on this model, we derive the following questions: 1) Is there a DAO (Data Access Object) layer that abstracts the physical database? 2) What is the general strategy for managing database connections? 3) Can the software be tested without the database being up and running? 4) Are database errors abstracted and propagated upwards in such a way that higher-level layers are not aware of databases? 5) Is there a common look-and-feel in the way database tables are accessed by different components? and 6) What are the different DBMSs the software supports?

In order to answer these questions, the analyst focused the initial analysis on dependencies to database tables. The analyst used a parser that identifies files that use database tables based on regular expressions involving key SQL statements, for example, “select”, “insert”, “update”, and “delete”. The extracted dependency from the SNAS source code files to database tables is shown in Figure 6-18, where arrows denote SQL queries from Java files of the db folders.

![Figure 6-18: Dependencies on database tables.](image)

Once the analyst had determined that such dependencies existed, the conclusion was that the system must be using a database in a direct way, instead of using indirect database dependencies that can be created using javax.persistence. Such indirect database dependencies can make use of a database without using any of the SQL keywords listed above. The analyst then made the observation that there is a good common look-and-feel in the way the files that are using database tables are organized on the disk because there is a db folder per subsystem, each containing the classes that interact with database tables using SQL statements. The analyst’s other observation was that snif, dsdm, sve, and sdif use a database, but oamclient, mocclient, and sam do not. Whether or not these components use the same database will be explored below.

Here, the answers to above questions are presented using the knowledge of external dependencies.

### 6.5.6.1 Discovering the Database Interaction Architecture of the snif

The analyst selected the snif subsystem, which is one of the four components that use a database, and proceeded to analyze the snif subsystem’s database interaction style.
The knowledge base knows that the methods of the `java.sql.PreparedStatement` class can be used to prepare and execute database queries in Java. Using that knowledge, the analyst queried the extracted code relations and found that all classes of the `snif` that prepares database queries are organized in one folder/package: `snif.db`, see Figure 6-19. In addition, the analyst observed that all Java files that use SQL statements such as `select`, `insert`, `update`, and `delete` are only present in the `snif.db` folder, thus confirming that all direct database interactions are limited to the `db` folder.

![Diagram of Data Access Objects (DAOs) of `snif`.](image)

The analyst then proceeded to analyze how the execution of SQL queries is managed. The analyst knew a basic fact that in order to execute SQL queries from Java, a `java.sql.Connection` object is needed. The analyst then found, by analyzing the extracted code relations for dependencies to `java.sql.Connection`, that each class in `snif.db` contains a method called `setDbConn` which takes the `Connection` object as a parameter, see Figure 6-20.

The analyst concluded that all classes of the `snif.db` folder can be safely categorized as DAOs because of the following evidence: a) all classes in `snif.db` depend on the `java.sql` package, b) all classes in `snif.db` use classes of `common`, which contains data beans as shown earlier, and these data beans are either used to convert SQL results into objects or to insert data into database tables as explained above, c) there are no outgoing dependencies from `snif.db` to other folders of `snif`, and d) each class in `snif.db` gets a database connection object from outside through the `setDbConn` method. Based on collected evidence and without reviewing all classes in `snif.db`, the analyst inferred that the `snif` subsystem clearly separates database table concepts from other concepts.

![Diagram of DB interactions in the `snif` subsystem.](image)
6.5.6.2 Clones in the DAO layer of the snif Subsystem

The analyst now knows that there is a dedicated DAO layer consisting of the classes in snif.db, which interact with database tables, see Figure 6-19 and Figure 6-20. The analyst also found that the classes of the DAO layer are independent of each other.

The analyst proceeded to run the similarity tool on the files in the DAO layer, which reported occurrences of clones. The analyst analyzed some of the reported clones to gain insights into the underlying reason behind cloning. The analysis showed that the catch and finally blocks in each file of the DAO layer are identical. In the catch block, the error code stored in the SQLException is processed and converted into a SQL independent error code. The catch block contains code that is used to roll back database transactions that did not complete properly. In the finally block, all classes call the close method of the java.sql.PreparedStatement object and commits successful database transactions.

In our opinion, the developers are not to blame for these clones in the DAO layer. Rather, this is an inherent limitation of the Java language and its way of supporting database programming because it leads to the creation of boiler-plate code that is identical across all DAO classes except for only a few parameters that differ. The boiler-plate code the catch block includes, is for example, code to a) manage the database connection, b) create an instance of PreparedStatement, c) handle SQL exceptions, and rollback of transactions, and d) close the PreparedStatement object. It is not straightforward to abstract the catch block into a modular unit. Modern COTS (e.g., Hibernate and javax.persistence) were invented exactly to solve these code redundancy problems in database interactions, making a solid business case with ample evidence to migrate to modern COTS software in the future.

6.5.6.3 Is the snif subsystem testable without a running database?

In order to evaluate testability from the database point of view, the analyst first had to understand how the DAOs (i.e., classes in the snif.db package shown in Figure 6-19) are used within the snif subsystem. More specifically, the analyst had to understand whether or not it is possible to avoid interactions with the database. To this end, the analyst checked whether or not the DAOs that interact with the database are instantiated by other classes of the snif in a hard-wired way. To achieve this, the analyst extracted all incoming dependencies to the DAO layer and found that no other class is using the DAO layer except the SnifDatabaseManager class, which creates instances of all classes of the DAO layer, see Figure 6-21 and Figure 6-22 (a). The analyst also found that the SnifDatabaseManager is instantiated only by the ServiceManager class, which gets all necessary database parameters such as login, password, and url, from the configuration file.
Based on these findings, the analyst concluded that the SnifDatabaseManager class is the gateway for interactions with the DAO layer, see Figure 6-21. By studying the extracted dependency relations, the analyst also noted that the SnifDatabaseManager class uses the DBConnectionPool class in the framework. This discovery is fully explained in the next section. The analyst reviewed the SnifDatabaseManager class and found that it creates a pool of database connections with the capacity of eight connections, see Figure 6-22 (b).

The review also showed that the SnifDatabaseManager distributes the eight database connections among the several classes of the DAO layer, based on how often the different database tables are accessed. This is done by calling the setDbConn method DAOs, see Figure 6-22 (c). The analyst concluded that the snif subsystem’s database interaction architecture is driven by performance goals because it creates several database connections and distributes them among the classes of the DAO layer. This example also shows the power of filtering the extracted code relations using the dependency graph.
concerns; otherwise we cannot easily see the beauty of hidden architectural structure in the source code.

```java
// Create data access objects
private void createDAOs() {
    this.tdrsDAO = new DbTdrs();
    this.sicDAO = new DbSic();
    this.alertDAO = new DbAlert();
    ...
}
```

```java
// Create a connection pool
private void setupDBConnectionPool() {
    this.connPool = new DBConnectionPool(
        poolCapacity, url, login, password);
}
```

```java
// Distributes database connections among DAOs
private void propagateDBConnection() {
    // Use a shared connection for non-frequently used tables
    Connection default_conn = this.connPool.getConnection();
    this.sicDAO.setDbConn(default_conn);
    this.alertDAO.setDbConn(default_conn);
    ...

    // Use a separate connection for frequently used tables
    Connection tdrs_conn = this.connPool.getConnection();
    this.tdrsDAO.setDbConn(tdrs_conn);
    ...
}
```

Figure 6-22: Methods of the snif DB manager class.

During the review of the SnifDatabaseManager class, the analyst also found that its constructor has a Boolean flag called isDbEnabled. If the flag is false, then the public methods of the SnifDatabaseManager call the “dummy” methods of the classes of the DAO layer. The analyst randomly picked one of the dummy methods of one of the DAOs and found that it populates dummy data for testing purposes. The extracted code relations also showed that each class of the DAO layer contains methods with the name “dummy” in it, which confirmed the analyst’s hypothesis that this construct existed to facilitate testing. For example, if the higher-level layers call the getSicList method of the SnifDatabaseManager then either the real `getSicList` defined in `DbSic` or the dummy `getDummySicList` method will be called, see Figure 6-23 and Figure 6-24.

Although the snif subsystem has the capability for testing without running the database, the analyst concluded that there is a mix of testing and production code: the isDbEnabled flag is used as a control to switch between real and dummy DAO methods at runtime, see Figure 6-24. Our recommendation is to separate the testing concern using dependency injection concepts proposed in Spring [267] or Google’s Guice [145] frameworks as follows. The core idea is to let each class within the DAO layer implement an interface of the services it offers. In addition, corresponding to each real DAO class, there is a separate dummy DAO class with the same interface but with an
implementation that populates fake or dummy data. Instead of creating instances of DAO classes in a hard way, as is currently the case, the database manager will create instances of DAO interfaces. These interfaces can be bound either to real DAO instances or to dummy instances for testing. This can be done with the help of configuration concepts used in the Spring or the Guice frameworks. Thus, using this design one could separate testing concerns from the real code, which would increase testability and readability significantly.

```
public SnifDatabaseManager(String dbUrl, String login, String password boolean isDbEnabled);
```

```
public DBResultSet getTdrs();
public DBResultSet getAlert();
public DBResultSet getSicList();
```

```
public DBResultSet getSicList() {
    DBResultSet result = null;
    if(this.isDbEnabled) {
        result = sicDAO.getSicList();
    } else {
        // return dummy data
        result = sicDAO.getDummySicList();
    }
    return result;
}
```

**Figure 6-23:** Design for testing without accessing databases.

**Figure 6-24:** Using dummy data for testing.

### 6.5.6.4 Discovery of the Database Connection Pool Design Pattern using External Dependencies

As mentioned above, the analyst noted that the SnifDatabaseManager class uses the DBConnectionPool class in the framework. The analyst’s next step was to understand the details of how the connection to the database was managed and discovered that the DBConnectionPool class within the framework uses the DriverManager class, see Figure 6-25.
The analyst then reviewed the `createPooledConnection` method because it calls the `DriverManager.getConnection` method, which drew the analyst’s attention because the knowledge base says that the `getConnection` method of the `java.sql.DriverManager` is used to create connections to the database. The review showed that a pool of database connections is created in a for-loop of the `createPooledConnection` method, see Figure 6-26.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnectionPool {
    private static int capacity = 10;
    private static String url;
    private static String user;
    private static String password;
    private static Connection[] connections = new Connection[capacity];

    public void initialize() {
        for (int i = 0; i < capacity; i++) {
            try {
                connections[i] = DriverManager.getConnection(url, user, password);
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public void createPooledConnection() throws Exception {
        try {
            for (int i = 0; i < capacity; i++) {
                Connection conn = DriverManager.getConnection(url, user, password);
                connections[i] = conn;
            }
        } catch (Exception e) {
            throw e;
        }
    }
}
```

The for-loop calls the `java.sql.DriverManager.getConnection` method, which returns a database connection object that will be stored in the connection pool. This discovered pattern is called the Database Connection Pool design pattern: The core idea is that a set of connection objects are created up-front, as demonstrated here, and when...
a component needs to access a database table they can take one already created connection object from the pool, and return it to the pool after using it. Thus saving the time it takes to create and destroy the connection [10]. This discovered view of the SNAS indicates that the database interaction architecture is driven by performance goals because experience reminds us that frequently creating and destroying connections to a database can affect performance of the system. This concrete example highlights the value of the knowledge base of external entities helping us in easily finding the file and the methods that implement the design pattern for database connections.

Next, the analyst proceeded to the analysis of the sdif subsystem and discovered issues in testability and error handling of the database concern.

### 6.5.7 Database Interaction Architecture in the sdif Subsystem

Similar to the analysis of the snif subsystem database interaction architecture, the analyst used the dependencies to the java.sql.PreparedStatement class and discovered that the only class that prepares SQL statements is sdif.db.DbInteractor. The extracted call graph of this class showed that almost all of its methods use methods of PreparedStatement in order to prepare and execute SQL queries. In addition, the analyst found that the only outgoing dependency from the DbInteractor class is to the common folder, which contains data beans as explained earlier. The only exception is dependencies to logging methods. Based on this evidence the analyst concluded that the DbInteractor is the only class of its DAO layer and that it is responsible for interacting with several database tables, in contrast to a collection of several DAO classes in other components.

```java
sdif.db.DbConnectionManager

// Create the singleton connection object
public Connection getConnection() throws SQLException {
    if (this.connection == null ||
        this.connection.isClosed()) {
        try {
            this.connection = DriverManager.getConnection(...);
        } catch (SQLException sqle) {
            throw sqle;
        }
        return this.connection;
    }
}
```

Figure 6-27: DB connection in the sdif subsystem.

The analyst also noticed that there were no dependencies from the DbInteractor class to the database connection creation method getConnection of the java.sql.DriverManager class. This led the analyst to investigate further how the sdif subsystem creates database connections. The extracted dependency relations
showed that the only class that depends on the DriverManager class is the sdif.db.DbConnectionManager class. The analyst reviewed this class and found that it uses the Singleton design pattern [88] and creates only one instance of the database connection, see Figure 6-27 and Figure 6-28, as opposed to dividing the database traffic among several database connections using a Database Connection Pool as was the case for snif. In addition, the analyst queried the extracted code relations and found that the database connection manager class gets all parameters (e.g., database url and login) from a configuration file, see Figure 6-28.

Figure 6-28: Singleton design pattern for database connections.

The extracted call graph of the methods of the DbInteractor class showed that all its public methods use the methods of the DbConnectionManager class in order to obtain an instance of a database connection, see Figure 6-29.
Figure 6-29: Database abstraction in the sdif subsystem.

The analyst reviewed some of the methods of DbInteractor and concluded that they all follow a general pattern: First, in order to obtain an instance of the DbConnectionManager, all methods of the DbInteractor call the static getSingletonJDBCInstance method. Second, using that instance of the database manager, all methods of the DbInteractor call the getConnection method of the database connection manager. Third, all methods of the DbInteractor run SQL queries and return results to their callers. These three steps are summarized in Figure 6-30.

Note that there is an architectural mismatch due to the way the DAOs of the sdif and snif components create a database connection: the DAOs of sdif are responsible for obtaining an instance of the database connection, whereas the DAOs of snif are assigned an instance by the data manager. Thus, these two components have different common look-and-feel with respect to the database connection concern.

Figure 6-30: DB interaction steps in the sdif subsystem.

6.5.7.1 Database Error Abstraction Issues in the sdif Subsystem

The analyst has concluded that the sdif.db.DbInteractor class is the gateway to interact with database tables, see Figure 6-29. The analyst also noted that all the public methods of DbInteractor throw SQLException, see the methods declarations in Figure 6-29. As a consequence, the knowledge of the database concepts had leaked into the higher-level layer because it has to handle SQLExceptions being thrown by the
methods of DbInteractor. Thus, in contrast to the other components, the DAO layer in sdif (i.e., the DbInteractor class) fails to abstract the SQLException into an error object type that is free of database concepts. This example shows that the developers implemented the DAO layer but did not give sufficient attention to abstracting the error raised by the lower-level layer.

6.5.7.2 Testability Issues in the sdif Subsystem

The analyst then proceeded to analyze whether it is possible to test the sdif subsystem without the database. Having known that the sdif.db.DbInteractor is the only class that interacts with the database, the analyst queried the extracted Call relation and found that all instances of the DbInteractor class is created within constructors, e.g., see Figure 6-31 (a), of higher-level layers.

The analyst recalled the fact that constructors cannot be overridden. As a consequence, none of the public methods of those classes that use methods of the DbInteractor are testable unless the database is running. Figure 6-31 (b) shows an example method that cannot be tested without the database because it uses an instance of the DbInteractor in a hard-wired way for calling the getUpd method, which accesses database tables. The analyst found that there is no way to stop the control flow from reaching the physical database and there are 20 classes which unfortunately create instances of the DbInteractor within their constructors similar to the pattern shown in Figure 6-31.

Hence, the sdif subsystem, in contrast to the other components, is not testable without a running database. The code could be refactored to allow for testing without database. However, experience reminds us that managers are generally nervous about investing in refactoring because it does not add value to the product from the end-user’s point of view. However, in our opinion, managers are open to refactorings if the proposed solution will facilitate testing and decrease testing effort, as in this case. Thus, this analysis helped creating a business case for refactoring to improve the testing capability.

In [136], Jacobson says, “To make the design minimally affected by the DBMS, as few parts of our system as possible should know about the DBMS’s interface.” Yes, this analysis has shown that the components of the SNAS satisfy this quote in general. There are, however, a few cases where the database exception knowledge is mixed with business logic as shown above, but the effect of that knowledge leak is limited and has only limited effect on the testing situation.
The other components (sve and dsdm) have a similar database interaction architecture. Thus, we will not discuss them here. The analyst did not review the stored procedures and Entity-Relationship models. Hence, the analyst cannot answer how the SNAS handles variants of DBMS (e.g., Oracle or MySQL). It depends on the needs and it was deemed not necessary at this point. This example shows that the ADAM is flexible because the analyst can decide whether or not to address each question mentioned in the analysis guide, based on the available effort and the needs.

Next, the analyst proceeded to the analysis of GUI architecture using external dependencies of the SNAS.

6.5.8 Analysis of the GUI Architecture using External Entities

Here, we show how the analyst statically detected architecturally significant performance and testability risks due to threading models with the help of external entities. We built a knowledge base for the Swing and AWT packages in order to support the discovery of GUI architectures and analysis of testability and performance risks, see Figure 6-32.

![Figure 6-32: A snippet of the knowledge base for Java GUI libraries.](image)

6.5.8.1 Some Performance Problems in the Event Notification Mechanism

In order to keep the GUI responsive, the threading model and the event dispatchers need to be carefully designed, thus the analyst wanted to analyze how those concerns were handled. The analyst used the fact that the knowledge base knows that the javax.swing.event.EventListenerList class is typically used to store the list of event listeners. Using that knowledge, the analyst discovered the method that calls the event listeners, see Figure 6-33.

```java
// This method is used to fire Notification to all listeners
public static void fireNotificationEventStarted(NotificationEvent evt) {
    Object[] listeners = SnasFrame.getListener.getListenerList();
    for (int i = 0; i < listeners.length; i+=2) {
        if(listeners[i] == NotificationEventListener.class) {
            listeners[i].eventStarted(evt);
        }
    }
}
```

![Figure 6-33: A performance risk due to the threading model.](image)
The analyst determined that there is a problem with this solution: if any one of the event listeners has a slow `eventStarted` method, it will affect other listeners too because all event listeners `eventStarted` methods are called in the same thread synchronously. If a new event listener is introduced into the system and its `eventStarted` method is slow, then the entire system has the risk of slowing down. The analyst noted that Java 5 has a new flexible threading model that addresses this synchronous event dispatching problem. The class `java.util.Executor` allows listeners to be executed asynchronously, using the concept of thread pools, so that slow listeners do not affect other event listeners. The SNAS team revealed that they are facing this issue and therefore the proposed solution, which takes advantage of the services of Java 5’s `Executor`, is being considered for the upcoming release at the time of writing this thesis in 2011.

6.5.8.2 GUI and Testability Analysis using External Entities

Here, we show how the analyst discovered testability issues due to the lack of a GUI abstraction layer. We illustrate how the unit testing of core logic can be affected due to the lack of GUI abstraction in the architecture.

The analyst already determined that the SNAS uses the `JPanel` class, which allows users to input data into various fields, and therefore wondered: Does the SNAS use the input verifier capability built-in the Java Swing architecture? The analyst queried the extracted code relations and found that the SNAS uses the `javax.swing.InputVerifier`, and overrides the call-back `verify` method as demanded by the Swing architecture. The analyst concluded, however, that the SNAS algorithm behind the `verify` method is not that trivial, which means it has to tested well, see Figure 6-34.
The ADAM in Action

Figure 6-34: Low testability and the lack of a GUI abstraction.

Unfortunately, the verify method is not easily testable without running the GUI and filling the input into the fields of panels. The analyst thus concluded that the risk is that the verify method is not tested in-depth (e.g., using JUnit) because it assumes that the data is provided by a GUI panel. Regular expressions can be error-prone, and unit test programs are needed to test them. If we refactor the verify method, as in Figure 6-35, thereby separating the GUI concept from the validation of the IP address, then the method isValidIpAddress can be easily tested using the JUnit test framework, for example, see Figure 6-36. This analysis, with the help of the knowledge base on Java’s Swing libraries, also detected other panels that verify the user input, such as range constraints, numeric constraints, and alpha-numeric constraints.
To sum up, even if one tries to construct a JUnit test suite, the source code has to be “open” for testing; in the same way logic is open for testing if the GUI is separated from it. The principle of abstraction and separation of concerns is one of the foundational pillars of software engineering [223] and [278], but perhaps developers (also code reviewers) either overlook this fact or there is a lack of concrete examples to really understand the concrete meaning behind this principle in order to apply in practice. That is why this chapter uses code snippets to demonstrate fundamental software engineering principles. Only because of the knowledge base of Java’s GUI classes, it was possible for us to easily discover code elements that affect testability.

### 6.5.9 An Analysis of the OS Concern using External Entities

In this section, we describe the analysis of the SNAS from the viewpoint of managing OS variants. We illustrate how the architecture should handle OS types to control complexity and facilitate testing on different OS types.
The model on which we base the analysis of the OS concern is derived from the following observations: a) every software system needs an OS to run and some systems need to run on more than one OS, and b) if the system needs to run on more than one OS then it must manage OS variants in some way. From this model, we derive the following questions: 1) What are the different OS types the system supports? 2) How does the system abstract underlying OS and when does binding to a particular OS take place? and 3) How are OS concerns separated from other concerns?

For the Java programming language, the snippet of the knowledge base, shown in Figure 6-37, helps in discovering architectural insights from the OS perspective.

Figure 6-37: (Snippet) knowledge base for analyses of the OS concern.

The analyst queried the extracted dependency model of the SNAS and discovered the files that use the exec method of the java.lang.Runtime class that helps for interacting with the OS (see Figure 6-37). The query showed that all OS commands are executed only through the PidService class (see Figure 6-38 (a)) defined in the framework folder. The analyst reviewed this class and found that the OS type and the OS command to run must be passed as arguments to the methods of PidService (see method parameters of Figure 6-38 (a)). The analyst extracted all dependencies to the methods of the PidService. The extracted dependency diagram showed that the higher-level layers (see Figure 6-38 (b)) must pass the OS command and the OS type to run the methods of the PidService. This clearly implies that OS concerns are mixed with other concerns, and the architecture did not offer a separate OS abstraction interface to hide the actual OS type.

Figure 6-38: A OS abstraction issue.

The recommended solution to this OS variation management problem is to let the PidService use Java’s APIs (e.g., System.getProperty(os.name)) and
automatically discover the OS type and then decide what OS commands to run. By doing so, the higher-level layers will be agnostic to OS variations. Thus, the complexity due to managing OS variants can be controlled in a clean and consistent way across all components, resulting in a good common look-and-feel. This example reminds us that having a system implemented in Java does not necessarily imply that the system is ready-to-run on all OS platforms; it has to be architected to manage OS variants. The devil is in the details; by analyzing the implementation using external dependencies we were able to discover novel insights and deep architectural problems, and furthermore offer constructive solutions where possible.

The analyst did not spend effort on identifying the different types of OS supported by the SNAS because when the analyst reviewed some of the configuration files used for configuring IP address, ports, etc., the analyst also saw a configuration variable called osType, with commented lines such as “set osType=UNIX or set osType=Windows”. It was commented in the configuration files that for testing purposes one can choose the OS type by modifying configuration parameters. Thus, the analyst admitted that it was some luck that pointed to the OS types used in the SNAS. Otherwise, the analyst had a strategy of searching for “/” or “\” used as path separators in file names, for example, to identify OS types. Usually, developers use “/” in UNIX and “\” in Windows. In addition, the analyst had a strategy to simply browse the high-frequent words present in the source code using the index of the system’s words as performed in a concordance analysis [172].

### 6.6 Summary of the SNAS analysis

#### 6.6.1 Differences in Common look-and-feel

During the analysis of SNAS, the analyst discovered differences in common look-and-feel, meaning that different components implement the same concern in different ways. For example, the sdif component has its own wrapper class offering logging services, whereas other components use the logging service wrapper defined in the reusable framework module. Similarly, the analyst discovered that the sdif has a different style for implementing database connections and the DAO layer. Discussions with the SNAS team revealed that the sdif was taken from the predecessor of the SNAS called SWSI. The SNAS team reported that they did a significant amount of refactoring to integrate the sdif with the SNAS, but that no time was spent on cleaning-up the architectural differences with other components. We noted earlier that some components by-pass wrappers to sockets, defined in the framework folder, and directly use Java socket libraries. Discussions with the SNAS team revealed that the wrappers were not there in the first-place, and therefore it is not practical to expect that all components use the wrappers. These are some “classical” examples of individual parts looking good but if we take one step up and see the whole system, there are architectural deviations in common look-and-feel, partly because of organizational factors and migrating or merging of existing systems.

#### 6.6.2 Summary of the Number of Files Reviewed

In order to determine the efficiency of the ADAM, especially from the point of view of how many files need to be analyzed in order to understand a reasonably large system,
the analyst took notes regarding the number of files that were reviewed during the analysis of the SNAS. It is important to note that the analyst has good experience with the RPA query language used for automatically finding files with certain characteristics. Thus, if an analyst not familiar with RPA-like query languages would do the analysis, then the analyst would have to use some other source code search tool and may have to open more files.

In addition, the analyst noticed that there was a learning effect during the analysis project that fortunately reduced the number of files to be reviewed as the analysis project progressed. For example, when the analyst opened a file to understand how data from the socket is read and delegated to data processors, the analyst also noted source code related to other concerns such as error-handling. The conclusion is that by reviewing 4% of files, see Table 6-15, the analyst was able to discover the modular and runtime structures and concern-specific architectural views of the SNAS, including communication, persistence, error handling, configuration settings, GUI, and OS variability concerns. In addition, the analyst was able to efficiently locate testability, performance, and maintenance risks.

Table 6-15: Summary of files reviewed for each goal

<table>
<thead>
<tr>
<th>Goals</th>
<th># of Files Manually Reviewed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery of Server-side Socket Ports</td>
<td>10 out of 1578 Java files</td>
</tr>
<tr>
<td>Discovery of Client-side Socket Ports</td>
<td>12 out of 1578 Java files</td>
</tr>
<tr>
<td>Discovery of Port Connections</td>
<td>5 out of 25 configuration files</td>
</tr>
<tr>
<td>Discovery of Data Beans</td>
<td>5 out of 384 Java bean files</td>
</tr>
<tr>
<td>Discovery of the DAO layer</td>
<td>10 out of 112 Java files dealing with database interaction</td>
</tr>
<tr>
<td>Discovery of GUI Architecture</td>
<td>20 out of 723 Java files in the mocclient, oamclient, shareclient folders</td>
</tr>
<tr>
<td>Discovery of OS Variability</td>
<td>4 out of 1578 Java file</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>61 out of 1578 Java files (i.e. 4% of Java files) were reviewed</strong></td>
</tr>
</tbody>
</table>

6.6.3 Lessons for Software Developers

In addition to being useful to other researchers, we believe the results presented in this chapter can also be used by software developers to learn about the relationship between software architectures and testability, performance, and maintenance risks. In particular, we presented several architectural features that can impede or facilitate testability, performance, and maintenance. In Section 6.5.5.4, we showed using the inter-component communication view that SNAS makes use of several performance inspired patterns including a) the transfer object design pattern for exchanging data among different components in order to overcome inherent limitations of remote procedure calls, and b) the reactor design pattern for handling multiple client sessions in order to minimize overhead due to context switching of threads for each client.

Unit testing in the presence of a database requires the architecture to have built-in flexibility to redirect all database interactions to in-memory dummy data [195]. In Section 6.5.6, we presented architectural views regarding the database concern using two components of the SNAS. In Section 6.5.6.1, we showed how one of the SNAS’ components (snif) implements the database access object (DAO) layer that clearly
abstracts and separates database concerns from other concerns. In Section 6.5.6.2, we highlighted clones in the DAO layer due to exception handling. In Section 6.5.6.3, we showed how to architect database interactions so that unit testing can be performed without having the database up and running. In Section 6.5.6.4, we showed how the SNAS system makes use of the connection pool design pattern for minimizing the performance overhead due to frequently creating and destroying connections to the database. In Section 6.5.7.2, we showed some testability problems in the sdif subsystem because connections to the database were initialized within constructors, which cannot be overridden. In Section 6.5.7.1, we showed an important issue that database errors were not abstracted and, thus, the high-level layers were not agnostic to the database concern.

In Section 6.5.8, we presented architectural insights of the GUI concern and discussed some performance and testability issues. In Section 6.5.8.1 and 6.5.5.5.2, we showed how the event notification mechanism and threading models can impact performance of the system. In Section 6.5.8.2, we showed a simple yet powerful example that affects unit testing because of the lack of abstraction and separation of GUI concerns from other concerns. In Section 6.5.9, we showed how the lack of an OS abstraction layer can increase (unnecessary) complexity.

One of the SNAS maintainers said that not only are the analysis results valuable, but the idea of leveraging external entities can also be used by other developers to understand and locate testability, performance, and maintenance risks in an effective and efficient manner.

6.7 Discussion

6.7.1 Can the analysis be done in any order?

Yes, we believe that the method of following external dependencies into the application can be done in any order related to concerns. For example, the analyst could have applied the ADAM first for discovering the architecture of the GUI concern and then for the persistence concern. However, the analyst found it useful to first discover the inter-component communication view before analyzing the implementation for each concern. For example, if the analyst did not perform a concordance analysis of word frequencies and/or did not know the fact that the SNAS is a distributed architecture using sockets as connectors, then the analyst may not have discovered the fact that GUI sends data to and receives data from a remote server. We also think that several architectural insights can be discovered before discovering the “big picture” shown in Figure 6-17. In fact the analyst first discovered database interaction structures presented in Section 6.5.6 before discovering the “big picture” that shows the runtime inter-process communication structure.

In another study, the ADAM was applied on an embedded real-time system at the US Food and Drug Administration (FDA), see Chapter 7. The system is implemented in C/C++ and is responsible for controlling an infusion pump device. In that system, the analyst was able to discover several architectural risks related to software testability, before discovering the inter-process communication view. Using the knowledge base, the analyst discovered that the system uses APIs of VxWorks as well as the
corresponding APIs of Windows. This led the analyst to discover the fact that VxWorks APIs are simulated by redirecting the function calls to equivalent windows APIs for testing purposes using conditional preprocessors. The analyst concluded that the system has been “tricked” to facilitate testing because the implementation has hard-wired dependencies to VxWorks APIs, which is redirected to Windows’ APIs. Similarly, by using dependencies to the external hardware interaction libraries, the analyst quickly found that the system can be configured to avoid interactions with hardware for testing purpose. This gives evidence that, in both scenarios, the system’s architecture did not define abstract interfaces, which would have facilitated testing in a clean way, for example, using a mock/stub implementation of abstract interfaces, as done in the NASA CFS architecture, see Chapter 3. Thus, that study offers evidence that it is not necessary to first discover runtime views before discovering other development views for different concerns.

### 6.7.2 Are there threats to the validity of the study?

There is a risk that the reverse engineered views are different from the mental models of the original architects, which we view as a threat to the validity of the study and the method. We evaluated our method by analyzing the SNAS documents after the independent study was completed. We found that our runtime view (see Figure 6-17) was in general consistent with the documented runtime view. There were two specific differences. First, the documented runtime view showed the database component in the same view, which was shown in a separate view in our case. Second, unlike our runtime view, the documented architecture did not show the communication ports. These two differences highlight an important fact that architects might fuse views into a single view, and might also leave certain elements out of an architecture document for next-level design.

The ADAM discovers architectures without analyzing all files of a system. It does that by classifying files as equal from a certain point of view. Thus, there is a risk that files are incorrectly classified as equal, especially if the sample is small in comparison to the number of available files. For example, the analyst was able to detect similarity patterns among 384 Java files without analyzing each of the files in detail. Based on this similarity, the analyst claimed that all 384 files are data beans because they all meet the criteria of a bean class (e.g., almost all methods are getters and setters, there are no calls to logging, they all implement the `Serializable` interface, etc.). The risk associated with this classification method is that there may be some files within the 384 files that are doing more than what a data bean is supposed to do. One risk mitigation strategy is to compute the text similarity among equivalent files because, in our experience, equivalent files tend to share quite a lot of words, as also mentioned in [172] and in Section 10.2.4.

Another possible threat to the validity is due to name clashes of keywords: For example, it might be possible that locally created header files coincidentally have the same name as the header files that indicate the use of an external entity. At this point, this is a risk that the analyst has to deal with by not coming to conclusion based on one header file name, instead search for the usage of several header file names in the system. If the analyst uses extracted code relations to match keywords of external entities, then this threat is avoided because the scope of each entity is defined.
A common threat related to any RE method is the correctness and completeness of the code relations extracted by parsers and/or regular expressions used for pattern matching. For example, in one system we were extracting all queue names, passed as arguments to send and receive message of external functions, using regular expressions, we realized later that something was wrong with our extracted data. It turned out that our regular expressions did not cover one specific scenario of line-breaks in function calls. These types of errors can influence the end result, for example, we might miss some architectural elements in the discovery process.

6.7.3 Is the ADAM repeatable on other systems?

It is worth stressing the fact that the analyst who performed the SNAS analysis neither possessed domain knowledge of the SNAS nor was involved in the development or maintenance of the SNAS system. Thus, we do not see any reason why the proposed method would not be repeatable on different domains. Experiences with formal query languages and basic knowledge of programming language libraries are important in order for other analysts to repeat and produce the same results discussed in the case study. The knowledge base of external entities can help analysts who are not familiar with the classes and methods of programming language libraries. In general, the analyst is like a detective, the method is trying to codify, reuse, and share experiences so that analysts can analyze even large systems efficiently.

Furthermore, for each concern the ADAM identifies, the necessary minimal set of elements needed to construct architectural views. This means that when analyzing other systems, the analyst knows “what to look for” to construct an architectural view for explaining the story and risks of a particular concern. For example, if the analyst has to evaluate another system for its testability in the presence of a database, the analyst has to follow the step by step guidelines given in Section 6.5.6.3 and Section 6.5.7.2 for detecting the presence of a database abstraction layer. If such an exercise is performed for several concerns, then we could formalize and develop a library of architectural meta-models for each concern. At the moment, in the software architecture literature, there is a lack of meta-models for each concern. As a consequence, automated reverse engineering of architectures is difficult because it is not necessarily clear what elements to look for to construct a view in order to highlight risks in the implementation.

Although the case study is in Java, the method has evolved from analyses of several systems implemented in the C/C++, ADA, and FORTRAN languages [275] and Chapter 2. The knowledge base of the ADAM has also been successfully applied on the NASA’s GMSEC software, implemented in C, C++, Java, and Perl, see Chapter 5 for details. The ADAM was also applied on another Java system to analyze the system for compliance to the CFR Part 11 regulations [48], which deal with data integrity and users login account management. The analyst used external dependencies to Java’s security classes and located files related to login management, and encryption and decryption strategies. Similarly, the analyst used dependencies to the Hibernate framework classes to understand and evaluate the auditing requirements related to the CFR Part 11. The ADAM was also applied on medical device software with promising results, as discussed in Chapter 7.

If the software is dynamic, similar to Ricoh’s office appliances (see Chapter 4), meaning that software components can be loaded, unloaded, stopped, and restarted at
runtime, then the ADAM would require dynamic analysis because the actual architectural configuration is known and changes at runtime. Note that in order to perform dynamic analysis we often have to do static analysis first (see Chapter 5) the ADAM offers support for static analysis.

6.7.4 Can reverse engineering discover architectural rationale?

No, it is difficult or even impossible to reverse engineer the rationale for architectural decisions. For example, in the SNAS case, one of the components’ (snif) database manager class creates instances of all DAO’s and reuse the same instances. On the other hand, another components’ (sve) database manager class does not create instances of all DAO’s upfront, instead DAO’s are instantiated for every new request and are not reused. We discussed this difference in the common look-and-feel with the SNAS team. It turned out that it is unclear to the current developers why the previous developers followed two different strategies. Similarly, we also discussed another aspect related to reusing of database connections for non-frequently used database tables and separate connections for frequently used database tables. The team members do not know on what basis the previous developers divided the database tables into not-frequently and frequently used types. Essentially, reverse engineering cannot extract all truths from the source code. Thus, developers need to record the rationale for architectural decisions they make.

6.7.5 What is the stopping criterion for the ADAM?

Similar to software testing, defining the stopping criterion is a non-trivial issue in architecture discovery and analysis of implemented systems, because usually there are several concerns to analyze and, thus the analyst can keep doing reverse engineering. Because the software architecture of a system is composed of several structures, including static and runtime structures, it is difficult to claim that one has successfully extracted the complete software architecture from the implementation. In practice, it is not possible or desirable to extract all structures. In the ADAM, the analyst typically concentrates on concerns including GUI, Persistence, Communication, and OS variants, because we found that a vast majority the source code can be covered when the implementation is analyzed with respect to these concerns. Further, these concerns shed light on how well the system has been architected for testability, performance, and maintainability. The stopping criterion is often specific to each project and its business context. For instance, in the NASA CFS project discussed in Chapter 2, the stopping criterion was decided after the analyst collected and verified about 40 architectural rules in the implementation. In other projects, the analyst may decide the stopping criterion simply based on the number of discovered testability, performance risks, and critical maintenance risks.

6.8 Comparison to Existing work

In addition to the existing work mentioned in Section 6.3, there are other works related to the ADAM on architecture analysis, knowledge-based program understanding, design pattern discovery, clustering, software clones, exception handling, and testability.
Architecture Analysis in the early phase of the lifecycle: The international working group on Software Architecture Review and Assessment (SARA) has produced a report that offers best practices and practical guidelines for the task of reviewing software architectures primarily at an early stage, before the detailed design and implementation phase has started [216]. The notion of reviewing the architecture with respect to concerns, such as persistence, error handling, GUI, and OS variants, is included in SARA. This notion can be mapped to the “Concerns” dimension of the ADAM. Analysis of quality properties, such as performance and testability of the architecture, is also included in SARA. This corresponds to the “Quality Properties” dimension of the ADAM. In our opinion, SARA could also be used for reviewing an existing implemented system. In such a scenario, the ADAM could be used in conjunction with SARA because the former can be used to discover the architecture from source code, where the architecture is the key input needed for SARA.

The authors of the SARA report note that: “the selection of a specific technology such as CORBA or EJB has profound implications on the architecture and non-functional requirements”, see page 13 in [216]. Based on the very same premise, the ADAM uses external dependencies as the main driver to reverse engineer the implemented architecture. In [248], Rozanski and Woods use a catalog of concerns to construct and analyze views of the architecture. In [24], Binder discusses design principles for testability. In [310], Wirfs-brock discusses best practices for managing exceptions, including abstraction of exceptions raised by lower-level layers. The ADAM uses these principles and best practices for evaluations of implemented systems. The ADAM shares the vision of storing best (or problematic) practices with Booch’s handbook of software architectures [29].

Knowledge-based Program Understanding: The ADAM share the LaSSIE’s high-level goal of solving the “invisibility” problem inherent in software systems [65]. LaSSIE helps in understanding of how and where features are implemented using its domain ontology. In our opinion, LaSSIE does not focus on discovering testability and performance risks as discussed in the ADAM. We are exploring ways to enrich our knowledge base with domain concepts for facilitating a domain-oriented architectural reasoning. The MIDAS approach uses a knowledge base for automatic reengineering of database programs from the network model to the relational model [51]. Blaha performs reverse engineering of database schemas to understand and identify design and modeling errors on the relational schema level [26]. MIDAS and Blaha’s method do not extract architectural views that explain how the system abstracts and interacts with databases. In contrast to the MIDAS method and Blaha’s method, the ADAM supports discovering and analyzing the database interaction architecture and its testability risks. The MORPH process uses a knowledge base for migrating text-based user interfaces into GUIs [207]. The ADAM supports discovering and analyzing the GUI architecture of systems that already have a GUI.

Design Pattern Discovery: In [68], Dong et al. review methods and research tools for recognition of design patterns from the source code. In [45], Canfora and Di Penta provide a retrospectives view on reverse engineering including architecture recovery and pattern identification methods. We have shown here that there are several patterns implemented simply by using the programming language libraries, which are often excluded in many methods. It would be interesting to investigate how pattern discovery
research could benefit from a knowledge base of external dependencies. We position design patterns different from architectures, in that the former is a detailed concept used to implement the latter. Therefore, the ADAM aims for discovering architectural abstractions first and then related design patterns, if any, used for implementing the architecture.

Clustering: In [193], Maqbool and Babri present a survey on clustering methods and how they could shed some light on the software structure. One of the challenges is that, especially in GUI parts, methods are invoked indirectly using event-driven concepts and implicit invocations [110], and thus the call graph is often broken into disconnected graphs. Also, many systems contain intermediate connectors for communication. Another challenge is that the architectural abstractions (e.g., Interfaces, Connectors, and Components) are invisible in the output of clustering, and is not easy to do a detailed analysis, because all concerns are still part of the clustered model. Clustering methods discover module views and do not discover runtime views. In general, clustering methods do not give names to components or summarize in a few sentences the role played (e.g., DAO layer, OSAL layer) by them. After all, it is the name and the brief summary that helps in understanding the architectural roles played by a huge collection of files. It would be fruitful to investigate how the existing clustering methods behave if they are combined with a knowledge base.

Software Clones: In [165], Koschke discusses several clone detection methods in detail. In the existing literature, we did not find discussions on how clones and the software architecture are related. Thus, it is difficult to make use of the detected clones for performing constructive improvements without insights on the software architecture. That’s why the ADAM interprets clones in an architecture-centric focus. Our focus was on analyzing the detected clones by concentrating on one concern at a time and interpreting them using the discovered software architecture so that we can offer constructive advices where possible. For example, we have discussed clones in GUI panels and across files in database abstraction layers. We offered concrete advices on how to migrate to new technologies in order to overcome inherent cloning problems due to the Java language.

Exception Handling: In [244], Robillard and Murphy used the Jex tool for analyzing the flow of exceptions. We analyzed exceptions using dependencies to external entities and selecting a concern of interest. We interpreted the flow of exceptions from an architecture point of view. For example, we have shown cases where the database is abstracted but database errors had leaked into the higher-level layers. Also, using the knowledge of dependencies to external entities, we have shown how we can find how the system handles specific exception types such as the socket timeout exception or host not available exception. Thus, we believe Jex can also benefit from a knowledge base. In [257], Shah et al. report that, in their survey, novices make mistakes in exception handling. Of course, the truth is in the source code, experts also make mistakes because exception handling is often not given much attention during the architecture design.

Assessment of Testability: We share the spirit of understanding “What is it that makes code hard to test” as asked by Bruntink and van Deursen in [42]. In contrast to their testability assessment model, our method covers testability in the presence of a GUI or a database. In [77], Feathers offers a piece of “clean” code (e.g., good
method/variable names, comments) that was not easy to test because of a hard-binding to a remote stock server, which cannot be replaced by a dummy server for testing purposes. We collect such anti-testing patterns into our knowledge base and analyze implemented systems for the existence. In Chapter 3, we provided deep insights on “What types of architectural decisions make unit testing easier/harder” in a product line context. In Chapter 7, we covered the testability subspace of the ADAM and discover architectural views of medical device software to evaluate testability.

6.9 Closing Remarks

This chapter has offered evidence that by leveraging the semantics of external entities, we can efficiently discover not only the software architecture hidden in the implementation but also locate testability, performance, and maintenance risks. Most architectural problems are hidden deep in the source code. As shown in this chapter, external entities help us to efficiently locate the details where devils hide. This chapter also proposed a knowledge base of the ADAM for reverse architecting and analysis. Construction of a knowledge base is an investment. We have discussed how one can incrementally build a knowledge base over time using external entities used by systems under analysis. If your organization is regularly conducting architectural analysis of several implemented systems, you could reap the benefits of your investment in a knowledge base. Our future prospects include a) improving the usability of the tool-chains so that analysts can easily add their knowledge of analyzing commercial systems, and b) building “intelligent” analysis environments to further improve the productivity of our analysts.
Chapter 7
Architecture Discovery of Medical Device Software

7.1 Abstract

In year 2010, new research was started at the FDA to investigate the benefits of integrating architecture analysis into safety evaluations of medical-device software. Due to the complexity in setting up testing environments for such software, the FDA is unable to conduct large-scale safety testing; instead, it must rely on other techniques to build an argument for whether the software is safe or not. The architecture analysis approach, formalized using relational algebra, is based on reconstructing abstract, yet precise, architectural views from source code to help build such arguments about safety. This chapter discusses the use of the formal approach to analyze the Computer-Assisted Resuscitation Algorithm (CARA) software, which controls an infusion pump designed to provide automated assistance for transfusing blood. The results suggest that a) architecture analysis offers many insights related to software quality in general and testability (i.e., the ease of testing) and its impact on safety in particular, and b) architectural analysis results can be used to help configure static analysis tools to improve their performance for verifying safety properties.

Keywords: Medical device, Safety, Testability, Verifiability, Static Analysis, and Reverse Architecting.

7.2 Introduction

Embedded software in medical devices is increasing in content and complexity. For example, state-of-the-art cardiac pacemakers may contain up to 80,000 lines of code (LOC), while infusion pumps may have more than 170,000 LOC [81]. These devices must perform safely and effectively, and the US Food and Drug Administration (FDA) has the regulatory responsibility for making determinations about safety and effectiveness in the case for equipment sold in the United States. However, recent studies using the FDA database of medical device failures are pointing to increasing failure rates of medical devices due to software errors [301] and [177]. In 1996, 10% of medical devices recalls were caused by software-related issues. In 2006, software errors in medical devices made up 21% of recalls [112]. From 2005 to 2009, more than 10,000 complaints were received annually by the FDA about infusion pumps, including reports of 710 patient deaths linked to problems with these devices [199]. A number of these deaths were attributed to malfunctioning device software.

---

11 Based on the paper appeared in the IEEE Working Conference on Software Architecture (WICSA 2011) paper [100].
As software in medical devices has become ubiquitous [166] and [264], it is not surprising to see a rise in the number of software-related problems. Increased complexity further contributes to the problem. What is disconcerting is the apparent lack of disciplined (safety critical) software engineering practices found during investigations of many of these problems.

The FDA increased its involvement in reviewing the development of medical device software in the mid-1980s when software coding errors in a radiation therapy device contributed to the lethal overdose of a number of patients [181]. The FDA has established a software laboratory within the Center for Devices and Radiological Health (CDRH), the Office of Science and Engineering Laboratories (OSEL), to evaluate software engineering technologies and to support investigations of potential software errors in medical devices. Due to the complexity in setting up test environments for medical device software, the FDA cannot in general test such software for safety and must rely on other techniques to build an argument for whether it is safe or not. As part of its investigations, analysts at the OSEL use state-of-the-art static analysis tools to examine source code to understand and identify the root cause of device failure (e.g., [140] and [224]). Static analysis tools verify the absence of runtime errors, such as null pointer dereferencing, and buffer overruns, by evaluating the syntax of the code without executing it.

While sophisticated static analysis tools can detect serious defects in software, they cannot uncover all safety problems in code. In particular, many safety-pertinent issues arise from design problems; an overly complex architecture, especially one that deviates from the documented design [185], for example, can result in inadequate testing and safety problems in the field. Static analysis tools are not intended to support these kinds of analysis. In this chapter, we propose that safety analysis should include a detailed architecture analysis to help verify software more comprehensively. This analysis can help build an argument about safety based on statements such as a) software that does not have a well-engineered architecture may be unsafe and b) software that has low testability most likely has not been tested enough and therefore may be unsafe.

### 7.2.1 Context

Premarket approval or clearance is the FDA process of a science based regulatory review of medical devices prior to being placed on the market. If a product is labeled, promoted or used in a manner that meets the following definition in section 201(h) of the Federal Food Drug & Cosmetic (FD&C) Act [287] it will be regulated by the Food and Drug Administration (FDA) as a medical device and is subject to premarketing and post marketing regulatory controls.

A device is: "an instrument, apparatus, implement, machine, contrivance, implant, in vitro reagent, or other similar or related article, including a component part, or accessory which is:

- recognized in the official National Formulary, or the United States Pharmacopoeia, or any supplement to them,
intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man or other animals, or

- intended to affect the structure or any function of the body of man or other animals, and which does not achieve any of its primary intended purposes through chemical action within or on the body of man or other animals and which is not dependent upon being metabolized for the achievement of any of its primary intended purposes."

Medical devices range from simple tongue depressors and bedpans to complex programmable pacemakers with micro-chip technology and laser surgical devices. In addition, medical devices include in vitro diagnostic products, such as general purpose lab equipment, reagents, and test kits, which may include monoclonal antibody technology. Certain electronic radiation emitting products with medical application and claims meet the definition of medical device. Examples include diagnostic ultrasound products, x-ray machines and medical lasers.

As part of a premarket device evaluation, the FDA reviews design artifacts such as architecture, requirements, hazard analysis reports, and test case results. They do not, as a rule, review device source code. The primary goal of the FDA’s Division of Electrical and Software Engineering (DESE) is to provide engineering expertise and training to device reviewers and investigators as needed. In the latter case, this may include forensic analysis of a device failure. The FDA worked with Fraunhofer CESE under a collaborative research and development agreement (CRADA) in order to strengthen its software engineering domain expertise related to software safety as well as software forensics. In this collaboration, we researched the benefits of computer-aided software reverse engineering in general, and architectural analysis of quality properties (e.g., safety and testability) in particular. In addition, we researched how architectural analysis can complement traditional static analysis tools that verify coding errors of software.

The CARA medical device software was used in these studies to demonstrate the value of reverse architecting for evaluating the quality properties of medical device software. It should be noted that this research was not conducted in reaction to the failure of the CARA software in the field. At the time of writing the thesis, the CARA software was not deployed in the field.

### 7.2.2 Detecting flawed architectures

All medical devices need to comply with quality regulations to ensure that they meet applicable current good manufacturing practices (CGMPs) [47]. In the case of software, this means ensuring that the best software-development techniques are employed during implementation. One way to assess the quality of the software-development process is to reason about the architecture of the implemented source code (in contrast to the intended architecture that may be described in design documentation). For example, if the device has modular blocks, then this suggests that a) individual blocks can be independently unit tested [101], and b) the device can be formally verified for safety properties by focusing testing and verification activities on each modular block [23]. If the system lacks well-defined module boundaries, then it is reasonable to question the testability and verifiability of that device. A system that is difficult to test
and verify most likely has not been tested enough and therefore is less likely to perform as intended. Such software may be considered flawed, potentially unsafe, and should require more detailed and persuasive arguments about its fitness for it to be approved.

Most static analysis tools cannot easily determine the quality of the software in terms of its architecture; they do not have the ability to tell whether the software being analyzed is well-structured. It falls upon the FDA analysts then to manually peruse software and its documentation to analyze and reason about its architecture. While device manufacturers do submit architectural design diagrams to the FDA, these are typically at a “high level” of abstraction. Their relationship to source code, testability and verifiability, and safety decisions is often tenuous due to missing details and traceability.

7.2.3 Overview of the Approach

We developed a large body of knowledge of architectural analysis of implemented systems (e.g., Chapter 2, Chapter 3, and Chapter 5) in the safety-critical aerospace domain. The FDA believes that analysis methods proven in other safety-critical domains should be evaluated and, if appropriate, adopted in the medical-device domain [81]. New research is underway at the OSEL to perform architecture-level analysis of medical software by reconstructing architectural structures (e.g., [120], [170], and [131]) from source code. The goal is to use such abstractions to draw conclusions about software quality properties that contribute to safety (e.g., testability and verifiability).

Using the CARA software, we formalize and demonstrate an architecture-reconstruction approach that extracts both static and runtime structures semi-automatically from source code to facilitate analysis of software safety. In order to recognize architectural features in the source code, the approach uses Fraunhofer’s knowledge base, including tool support, to analyze the CARA system from different architectural viewpoints, such as modularity, layering, inter-task communication, and built-in support for testing and verification.

The CARA software was independently analyzed from two perspectives. First, an FDA analyst ran a static analysis tool on the software. The tool reported that operating system (OS) library files needed to analyze the CARA were missing. Second, Fraunhofer analysts, working in the OSEL software laboratory, used their architecture reconstruction approach to establish architectural views of the CARA that show a) how the CARA architecture supports configuration points for running on different types of OSes, and b) how the source code files are transformed into runtime components. These insights helped the FDA analyst to configure the static analysis tool in order to overcome the missing files issue and perform unit-by-unit software verification for improved performance.

Contributions of this chapter. This chapter contributes the following:

1. An improved understanding of how architectural analysis of software implementations can offer additional insights on software quality, especially testability and verification risks, that are not fully derivable from state-of-the-art static analysis tools.
2. An architecture reconstruction, or “reverse architecting” process and algebraic formalization of our approach for the analysis of static and runtime structures from software implementations.

3. A detailed case study demonstrating the use of extracted architectural knowledge to configure static analysis tools for improved performance.

While other architecture reconstruction techniques exist (e.g., [124], [148], [169], and [253]), to our knowledge, this case study is the first to make use of the extracted static and runtime structures to reason about software testability at the architecture level, and use this information to configure static analysis tools for improved analysis.

### 7.3 Reverse Architecting of the CARA

**Overview of the CARA:** The CARA is a control-loop software system implemented in C/C++ that infuses resuscitation fluids in a patient to maintain the blood pressure. It displays a graph showing the subject’s blood pressure and the infusion pump’s flow rate. In the auto-control mode, a graph indicates the set-point to which the CARA maintains the blood pressure. In the manual mode, the CARA monitors the speed and the volume pumped.

The architecture analysis process used to analyze the CARA has five major steps, see Figure 7-1.

![Figure 7-1: Reverse Architecting Process.](image)

**Step 1 - Clean the codebase:** The goal of this step is to identify and filter out noise in the codebase. The codebase often contains duplicate files, which is confusing both to parsers and analysts because of duplicate definition of code elements. Analysts at Fraunhofer have developed an indexing tool that identifies duplicate files based on information retrieval technologies. The Fraunhofer analyst applied the tool on the CARA’s codebase and automatically detected several duplicate files, e.g., temporary duplicate files. The analyst excluded all but one copy of each set of duplicate files because the other copies were redundant. The analyst rebuilt the index after excluding...
duplicate files because the next steps require a clean index. The output of this step is a cleaned version of the input codebase for next steps.

**Step 2 - Generate summary:** The goal of this step is to identify architecturally significant features present in the source code. Fraunhofer analysts have analyzed more than two-dozen real-world industrial implemented systems for the past ten years or so. These engagements enabled us to build a knowledge base of keywords present in external entities, such as programming language libraries, OS libraries, and commercial off-the-shelf (COTS) software. The knowledge base stores keywords from header file names and function names that play significant roles in the discovery of the implemented architecture. For example, Table 7-1 shows a small snippet of the knowledge base. If a system contains `vxWorks.h` and `taskLib.h`, which handles tasks in the VxWorks OS, then it is reasonable to assume that the system makes use of several tasks and runs on VxWorks.

Table 7-1: Snippet of the knowledge base

<table>
<thead>
<tr>
<th>OS Type</th>
<th>Header file</th>
<th>Function name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>VxWorks</td>
<td>taskLib.h</td>
<td>taskSpawn</td>
<td>Spawn a task</td>
</tr>
<tr>
<td>VxWorks</td>
<td>msgQLib.h</td>
<td>msgQCreate</td>
<td>Creating a message queue</td>
</tr>
<tr>
<td>VxWorks</td>
<td>msgQLib.h</td>
<td>msgQSend</td>
<td>Send a message to a queue</td>
</tr>
<tr>
<td>VxWorks</td>
<td>msqQLib.h</td>
<td>msqQReceive</td>
<td>Read a message from a queue</td>
</tr>
<tr>
<td>Windows</td>
<td>windows.h</td>
<td>CreateThread</td>
<td>Creates a thread to execute</td>
</tr>
</tbody>
</table>

We have developed a summary generator that takes as inputs the knowledge base and the codebase and outputs a summary of the codebase for further analysis. The analyst applied it to the CARA codebase resulting in a summary similar to Figure 7-2. The summary provides insights about the implemented architecture, including a) the CARA support two OSes, b) the CARA uses multiple tasks, and c) the CARA uses message queues for inter-task communication [123].

<<CARA>> has
- Several keywords of Windows libraries (e.g. `windows.h`)
- Several keywords of VxWorks libraries (e.g. `vxWorks.h`)
- Multitasks because it has the `taskSpawn` keyword
- Inter-Process because it has `msgQSend/Receive` keywords
- Semaphores because it has `semBCreate` and `semTake` keywords
- GUI because it has keywords of GUI libraries (e.g. `afxwin.h`)

Figure 7-2: Summary of the CARA using the knowledge base.
Step 3 - Discover static structures from code: The goal of this step is to reverse architect static structures from the source code to facilitate testability analysis. Static structures show the organization of source code files (in modules) on the file system and the inter-dependencies of these modules. Static structures show if the modules are organized into layers and potential inter-dependencies. Static structures also indicate a) support of graphical user interfaces (GUI), b) interactions with external hardware (e.g., blood pressure monitor, infusion pumps), and c) use of OS features.

The summary generator of the previous step reported that the CARA has a GUI and uses Windows as well as the real-time VxWorks operating systems. Using the output of the summary generator, in Section 7.3.1, we extracted the static structures of the CARA from the perspectives of GUI, OS, and hardware interaction to facilitate reasoning about testability. We have selected these perspectives because our experience indicates that testability is influenced by how the implementation abstracts certain components. If the implementation has hard bindings to GUI’s, hardware elements, and real-time OS features, then it is typically difficult to test [101]. We used the Relation Partition Algebra (RPA) language [78] and [169] to discover static structures from the source code. We used the Fraunhofer SAVE-LIGHT tool to visualize static structures. The output of this step is a set of static structures and insights about testability.

Step 4 - Discover runtime structures from code: The goal of this step is to reverse architect runtime structures from the source code. Runtime structures capture how the source code is partitioned into concurrent tasks and how the tasks communicate and synchronize with each other [120]. The analyst has to discover runtime structures because there could be a many-to-many relation between modules and runtime components (e.g., tasks). The analyst used runtime structures to a) understand how static structures are transformed into tasks and how the tasks communicate and synchronize, and b) reason about testability and variability of tasks. Often medical devices are implemented with multitasking capabilities. CARA is one such example, as reported by the summary generator. Based on the summary report, we formalize the extraction of runtime structures using RPA. The formulas presented in Section 7.3.2 automate the process of reverse architecting runtime structures using extracted code relations. We used the Prefuse interactive visualization tool [236] for visualizing runtime structures. The output of this step is a collection of runtime structures as well as insights about testability.

Step 5 - Configure and run static analysis tools: The goal of this step is to demonstrate how static analysis tools can be configured using the extracted static and runtime structures for improved performance, see Section 7.4.

7.3.1 Discovering Static Structures

The goal of this section is to reason about testability by reverse architecting static structures of the CARA.

7.3.1.1 Static structure of OS abstraction

The summary generator indicated that the CARA codebase has keywords related to both VxWorks and Windows libraries, so the analyst decided to investigate how the OS
variants are handled in the implementation. The analyst analyzed the file `VxWorksSim.cpp` because it uses keywords of both OS types and found that the CARA can simulate VxWorks using Windows libraries. The analyst used a regular expression to extract the `Include` relation and found that files that include the standard `VxWorks.h` header file also include the `VxWorksSim.h` file, see Figure 7-3. The simulation capabilities are implemented in the file `VxWorkSim.cpp` and declared in `VxWorkSim.h`.

Figure 7-3: (Snippet) Static structure of OS variants.  

Since it makes little sense to include header files of more than one OS type, this caught the analyst’s attention. The analyst then analyzed some related files (e.g., `cara_interface.c` in Figure 7-3), and found that the CARA has a configuration point based on a conditional preprocessor statement: `ifdef WIN32` includes `VxWorksSim.h`, otherwise `VxWorks.h`.

**Testability and OS Variants.** The CARA is a real-time embedded software system [238]. However, this analysis of the OS variants showed us that the CARA can be configured to execute on machines with Windows, without requiring the VxWorks real-time features to be available. This OS dependency view highlights the fact that the CARA has built-in capability to facilitate testing. From FDA’s point of view, this view is valuable because it conveys that the developers of the CARA most likely tested the software.

**OS variants and an architectural issue.** This snippet of the architectural analysis shows that the CARA architecture did not separate OS concerns from other concerns in a clean way. The knowledge of OS variants are scattered in different parts of the system. In some files, there are sixty `ifdef WIN32` statements, for example. This extra complexity of handling OS variants with conditional preprocessor statements and simulation using Windows could have been avoided if there were an OS abstraction layer (OSAL) [219] built into the architecture, similar to the NASA CFS, discussed in Chapter 2 and Chapter 3. The OSAL would facilitate adding and removing new OS types without changing the source code of higher-level layers because the OSAL would

---

12 For readability, we manually redrew all figures.
offer OS independent interfaces. From FDA’s point of view, complexity can be a source of problems. If the source code is complex, then it is difficult to understand and test, and bugs could be lurking in the code. An overly complex architecture might even be considered flawed and unsafe. Developers should strive to minimize complexity by using appropriate architecture design principles (e.g., an OS abstraction layer).

7.3.1.1.1 Parsing the source code of the CARA

Based on the finding that the CARA makes use of mutually exclusive OS variants, the analyst ran the Fraunhofer’s C/C++ parser to collect data for further analysis. The analyst configured the parser to choose Windows OS because VxWorks was not installed. The extracted data is listed in Table 7-2 and Table 7-3.

Table 7-2: Definition of extracted sets of static structure data

<table>
<thead>
<tr>
<th>Set</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>List of functions</td>
</tr>
<tr>
<td>GV</td>
<td>List of global variables</td>
</tr>
<tr>
<td>Class</td>
<td>List of classes</td>
</tr>
<tr>
<td>Files</td>
<td>List of files</td>
</tr>
<tr>
<td>PS</td>
<td>List of preprocessor symbols</td>
</tr>
</tbody>
</table>

Table 7-3: Definition of extracted relations of static structure data

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Function</td>
<td>Function</td>
<td>Function f calls function g</td>
</tr>
<tr>
<td>FG_use</td>
<td>Function</td>
<td>GV</td>
<td>Function f refers to global variable v</td>
</tr>
<tr>
<td>Include</td>
<td>File</td>
<td>File</td>
<td>File m includes file g</td>
</tr>
<tr>
<td>Inherit</td>
<td>Class</td>
<td>Class</td>
<td>Class C inherits from Class P</td>
</tr>
<tr>
<td>Partof</td>
<td>Child</td>
<td>Parent</td>
<td>Function (child) is defined in file (parent)</td>
</tr>
<tr>
<td>Fsl</td>
<td>Function</td>
<td>Integer</td>
<td>Definition of Function f starts at line n</td>
</tr>
<tr>
<td>Fed</td>
<td>Function</td>
<td>Integer</td>
<td>Definition of Function f ends at line n</td>
</tr>
<tr>
<td>MP_use</td>
<td>File</td>
<td>PS</td>
<td>File m uses a preprocessor symbol p</td>
</tr>
</tbody>
</table>

7.3.1.2 Static structure of hardware abstraction

The analyst visualized dependencies from the CARA files to external files using the Fraunhofer SAVE-LIGHT Tool. External files are those files which are used by the CARA but do not have a definition. The analyst found that the CARA uses an external header file called dscud.h. Because the knowledge base did not contain the dscud.h, the analyst searched the Internet and found that it is related to device driver software called the Diamond Systems Corporation Universal Driver (DSCUD) [286]. The analyst also found a description of the DSCUD in the premarket submission.
document. The DSCUD offers functions for conversions of analog input used in combination with analog input hardware boards that measure blood pressure.

From a testability point of view, the natural question is whether the CARA is vendor locked or whether it uses abstract interactions with hardware boards so that testers can test the CARA software on their regular computers without access to analog signals and hardware sensors. The analyst wrote a RPA query that helped extract a view of how the CARA interacts with the DSCUD. This revealed that the CARA uses eight C functions and four data types of DSCUD, accessed through a wrapper only. The wrapper is declared in cara_da.h and implemented in cara_da.c, which includes the COTS software header file dscud.h, see Figure 7-4. Thus, the implemented architecture is not vendor locked because of the wrapper and all dependencies to the external COTS software are only through the wrapper (cara_da.c).

![Diagram 7-4: Wrapper view of DSCUD COTS abstraction.](image)

The analyst queried the Call relation to identify functions that use the wrapper functions of the DSCUD COTS, see Figure 7-5. The purpose of this query was to understand if it is possible to redirect the control flow to stub functions in order to avoid accessing the functions of the unavailable COTS software during unit testing.

![Diagram 7-5: (Snippet) Call graph of COTS interaction.](image)
The code relations, as revealed by the query, do not contain stub or mock definitions of the wrapper functions of cara_da.h. Thus, the analyst analyzed higher-level layer functions (i.e., functions defined in cara_interface.c) that call the wrapper. Within cara_interface.c there is a conditional preprocessor statement called #ifndef TEST that decides whether or not to use the wrapper functions. If the TEST switch is disabled, the CARA will interact with the hardware board through the wrappers, otherwise a constant will be returned. For example, in Figure 7-6, the AD wrapper function of the hardware that will only be used if the TEST is disabled.

```
double CARA_READ_EMF(void)
{
    #ifndef TEST
        /* Read the EMF value from the A/D board. */
        float Actual_Value;
        Actual_Value = AD(EMF_CHANNEL);
        ...
        return (double) Actual_Value;
    #else
        return CARA_EMF_VALUE;
    #endif
}
```

Figure 7-6: Avoiding hardware interaction for testing.

Using the extracted preprocessor symbol usage relation MP_use, the analyst confirmed that the #ifndef TEST statement is used in all places that interact with the hardware board using the intermediate wrappers (defined in cara_da.h). In general, over usage of IFDEFs can cause confusion and increase complexity.

7.3.1.3 Static structure of GUI abstraction

The source code files of the CARA were delivered as two folders. Using the knowledge base, it became clear that the files in one folder deal with the GUI concern while the other folder deals with non-GUI concerns. By visualizing dependencies to external GUI libraries, the analyst concluded that the CARA implementation separates the GUI concern from core logic parts, which is a good architectural principle. This structure indicates that the CARA can be tested without the GUI, because testing in the presence of a GUI is difficult, if not impossible.

7.3.1.4 Insights of extracted static structures

The first insight is the indication that the developers of the CARA indeed tried to test the software without depending on necessary hardware-in-the-loop [268], see Figure 7-6. The second insight is that the testing feature was not designed into the software; it was inserted into the code afterwards. We can say this because of the extensive use of #ifdef to choose between the production and the test code. This could have been avoided by defining a stub implementation of wrapper functions. At link-time,
developers/testers can then choose whether to bind to the real wrapper implementation or to the stub implementation, similar to the NASA CFS [101].

7.3.2 Discovering Runtime Structures

The goal of this section is to reason about testability by reverse architecting runtime structures of the CARA.

7.3.2.1 Data extraction for runtime structure

The summary generator reported that the CARA system uses keywords related to OS libraries that deal with multitasking and inter-task communication using message queues. The analyst extracted associated data to reverse architect runtime structures by developing regular expression based pattern matchers of function signatures of standard OS libraries. These pattern matchers emit data defined in Table 7-4 and Table 7-5. For example, in order to extract which function writes to which queues (i.e., the $F_{Q write}$ relation), our pattern matchers emit a table (called Send_Table) that contains file names, queue identifiers, and the line numbers that match the syntax shown in Figure 7-7. The first argument to $msgQSend$ is the queue identifier. The C/C++ parser emitted starting and ending line numbers of each function definition (see $F_{sl}$ and $F_{el}$ in Table 7-3). The analyst performed a join operation using the relations $F_{sl}$ and $F_{el}$ with the table Send_Table and derived the $F_{Q write}$ relation.

```
msgQSend (msgQId, buffer, nBytes, timeout, priority)
```

Figure 7-7: VxWorks syntax to send messages on a queue.

<table>
<thead>
<tr>
<th>Set</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>List of task names</td>
</tr>
<tr>
<td>Queue</td>
<td>List of queue identifiers</td>
</tr>
<tr>
<td>SemId</td>
<td>List of semaphore identifiers</td>
</tr>
</tbody>
</table>

Table 7-4: Definition of extracted sets of runtime structure data
In general, simple pattern matching would not be sufficient to extract this type of data because queue identifiers can be passed as arguments among different functions or be dynamically generated, where the actual value of queue identifiers cannot be retrieved until a comprehensive data-flow-based evaluation of the regular expression is conducted. However, in the CARA, all queue identifiers and semaphore identifiers are global constants. The analyst also found that there is no indirection of the function pointer (entryFunPtr in Figure 7-8) passed as an argument to the taskSpawn function for creating an OS task. In fact, the taskName and entryFunPtr are hard-coded strings in the CARA codebase. Thus, our regular expressions based pattern matcher could extract the necessary data.

```plaintext
(taskSpawn (taskName, priority, options, stackSize, entryFunPtr, ...))
```

Figure 7-8: (Snippet) VxWorks syntax to create a task.

### 7.3.2.2 Discovery of runtime structures

The analyst used the extracted relational data in Table 7-5 and derived architecturally significant relations by using RPA queries. These derived relations exposed the latent runtime structures of the CARA. Several runtime structures were immediately derived using RPA’s relational and set operators, including the powerful transitive closure operator for reachability analysis. Using RPA queries, the analyst discovered several runtime structures including the following examples:

1. Which code elements are shared among tasks?
2. Which code elements are unique to tasks?
3. Which tasks create/delete/restart other tasks?
4. Which tasks read from/write to queues?
5. Which tasks create/take/give semaphores?
The automatically derived relations are listed in Table 7-6. Each derived relation refers to one concern of the runtime structures. Each derived relation was visualized separately to obtain concern-specific insights. These relations were automatically extracted by running the corresponding queries formulated in Figure 7-10. The algebraic notations are explained in Figure 7-9.

Table 7-6: Automatically derived runtime structures

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFuse</td>
<td>Task</td>
<td>Function</td>
<td>Task t uses function f</td>
</tr>
<tr>
<td>TGuse</td>
<td>Task</td>
<td>Variable</td>
<td>Task t uses variable g</td>
</tr>
<tr>
<td>TMenter</td>
<td>Task</td>
<td>File</td>
<td>Task t entry point is file m</td>
</tr>
<tr>
<td>TMuse</td>
<td>Task</td>
<td>File</td>
<td>Task t uses file m</td>
</tr>
<tr>
<td>TTcreate</td>
<td>Task</td>
<td>Task</td>
<td>Task t creates task u</td>
</tr>
<tr>
<td>TTdelete</td>
<td>Task</td>
<td>Task</td>
<td>Task t deletes task u</td>
</tr>
<tr>
<td>TTrestart</td>
<td>Task</td>
<td>Task</td>
<td>Task t restarts task u</td>
</tr>
<tr>
<td>TQcreate</td>
<td>Task</td>
<td>Queue</td>
<td>Task t creates queue q</td>
</tr>
<tr>
<td>TQdelete</td>
<td>Task</td>
<td>Queue</td>
<td>Task t deletes queue q</td>
</tr>
<tr>
<td>TQread</td>
<td>Task</td>
<td>Queue</td>
<td>Task t reads from queue q</td>
</tr>
<tr>
<td>TQwrite</td>
<td>Task</td>
<td>Queue</td>
<td>Task t writes to queue q</td>
</tr>
<tr>
<td>TScreate</td>
<td>Task</td>
<td>SemId</td>
<td>Task t creates semaphore s</td>
</tr>
<tr>
<td>TStake</td>
<td>Task</td>
<td>SemId</td>
<td>Task t takes semaphore s</td>
</tr>
<tr>
<td>TScancel</td>
<td>Task</td>
<td>SemId</td>
<td>Task t cancels semaphore s</td>
</tr>
<tr>
<td>TStakecancel</td>
<td>Task</td>
<td>SemId</td>
<td>Task t cancels semaphore s</td>
</tr>
</tbody>
</table>

For example, in order to extract which set of functions are needed to run a task one can run the query shown in equation (1) of Figure 7-10. The query works as follows: The first part computes the transitive closure (*) after computing the union of extracted relations TFenter and Call. The second part of the query restricts those tuples whose domain is in the Task set.

<table>
<thead>
<tr>
<th>A*</th>
<th>Transitive closure of the relation A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A\S</td>
<td>Restrict the domain of the relation A to the set S</td>
</tr>
<tr>
<td>A o B</td>
<td>Composition of relations A and B</td>
</tr>
<tr>
<td>A U B</td>
<td>Union of two relations/sets</td>
</tr>
</tbody>
</table>

Figure 7-9: Definition of RPA notations.
7.3 Reverse Architecting of the CARA

7.3.2.3 Structure of task creation

Storing the runtime structures as relations enabled the analyst to visualize them interactively. For example, the analyst loaded the TT\textsubscript{create} relation into the Prefuse visualization tool and found that all tasks are created by the CARA\textsubscript{Main} task. By querying the TT\textsubscript{create} relation, the analyst found that the CARA has eleven tasks or runtime components, see for a snippet of the task creation structure.

\[
\begin{align*}
TF\textsubscript{use} &= (TF\textsubscript{enter} \cup \text{Call})^* \downarrow \text{Task} \quad (1) \\
TG\textsubscript{use} &= TF\textsubscript{use} \circ FG\textsubscript{use} \quad (2) \\
TM\textsubscript{enter} &= TF\textsubscript{enter} \circ \text{Partof} \quad (3) \\
TM\textsubscript{use} &= (TF\textsubscript{use} \cup TG\textsubscript{use}) \circ \text{Partof} \quad (4) \\
TT\textsubscript{create} &= TF\textsubscript{use} \circ FT\textsubscript{create} \quad (5) \\
TT\textsubscript{delete} &= TF\textsubscript{use} \circ FT\textsubscript{delete} \quad (6) \\
TT\textsubscript{restart} &= TF\textsubscript{use} \circ FT\textsubscript{restart} \quad (7) \\
TQ\textsubscript{create} &= TF\textsubscript{use} \circ FQ\textsubscript{create} \quad (8) \\
TQ\textsubscript{delete} &= TF\textsubscript{use} \circ FQ\textsubscript{delete} \quad (9) \\
TQ\textsubscript{read} &= TF\textsubscript{use} \circ FQ\textsubscript{read} \quad (10) \\
TQ\textsubscript{write} &= TF\textsubscript{use} \circ FQ\textsubscript{write} \quad (11) \\
TS\textsubscript{create} &= TF\textsubscript{use} \circ FS\textsubscript{create} \quad (12) \\
TS\textsubscript{delete} &= TF\textsubscript{use} \circ FS\textsubscript{delete} \quad (13) \\
TS\textsubscript{take} &= TF\textsubscript{use} \circ FS\textsubscript{take} \quad (14) \\
TS\textsubscript{give} &= TF\textsubscript{use} \circ FS\textsubscript{give} \quad (15)
\end{align*}
\]

Figure 7-10: RPA queries for derived relations of Table 7-6.

Figure 7-11: (Snippet) Task creation structure (i.e., TT\textsubscript{create}).
7.3.2.4 Structure of common code and task-specific code

The analyst queried and visualized the derived relations $TF_{use}$ and $TG_{use}$ in order to identify those functions and global variables that are shared among tasks or used only by one task. Similarly, the analyst used the $TM_{use}$ relation and visualized how source code files are shared among tasks or used only be one task. A snippet of how files are partitioned among tasks is shown in Figure 7-12. Such partition structures show which files are task specific and which ones are shared among tasks. These structures gave remarkable insights regarding how the static source code elements are transformed into runtime components. For example, by querying and visualizing the $TM_{use}$ relation, the analyst found that there is an 11:1 usage mapping from eleven tasks to one C file called `cara_interface.c`. That is, all eleven tasks use this file. The analyst visualized the $TM_{enter}$ relation and found that the entry functions of seven of the ten tasks that are started by the main task are defined in `cara_interface.c`. The entry function of the main task is defined in `cara_main.c`, and the entry functions for the remaining three tasks are defined in three different files, see Figure 7-13.

![Figure 7-12: (Snippet) Partition of files among tasks.](image)

7.3.2.5 Testability and partition of files among tasks

One consequence of defining seven entry points of seven tasks in one file (i.e., `cara_interface.c`, Figure 7-13) is that changes to code elements of this file require recompilation and retesting of all tasks using the newly compiled binary code. This time could be reduced if the entry functions as well as functions uniquely used by the entry functions are moved to separate files. Only the tasks that are affected due to recompilation need to be unit tested again. Because the current partition structure does not allow independent compilation of task-specific code from other tasks, it is impossible to produce a separate executable for each task to facilitate unit testing. This partition structure as such is not amenable to task-by-task verification of safety properties and difficult, if not impossible, to unit test and verify.
7.3 Reverse Architecting of the CARA

7.3.2.6 Structure of inter-task communication

After understanding how the source code elements are partitioned among tasks, the next step was to discover how the tasks communicate with each other. The analyst visualized the inter-task communication structure using the derived relations \( TQ_{\text{read}} \) and \( TQ_{\text{write}} \) that show which tasks read from queues and which tasks write to queues.

The inter-task communication structure offered several insights including:

- **the CARA has five queues,**
- **I/O bound functionalities, such as displays service (Display_Svc and CUII_Svc in Figure 7-14), and the logging service (Log_Svc in Figure 7-14), are assigned to separate tasks as recommended in software design methods [119],**
- **some of the tasks, for example, Alarm_Svc in Figure 7-14, do not use queues to communicate with other tasks but instead use shared global variables,**
- **some of the tasks only read messages from queues and do not write messages to queues (e.g., Log_Svc),**
- **there is one central queue (e.g., CARA_MSGQ) used by several tasks,**
- **there is no connector abstraction, i.e., all tasks are responsible for reading from / writing to queues using standard VxWorks APIs as well as handling low-level errors, such as unable to write to or read from a queue.**

Using queues to communicate between tasks result in a complex network because a queue is a binary connector between two tasks. For the same message (e.g., shutdown) to be sent from a task to all other tasks would require a one-to-many queue connection. In the NASA CFS and NASA GMSEC cases, a software bus (SB) is used to exchange messages between tasks by applying the publish-subscribe paradigm [97] and [101]. The SB takes care of low-level communication and synchronization concerns. Thus, tasks can focus on computation instead of coordination.
Creation and deletion of queues. The analyst visualized the derived relation $TQ_{create}$ and found that all queues are created by the $CARA_{Main}$ task. All queue identifiers are global variables and therefore all tasks are able to use queues. The $TQ_{delete}$ relation showed that all tasks are deleted by the $CARA_{Main}$ task.

Semaphores. By visualizing the derived relation $TS_{create}$ the analyst found that all semaphore identifiers are created by the $CARA_{Main}$ task as global variables. By visualizing the $TS_{take}$ and $TS_{give}$ relations, the analyst found that the semaphore give and take concepts [66] are scattered across all tasks. Such concepts that are needed for all tasks could be abstracted into a common set of services. Ideally, each task is not aware of the fact that semaphores are used for synchronization.

7.3.2.7 Testability and inter-task communication structure

The analyst used the discovered inter-task communication structure to assess the difficulty of testing these tasks. It is a fact that the higher the number of queues the more difficult it is to test the system. For example, if a task reads input messages from only one queue (e.g., $Log\_Svc$ of Figure 7-14), then a small test program can be developed to write messages on the other end of the queue. On the other hand, a task similar to the $CARA_{Main}$ reads messages that are written by several tasks into the input queue. Thus, testing the $CARA_{Main}$ is relatively difficult because several different types of messages have to be written into the input queue in order for the test to be realistic. In addition, permutations of inter-leaving of messages need to be simulated and tested. Because the semaphore and the queue concepts are not abstracted, unit testing of each task would require initializing the semaphores and queues as well,
even if only one task is running at a time during unit testing. In addition, some of the CARA tasks share global variables. Such communication structures require significant testing in order to make sure the state space of the global variables is properly understood by all tasks using them. Based on this analysis, the analyst concludes that the implemented architecture is not unit-test friendly, if not impossible to unit test.

7.4 Verification using Extracted Structures

The findings from reverse architecting of the CARA system were used to aid in static analysis of the source code. We used a commercial grade static analysis tool to analyze the CARA, first using the default configuration, and then with annotations garnered through reverse architecting.

In this section, we discuss how the discovered implemented architecture was used for: a) configuring the static analysis tool, b) adapting the source code of the CARA for improved performance in terms of code coverage, and c) interpreting the warnings generated by the static analysis tool.

Configuring the static analysis tool using the extracted structures. An analyst at the FDA’s OSEL laboratory independently ran the static analysis tool on the CARA. The analyst had no prior knowledge of the implemented architecture of the CARA. The result was that the static analysis tool reported that it cannot run due to missing VxWorks OS files. At this point, the results of the architectural analysis came to light. The CARA has a configuration point that allows it to compile and run under Windows. We used that extracted architectural knowledge to configure the static analysis tool and resolved the problem of missing files because the missing files were no longer needed under the selected configuration. Thus, the static analysis tool was able to analyze the CARA using Windows as the OS configuration when compiling the source code.

Using the extracted structures for adapting the source code. The static analysis tool reported that the CARA has several dead functions that cannot be verified for safety properties. By visualizing the extracted runtime structures, it became clear that those dead functions are entry functions to different tasks. The entry functions are intended to be indirectly called by the OS’s task spawning function. Analysis of the stub code of the task spawn function generated by the static analysis tool revealed a surprising fact that the stub did not “understand” the meaning of the task spawn function and failed to activate all entry functions to tasks. As a result, entry functions were reported wrongly as dead code by the static analysis tool. This is a weakness of the tool and not the source code of the medical device. In order to overcome this problem, the analyst defined his own stub implementation of the task spawn function that activated the entry function.

Because the CARA uses queues for communication among tasks, the analyst checked how the static analysis tool handles functions related to queues. The static analysis tool did not “understand” the meaning of OS functions used for sending messages to queues and receiving messages from queues. Thus, the static analysis tool was not able to handle the indirection involved in sending/receiving messages. The analyst had to adapt the generated stub functions, making them closer to the reality.
Interpreting the warnings generated by the static analysis tool using the extracted structures. The static analysis tool reported that the CARA has a redundant predicate `if (!exiting)`. The implemented runtime structure, stored in the `TGuse` relation, showed that `exiting` is a global variable shared by all the eleven tasks, see Figure 7-15. This runtime structure helped the analyst to refute the warning because it turned out that the static analysis tool was not able to handle concurrency aspects (e.g., semaphore, mutual exclusion) well. Also, even if `exiting` was never non-zero, it is prudent to have redundant checks for safety reasons to ensure that critical operations are performed only if the system is not exiting.

![Figure 7-15: (Snippet) Tasks and global variables usage.](image)

Thus, if the analyst does not know the implemented architecture of the CARA, then there is a risk that the static analysis tool results lead the analyst to the wrong conclusions about the safety of the medical device. Bugs can be lurking in the code due to limitations of static analysis tools.

### 7.5 Discussion

**Recommendations to device manufacturers.** This study has shown that architecture analysis can shed light on quality issues, especially testability and verifiability properties that can translate into safety issues. Based on this research, testability and verifiability properties should be addressed in the construction of a device (safety) assurance case. Arguments of the assurance case might refer to static and runtime structures, including how the implemented software handles OS variants, COTS software interactions, partitioning of the source code elements into runtime tasks, and how tasks communicate and synchronize with each other. Approaches such as those discussed can provide the evidence needed to justify the arguments.

**Recommendations to static software analysis tools users.** This study identified several issues related to using a static analysis tool in the absence of knowledge about the implemented architecture of the system under verification. In order to avoid misleading results, static analysis tool users must pay significant attention to understanding how the static analysis tool handles multitasking as well as OS and library calls. Using the extracted runtime structures we learned the importance of mock implementations of analog/digital converter APIs’ to facilitate verification of safety properties of tasks that interact with hardware boards. Otherwise, there is a high risk for incorrect conclusions about the software safety properties believed to be verified by the static analysis tool. Thus, in order to leverage the full power of static analysis tools, the
7.6 Closing Remarks

user needs to be aware of architectural structures of the software under analysis so that static analysis tools are configured appropriately.

**Prefering for modular verification.** Verifying safety properties using static analysis tools takes time. Many static analysis tools allow running verification as a collection of parallel jobs for improved performance. However, static analysis tool users must identify modules of the system under verification. Using the runtime structure of the CARA, the OSEL analyst identified how to run the static analysis tool in a modular fashion. The OSEL analyst defined a verification job for code unique to each task. The runtime structure helped the analyst plan for modular verification. At the time of finalizing this thesis, work was underway to measure response time of verifying several jobs in parallel against verifying the CARA as one job.

**Generalizability of the proposed approach.** In order to replicate this case study on other systems, we believe the reader can benefit from the step-by-step description of how we did reverse architecting of software structures including the static and runtime structures. Using the CARA, this chapter discussed a set of abstract, yet precise, structures that are used to systematically reason about software testability. Medical devices often handle OS variants, hardware interaction, GUI, COTS software interactions, and multitasking concerns. Thus, the analysis steps outlined in this chapter are repeatable in other devices.

The formulas for statically extracting runtime structures defined in Figure 7-10 are independent of the CARA, and therefore are reusable for other medical devices that are based on task-oriented architectures. A knowledge base of external libraries are of significant value to first get a high-level summary of architectural features in addition to a parser that extracts code relations. In this case we did not have the need for performing data flow analysis. For other systems, such analysis might be needed. A query language similar to the RPA facilitates reverse architecting significantly because questions related to software architectural properties are quickly answered by using a suite of relational queries. A graph visualization tool is also important to visualize software structures and recognize patterns visually.

**7.6 Closing Remarks**

We investigated the benefits of incorporating architectural analysis of implemented systems for safety analysis of a medical device. We reconstructed a catalog of static and runtime structures from the system’s source code. We then did a rigorous and detailed analysis by extracting one structure at a time, ignoring irrelevant details with respect to the selected structure. Evidence that architectural analysis can offer useful insights on testability and verifiability, from which we can form an assurance case for safety was collected. We discussed how architectural analysis was integrated with static analysis tools to verify safety properties. We have not extracted and analyzed all structures related to testability and safety. For example, our runtime structures did not address interrupt handling and real-time watchdog behaviors of the CARA. Specific analysis questions are a) how easy is it to trigger interrupts and test their handling capability? b) how easy is it to test real-time features such as the behavior of the watchdog, if some functions do not reply within expected time constraints? In addition, we have not analyzed the testability of safety properties, for example, how the CARA
handles unplugging the infusion pump. Future research will address the extraction of those runtime structures. Our future prospect is to collect a catalog of structures to reason about testability and safety of medical devices at the architecture level.
Chapter 8
Discovering Organizational Aspects for Migration to a SPL\textsuperscript{13}

8.1 Abstract

How to introduce software product line engineering (PLE) in the presence of existing stand-alone similar systems remains a challenging question for many organizations. This chapter reports on a reverse engineering approach to understanding organizational aspects during the product line planning phase. The organizational aspects include domain expert identification, understanding the organization's development or team structure, and predicting the existing product architecture using the organization's architecture. In addition, this chapter highlights how the source code change history log provides valuable data for various product line related activities, such as scoping, architecture evaluation, reengineering towards product line and project management in the product line context. The proposed approach is discussed using the engine control systems of Hitachi.

**Keywords:** Organizational Aspects, Software Product Line, Ownership Architecture, and Commonality Analysis.

8.2 Introduction

Deploying a product line in the presence of similar existing stand-alone systems remains a non-trivial task for many organizations. One possibility is to reengineer the existing systems into a product line [75]. This means that a common architecture (or reference architecture) should be defined by considering the commonalities and variability among the present and envisioned products. Before performing the actual reengineering, organizations need to identify where to invest for reuse so that the economic benefits of PLE are visible and significant [27], [106], [293], [294], and [292]. After that, the selected, existing components shall be consolidated and adapted into reusable components. However, organizing such reengineering projects is a challenge; a major task thereby is to assign the right people to the right tasks. In this chapter, we define an approach for the identification of product experts by mainly using the source code change history log when migrating towards product line engineering.

PLE usually comprises two development stages, namely, development for reuse (or family engineering), and development with reuse (or application engineering). Before introducing PLE into practice, an organization should carefully plan family and

---

\textsuperscript{13} Appeared in IEEE Working Conference on Reverse Engineering (WCRE) 2006. IEEE Computer Society [107].
application engineering. That is, they should form an appropriate group or team structure so that family and application engineering are done by the right people with the right skills with clear roles and responsibilities.

The existing literature [18], [135], [304] advises that organizational factors should be given significant importance for the success of reuse. It also warns us that organizational factors can hinder the benefits of reuse, if not carefully addressed. In [28], it is suggested that “The adoption of a product line engineering approach starts with the assessment of the current state”. The assessment of current state include: organizational stability, maturity, staff turnover, domain expertise, and project management maturity. On the other hand, to our knowledge, there are no supporting guidelines to assess the current state. This chapter addresses this open issue in order to support the planning phase of PLE.

In this chapter, we aim to understand the current development style of the existing products. The basic principle is to introduce PLE with minimal impact on an organization’s way of working. To this end, we seek answers to the following questions:

- What is the development style followed by the organization? That is, is there a static team structure -fixed component team from one release to another - or a dynamic team structure?
- Who has domain knowledge in which parts of the existing products? Or who owns which parts of the existing products?
- Who are the senior developers of the existing products and are they still working in the business unit or organization?
- What is the commonality among the developers? That is, are the existing systems or components engineered by the same set of developers?

By interviewing the managers of the existing products, one may be able to obtain partial answers to the above questions. However, this requires additional effort and the answers may not be correct and might not be obtainable fast. Moreover asking many questions is also not pleasant during technology transfer activities. Also, the managers might have moved from one business unit to another unit due to organizational restructuring, and hence they may have limited knowledge of the history of the development team. So, a more practical approach is needed to better understand the current development style, without raising many questions to the managers, and eventually minimizing the effort of management involvement.

As shown in the latter part of this chapter, one promising possibility is to analyze the historical relationship between the developers and the source files they implemented or changed. This relationship, hereafter called ownership architecture, is usually stored by configuration management systems such as CVS. The key idea is to investigate this relationship at different levels of abstraction (e.g., system level, subsystem level, component level). To raise the level of abstraction of the relationship stored in the configuration management systems, the proposed approach uses the directory structure decomposition and the knowledge of architects to define a meaningful mapping between directory structure and the abstract components. If the architects are not available, then the level of abstraction is until directory structure decomposition.
Merging similar systems into a product line goes beyond the technical aspects of reengineering. We have to bring people together to perform reengineering. To better understand this critical aspect, we also discuss commonalities among the developers of existing products by introducing the metrics called Developer Containment Ratio (DConR) and Developer Commonality Ratio (DComR). The DConR metric is used to answer questions such as: what percentage of developers of product A is contained in product B? The DComR metric is used to answer the question: what percentage of developers is shared between product A and B? DConR and DComR metrics are simple, easy-to-calculate, and are applied at different levels of abstraction: system level, subsystem, component, and file level.

The main contributions of this chapter are:

- A simple process for reconstructing ownership architecture.
- Outlining the benefits of ownership architecture in the product line planning phase.
- Simple metrics (DConR and DComR) for comparing the ownership architecture of existing products.
- Analysis of the approach using an industrial case study from the automotive domain.

The rest of this chapter is structured as follows: Section 8.3 compares this chapter with the related work. An overview of product line engineering is the subject of Section 8.4. The approach to ownership architecture recovery is the subject of Section 8.5. A case study from Hitachi engine control systems is used to demonstrate the applicability of the approach in Section 8.6. The conclusion of this chapter is given in Section 8.7.

8.3 Related Work

The presented work is related to research in the area of organizational aspects of software product lines, ownership, and software architecture reconstruction.

Different organizational models for realizing organization-wide reuse programs are introduced in [32] and [74]. In [56], project management and organizational structure in the context of PLE is discussed. Our work complements these existing works by extracting the current organizational style, and thus can be used as a baseline for adopting different organizational models proposed by them.

In [34], a method for reconstructing ownership architecture is introduced. The difference, however, lies in the context: our motivation for reconstruction is for product line planning purposes, whereas their motivation is to mainly support reengineering. We introduce metrics for comparing the ownership architecture of existing products.

Ownership Map visualization to understand the relationship between files and developers is introduced in [116]. They identify different behavioral patterns of the developers using the Ownership Map. The level of abstraction of the Ownership Map is, however, confined to the file level, in contrast to our approach, which supports the hierarchy of abstractions. Their visualization concepts are appealing, and we are
exploring them further (part of future work). In [204], an expertise browser to identify experts for given elements of a software project is introduced. Our approach uses a textual database of the relationship between files and developers. Users can query the database, for example, using relation partition algebra (RPA), to identify experts at different levels of abstraction (e.g., subsystems, components, files). Their approach uses visual representation, and the intended application is to mainly support coordination of multisite projects. Also, the expertise browser is limited to a single project; in contrast, we setup the expertise browser for all existing related projects.

In [210] and [169], an architecture reconstruction and verification method to compare the specified architecture with its implementation is proposed. The mapping step we followed corresponds to their approach. In [169], RPA is used for the purpose of software architecture reconstruction; we have applied RPA for the purpose of ownership architecture reconstruction – an additional complimentary application of RPA. However, other query languages (e.g., Grok [132], SQL) might also be applicable for ownership architecture reconstruction.

In [172], an approach for recovering management information from source code is proposed. They employ source code analysis techniques in order to measure IT risks and propose risk mitigation strategies. Their approach provides various managerial insights into the IT-portfolio management, such as maintenance of almost 60% of the portfolio source modules depends on expensive CASE tools, almost 40% of the modules rely on the no longer supported IBM OS/VS COBOL compiler. In contrast to their approach, our approach focuses on extracting management information in the context of migration to a software product line.

8.4 Fraunhofer PuLSE™

As early as 1997, Fraunhofer IESE started to focus on product line engineering as an approach to improve productivity and quality while efficiently producing many product variants. The gained experience and solutions together quickly evolved towards an integrated approach of its own: Fraunhofer PuLSE™ (Product Line Software Engineering) [18].
Reconstructing Ownership Architecture

In this section, the approach we followed for reconstruction of the ownership architecture is explained.

8.5.1 Reconstruction Process

We used the relation partition algebra (RPA) formalism to explain some of the steps in the reconstruction process. RPA notations and operators are well explained in [169]. However, informal explanations are also provided here for readers' convenience.
The reconstruction process (see Figure 8-2) contains the following five steps. First, we give an overview of the reconstruction process before going into the details of each step:

1. Derive an architecture decomposition of an existing product from its architect.
2. Define a mapping between architecture and source code structure.
3. Extract the developer-file relation from the source code change log.
4. Build an abstraction using the mapping and the developer-file relation.
5. Present the abstracted architecture for analysis.

**Step 1 - Derive Architecture Decomposition:** In this step, using the existing documentation and the architects of the current product, we derive the architecture decomposition. Basically, architecture decomposition captures the structure of the software, and the dependencies among abstract components.

**Step 2 – Mapping:** Because architecture is an abstract entity and cannot be found easily in the implementations, we define a mapping between abstract components and the source code structure (e.g., directory structure) with the help of the architects. Formally, mapping is nothing but a binary relation with the domain as architecture components and the range as directory structure. For example, consider the layered architecture decomposition shown in the left part of Figure 8-3.
Each of the layers is mapped to one or more directories (e.g., the top layer is mapped to Dir1, Dir2, and Dir3). We can model this mapping mathematically as follows:

$$\text{Mapping} = \{\langle \text{Top}, \text{Dir1} \rangle, \langle \text{Top}, \text{Dir2} \rangle, \langle \text{Top}, \text{Dir3} \rangle, \langle \text{Middle}, \text{Dir4} \rangle, \langle \text{Bottom}, \text{Dir5} \rangle\}$$

To define such mappings, it is necessary to have a good overview of the implementation structure. To solve that issue, we use the expertise of software architects. However, this step is usually iterative as mentioned in [210].

**Step 3 – Extraction:** In this step, we extract the relationship between the developers and the files they have committed (implemented/changed). In a large product development environment, configuration management systems (e.g., CVS) are usually used to control concurrent developments. These systems track the change history and store the names of developers working on different files.

More formally, using the change log, we extract a binary relation called Developer_File, with the domain as developers and the range as the files they committed to the CVS. That is,

$$\text{Developer } _\text{File} = \{\langle x, f \rangle \mid x \text{ committed file } f\}$$

One can also query the extracted Developer_File relation to derive basic knowledge about an existing product. For example, in order to know how many developers were involved in an existing product development, the RPA query is $|\text{dom(Developer_File)}|$, where $\text{dom}$ of a relation denotes the domain of the relation, and $|S|$ denotes the number of elements in a set $S$. Note that we do not count the number of commits made by a developer in each file because our aim is to identify experts at much higher level of abstraction. We consider developers as experts of a directory if they worked in many files within the directory. This definition applies recursively to the further levels of abstraction. This simple definition seems to be good enough, as will be discussed in the case study section.

**Step 4 – Abstraction:** A limitation of the Developer_File relation is the level of abstraction – too finely grained at the file level. Therefore, we build an abstraction
using the Mapping (see Step 2) and Developer_File relations. Fundamental operators of the Set theory, such as composition, and inverse, are employed to build abstraction as shown below.

\[ Part\_of = \{ (f, d) | f \text{ is a file in directory } d \} \]

To lift the Developer_File relation to directory level, we use the lift range operator of RPA as follows:

\[
\text{Developer.Dir} = \text{Developer.File}_{ran}\text{Part.of}
\]

Informally, the Developer.Dir relation captures the relationship between developers and the directories they have committed. Currently, we define a developer as the owner of directory if the developer has worked on many files (relative to other developers) within the directory.

To further raise the level of abstraction to the component level, we use the Mapping defined in Step 2 and the lift range operator of RPA as follows:

\[
\text{Developer.Component} = \text{Developer.File}_{ran}\text{Mapping}
\]

The Developer_Component relation captures the relationship between the developers and the components they have committed. The outputs of this abstraction step are the relations Developer.Dir and Developer_Component.

**Step 5 – Presentation:** In this last step, we present the abstracted relation between the developers and the components they have worked on. We use both the graph and table representation to show the ownership architecture. Sometimes table is better than graph when there are many nodes and edges to visualize. Figure 8-4 shows an example of the ownership architecture using the SAVE tool. In addition, we also present a developer distribution graph that shows the number of developers per component and how many developers are shared between components. More details will be discussed using the Hitachi’s ECS in case study section.

![Figure 8-4: Ownership architecture – An example.](image-url)
8.5.2 Expert Identification

System level experts are those developers with a good overview of the overall existing implementations, since they have worked on different components of the system. To identify system level experts by using the already extracted Developer_Component relation, we executed the following RPA expression:

\[
\text{dom}\left(\text{Developer \_ Component}\right)
\]

Component level experts are developers with detailed knowledge about the component internals. Since components are made up of one or more files, we consider those developers who worked on many files within a component as potential experts. The following RPA expression calculates, by using the already extracted Developer_File and Part_of relations, within a given component \(K\), the number of files that were committed by each developer.

\[
\text{dom}\left(\text{Developer \_ File \mid (Part \_ of \ K)}\right)
\]

8.5.3 Developer Commonality Analysis

In this subsection, we formalize the metrics Developer Containment Ratio (\(DConR\)) and Developer Commonality Ratio (\(DComR\)).

Let \(K\) be a component, and \(D^A(K)\) and \(D^B(K)\) denote the set of developers involved in the implementation of the component \(K\) in product \(A\) and \(B\), respectively. Then we define \(DConR\) as follows:

\[
DConR^{AB}(K) = \frac{|D^A(K) \cap D^B(K)|}{|D^A(K)|}
\]

Note that \(0 \leq DConR \leq 1\). If the value of \(DConR(K)\) is towards 1, then this implies that almost all the developers of component \(K\) of product \(A\) are also the developers of component \(K\) of product \(B\). If the value is towards 0, then this is vice-versa.

One limitation of the \(DConR\) metric is that it only captures the developer containment from product \(A\) to \(B\). For example, if all the developers of product \(A\) are also involved in \(B\), but product \(B\) can also have a new set of developers, who are not in \(A\), then this situation is not clear in the \(DConR\) metric. For that purpose, we define development commonality ratio \(DComR\) as follows:

\[
DComR^{AB}(K) = \frac{|D^A(K) \cap D^B(K)|}{|D^A(K) \cup D^B(K)|}
\]

Note that \(0 \leq DComR \leq 1\). The value of \(DComR(K)\) are used to understand which percentage of developers is shared in the implementation of component \(K\) in product \(A\).
and B. If $DComR(K)$ is towards 1, then component $K$ is almost implemented by the same set of developers in both the products.

In the next section, we apply the approach to Hitachi’s engine control systems (ECS).

8.6 Case Study

8.6.1 Background

Hitachi releases – among various other products - many variants of engine control systems (ECS). From a domain point of view, these variants share significant commonalities among them. With the increase of the ECS business, the types of ECS variations (see Table 8-1) has increased dramatically and software development costs are increasing at an alarming rate. This situation and the business goals of Hitachi ECS match the scenarios for introducing product line engineering.

Table 8-1: An example of ECS variants

<table>
<thead>
<tr>
<th>Customer</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine Type</td>
<td>Direct Injection</td>
<td>Multi Port Injection</td>
</tr>
<tr>
<td>Engine Size</td>
<td>1.6</td>
<td>2.0</td>
</tr>
<tr>
<td>Market</td>
<td>Japan</td>
<td>X</td>
</tr>
<tr>
<td>U.S.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Europe</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

To support and prepare for a systematic product line migration, we researched on the following aspects: (a) recovery of implemented architecture from the existing products, (b) assessment of the product line characteristics of current products, and (c) assessment of the organizational and economic effects of a product line. The results of (a) and (b) can be found in [315] and [316], respectively. The economic effects of a product line on the ECS business were already investigated in [106]. The focus of this chapter is confined to the organizational effect of a product line.

The remaining chapter is organized as follows: First, a few important objectives of the case study are listed. Second, a brief overview of ECS is given. Third, the ECS ownership architecture recovery process is given. Then, different analyses on the recovered ownership architecture are provided. Next, a comparison of the ownership architecture of two ECS versions is discussed.
8.6.2 Objectives of the Case Study

The main objectives of this case study include the following:

- Identify who worked on which part of the ECS. This knowledge is useful for organizing the interviews and asking the right questions to the right people, amongst other purposes.

- Identify the owners/experts at different levels of abstraction (e.g., subsystem, component level).

- Identify the current development style. This knowledge directs us to define a potential organizational structure for product line engineering.

- Compare the ownership architecture of two ECS variants to know whether the existing components are implemented by the same set of developers. This knowledge will help us to bring the developers together to merge the existing components into product line components.

8.6.3 Overview of ECS

ECS (see Figure 8-5) is one of the core components of Hitachi’s engine management systems for automobiles. ECS observes engine status and driver requests and controls the engine by operating the amount of fuel injection, the timing of the ignition, the quantities of intake air, and so on.

![Figure 8-5: Overview of ECS.](image)

The management of variations is a key issue in ECS. Hitachi has many global customers, each of whom has their specific requirements and wants to use the same ECS for different models. Moreover, there are market-specific regulations, too. To protect our environment from air pollution and global warming, increasingly stringent automotive regulations have been issued year after year, such as reductions in gas emissions and improvements in fuel efficiency. Such regulations - as well as many other requirements - vary among products and markets, such as Japan, the United States, or Europe. Not exclusively but also due to this development, the size of flash memory in a typical Hitachi ECS continuously increases as depicted in Figure 8-6, for the past 12 years or so, by a large development crew size that is difficult to expose because of the confidential reasons.
To avoid an increase in development costs, similar to the pattern of flash memory size, reuse of software for ECS systems has become a key issue. To tackle variability and reusability issues, Hitachi has started to concentrate on product line engineering for ECS, and thus started a joint project with Fraunhofer IESE.

### 8.6.4 Ownership Architecture Recovery of ECS

In this section, we explain the application of the ownership architecture reconstruction process to one of the ECS variants.

**Step 1 - Architecture Derivation:** In the first step of reconstruction, we derived the conceptual architecture of ECS from an expert who is familiar with the engine control domain and the structure of ECS. The resulting architecture is shown in Figure 8-7.

**Step 2 - Mapping:** An ECS expert defined a mapping between the abstract architecture components and the directory structure of ECS. That is, the abstract entities shown in Figure 8-7 are mapped to the directory structure of ECS (as in the reflexion model [210] and [211]).
Step 3 - Extraction: We wrote a few Perl scripts that parse the change log in each file and extract developers who worked in a file.

Step 4 - Abstraction: In this step, we abstract the extracted relation between developers and files to the architecture level, and stored the results in a tabular format. Basically, the abstracted relation captures the developers who worked on a component.

Step 5 - Presentation: The analysis is explained in the next sections.

8.6.5 Analysis

The analyses that were performed on the recovered ownership architecture of ECS include the following:

- Predicting Current Organizational Style.
- People Risk Assessment.
- Implemented vs. Ownership Architectures.
- (Developer) Commonality Analysis of two ECS variants.

Predicting Current Organizational Style: We aimed at predicting the current organizational style using the recovered ownership architecture. That is, is there any fixed team structure or some kind of component groups? For that purpose, we investigated the ownership architecture for potential patterns in the developers’ distribution. In Figure 8-8, a node denotes a component, and a directed edge from node X to Y denotes the percentage of developers of X contained in Y. A loop in a node denotes the percentage of developers who worked on a component with respect to the complete system. For example, the Sensor_Actuator component has been implemented or changed by 75% of the ECS developers and 74% of them worked in the Application component.

Figure 8-7: Conceptual architecture of ECS.
There are a few interesting patterns that became visible: A large portion of the ECS developers worked on the Sensor_Actuator and Application component. This, in fact, matches the fundamentals of ECS; both components are important core components of ECS, responsible for interfaces with external sensors and engine control functions, respectively. Another pattern: almost all the ECS developers who worked on the Application component also worked on the Sensor_Actuator component.

In summary, from the ownership architecture we found out that almost everybody gets a chance to work in every component of ECS. We discussed these patterns with the ECS experts and it later became clear that the current organization style is "dynamic". That is, the ECS business unit tries to maximize the productivity of its developers. Therefore, whenever there is a new release or major functional updates, a new "dynamic" team of developers is formed for the implementation. In other words, there is no rigid static team structure or component group within the current ECS implementations.

**People Risk Assessment:** When migrating an existing software to a product line, it is wise to understand the risks associated with a lack of expert developers who performed the existing implementations. For analyzing such kinds of risks, we used the recovered ownership architecture and the corporate database of the current employees. We found that most of the system and component level experts are still part of the organization, which is good news for planning the product line introduction in a real-time embedded domain, where the domain knowledge is important.
**Implemented vs. Ownership Architecture:** It was our initial assumption that the implemented architecture of ECS will reflect its ownership architecture. To validate that assumption, we recovered the implemented architecture of ECS (see Figure 8-9) using the SAVE tool of Fraunhofer IESE.

![Implemented software architecture](image)

Figure 8-9: Implemented software architecture.

By comparing Figure 8-8 with Figure 8-9, we noticed that the ownership architecture seems to be a reasonable predictor of the implemented architecture. That is, whenever the developers are shared between components, there is a dependency at the implementation level. It looks like one can already predict the implemented architecture by using the developer distribution graph. However, more study is needed to generalize this claim. We feel that the distribution of developers can influence the quality of the implemented architecture; in this case, almost every developer gets to work on every component, and every component is connected to every other component at the code level.

### 8.6.6 Developer Commonality Analysis

As mentioned earlier, Hitachi releases many variants of ECS. It is, however, not clear which developer contributes to which variants, whether the existing components of the variants are implemented by the same group of developers or whether they are a completely different group. The key message is that to efficiently merge the existing variants into a product line, we should bring together those developers who implemented the existing components. To obtain a quick insight into this aspect, we computed the DConR and DComR metrics, defined in Section 4.3, at different levels of abstraction, for the two ECS variants A and B. At the system level, DConR was 98%, which means that nearly all the developers of product A were also involved in implementing product B. We also computed the DConR metric at the component level,
to support merging the existing components. For the large components of ECS, nearly 80% of developers of product A are also involved in the corresponding component of product B (see Table 8-2).

While DConR was useful for knowing the containment of developers from one product to another, it doesn't capture the information about the developers who are in one product, but not in another product. For that purpose, we computed the DComR metrics, starting from the system level to the component level. At the system level, DComR was 60%, which means that 60% of the developers are involved in implementing both products A and B. Also, at the component level, the DComR metric indicates that the new developers are involved in the implementation of component of product B. It is worth noting that in the change history log, we were able to distinguish logs of product A from product B because of different history templates used in the code.

<table>
<thead>
<tr>
<th>Component</th>
<th>DConR</th>
<th>DComR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor_Actuator</td>
<td>0.88</td>
<td>0.41</td>
</tr>
<tr>
<td>System_Service</td>
<td>0.85</td>
<td>0.4</td>
</tr>
<tr>
<td>Application</td>
<td>0.83</td>
<td>0.47</td>
</tr>
<tr>
<td>Complex_IO_Driver</td>
<td>0.8</td>
<td>0.49</td>
</tr>
<tr>
<td>Memory_Service</td>
<td>0.63</td>
<td>0.32</td>
</tr>
<tr>
<td>Communication_Driver</td>
<td>0.62</td>
<td>0.39</td>
</tr>
<tr>
<td>IO_Driver</td>
<td>0.5</td>
<td>0.2</td>
</tr>
</tbody>
</table>

The key message is that by computing DConR and DComR at different levels of abstraction, we get good insights into the organization of existing variants. This knowledge is something valuable for product line planning, because we have a good overview of the current status of the organization. Also, managers can already start reasoning about how to manage the existing human resources for the future. There is no need to have two separate groups implementing conceptually similar components. DConR and DComR point us to this valuable information in advance.

### 8.6.7 Manager’s Feedback

In order to validate the results and get confidence about the experts predicted by our approach, we requested a ECS manager to review the top 10 system and component level experts. Almost all the candidates proposed by our approach matched well with the personal opinion of the manager. Some component level experts were not system level experts, and the reverse was also true. It looks as though a simple definition of the component expert - who committed many files - and the system expert - who worked on different components - seems to be sufficient to predict the system and component level experts.
8.6.8 Benefits of Ownership Architecture

This section discusses some of the important applications of ownership architecture in the product line planning phase of ECS.

**Scoping:** Before an organization introduces product line engineering (PLE) into practice, it has to carefully investigate which products will (or will not) be part of a product line, and which components will be developed for reuse. This activity is called scoping. The main goal of scoping is to identify where to invest for reuse to obtain maximum benefit. In our experience, scoping is a critical activity in the planning phase of PLE [143]. Usually, workshops or interviews with domain experts or expert developers are necessary to define the scope of a product line. We will use the ownership architecture of ECS products to invite the right people for participating in the workshop. As a consequence, the workshop will hopefully be productive and contain the right participants.

**Architecture Evaluation:** Software architecture is the key artifact for the success of PLE, because it is used to derive a family of related products. So, before introducing PLE, a common architecture has to be defined. Furthermore, the architecture has to be evaluated to find trade-offs of quality properties. Similar to scoping, workshops with domain experts are necessary to evaluate the architecture. Engine control systems are real-time embedded control systems. Architecting and evaluating such systems require good understanding of the ECS fundamentals. The ownership architecture facilitates the preparation of an evaluation workshop by showing the owners of the existing components.

**Project Management:** Before introducing PLE an organization has to define an organization or group structure that facilitates family and application engineering. This means that organizations have to identify the right people with the right skills. Moreover, project management activities become more challenging if organizations have to introduce a product line in the presence of the existing systems. They should continuously support the existing systems’ evolution, while introducing a product line for the future. There is a continuous pressure in the automotive domain to bring to market new products, so it is impossible to put product development on hold to focus on developing a product line infrastructure.

Managers need to carefully plan available human resources and skills so that both the future product line and the current development cycle are on the right track. By using the ownership architecture, managers can assign different roles and responsibilities to their people. For example, the owners of the core components can be assigned as technical leaders of the family engineering activities. Although the proposed benefits might sound like the envisioned ones, our approach helps to quickly obtain a sound overview of the development organization by just using the source code history log, instead of asking many questions to the managers.
8.6.9 Summary of Results

In this section, we summarize the important results of our case study with Hitachi ECS:

- Ownership architecture recovery of existing products helped us to better understand the current development style (which is a dynamic team structure) without asking the managers a lot of questions. It gives a good insight into the current organizational style into how the development team is organized so far.

- We identified the potential system and component level experts. This knowledge will be used for assigning new roles and responsibilities in the future product line. We know whom to ask what and what not. Asking the right things to the right people is a critical aspect - ownership architecture just supports this practical aspect. Moreover, management effort is minimized to review top 10 candidates per component.

- Using the DConR metric, we found that nearly 100% of the developers of product A are also developers of product B. At the component-level, almost 80% of the developers developed the same component for both products. Using the DComR metric, we found that the new developers (i.e., those who were not in product A) are involved in the components of product B. Of course, the management knows this information, but it was also valuable to extract this data from the change history log to understand organizational insights.

- We also found that the recovered ownership architecture matches very well with the implemented architecture of the products. This is yet another analysis of Conway's hypothesis [60], which is also acknowledged by Bowman and Holt [34].

- By mainly using the source code history, we were able to assess the following: Domain expertise, staff turnover, and project management style. This assessment is the first step in adopting and institutionalizing a product line culture [28].

8.6.10 Limitations of the Case Study

Our definition of ownership is "just" based on the commit history. Although this definition works well for both the Hitachi's engine control variants, we cannot easily generalize this definition. What happens if the source code was refactored in the past and somebody else other than the original developer is the owner? For our purpose, such refactorings were not that critical because our interest is in identifying who has domain expertise in different parts of existing system even though that person may not own a component anymore. Another limitation is with respect to the "commit" process itself. Before applying our approach, it should be clarified with architects how the commit is actually performed. Otherwise, the commit history might not reflect the owners correctly.
8.7 Closing Remarks

This chapter reported a simple process for reconstructing an architectural view, called ownership architecture, which captures the relationship between the developers and the software components they work on. In the context of the planning phase of a product line, the ownership architecture helps to plan and perform the activities such as workshops for scoping, architecting, and architecture assessment and reengineering activities. Also, the ownership architecture supports management by providing guidelines for assigning new roles and responsibilities to perform product line engineering. The case study also shows that ownership architecture is a good indicator of the implemented architecture of the existing products. We conclude that there is so much one can learn from the history. To generalize the results of this chapter, it would be interesting to replicate the study on other organizations that dealt with mergers, spin-outs, international moves and external contractors.
Chapter 9
Quality by Design - Some Recommendations

9.1 Abstract

In the previous chapters, we analyzed several implemented systems and discovered architectural violations, testability, performance, and maintenance risks. Although such an analysis is valuable for improving the quality of existing systems, however, the detected architectural issues are sometimes “too late”, because the systems were already implemented, tested, and even fielded. In order to remove such issues, especially structural violations that are related to maintainability and often do not impact the immediate performance of the running systems, engineers would need strong support from managers; otherwise the detected issues may not be prioritized and removed. Based on architectural analyses of nearly two dozen industrial systems, we will now characterize the types of architectural rules that typically are violated. Furthermore, we discuss some practices that prevent architectural violations as well as practices that can be used to avoid testability and performance problems. We believe this body of architectural knowledge will help other practitioners during the architecture design phase to avoid taking “shortcuts” that need not pay-off in the long run.

9.2 Recommendations for Avoiding Structural violations

We all agree that if we do not follow the syntax of programming languages, the compiler will detect the problem at compile-time. Similarly, if our program contains functional errors, the users will most likely, sooner or later, encounter them and report them to the developers. In contrast, many of the architectural violations that our method is able to detect will neither by detected by compilers nor users. In this section, we investigate characteristics of such architectural rules and offer recommendations for avoiding the violations of rules in the first place.

9.2.1 Generate skeleton code templates using the specified architecture

In Chapter 2, the analysis of the CFS product line showed that there were a few architectural rules violated by its implementation. Especially, the module decomposition restriction rule, which define constraints on the decomposition of each application module’s internal structure, was violated by the implementation. Here, we elaborate more on this rule and discuss the reasons for violations of this rule and how it would have been avoided at the first place.

Consider the CFS’ specified module decomposition structure shown in Figure 9-1. Each box at the leaf-level denotes a private C function. The position of each box
9.2 Recommendations for Avoiding Structural violations

resembles the call structure, for example, QQ_AppMain must call QQ_AppInit and then QQ_AppPipe. This structure was documented in the developers’ guide so that developers can develop application modules with the same common look-and-feel, which helps programmers’ understand modules that were developed by others. In a nutshell, this specified decomposition structure is a detailed guide that explains the right way to use the APIs of the software bus. In fact, the documentation even contains code snippets for each of the routines as shown in Figure 9-1.

However, our analysis detected violations of this decomposition structure, that is, some of the application modules of the CFS did not follow this decomposition structure, thus, compromising the common look-and-feel. Figure 9-2 shows one “real” application Stored Command (SC) which did not follow the specified decomposition structure of the QQ demo application. The red crosses denote the absence of definitions of the specified private functions in sc_app.c file. That is, the internal structure of SC did not contain private functions AppPipe, NoopCmd, HouseKeepingCmd, and ResetCmd, in contrast to the specification of the QQ demo application. SC has its own decomposition of the main function and thus it has slightly different internal call structure.

![Figure 9-1: The specified module decomposition structure.](image-url)
In order to understand the reasons for these types of violations, we discussed the issues with the CFS team. It turned out that some of the team members were not aware of the fact that they have to follow the QQ demo application structure. Thus, different application modules have their own internal module decomposition structure. We “played” a bit with Google’s Android platform for mobile applications [7]. One related note here is that the Android IDE for Eclipse allows the developer to generate a skeleton of an Android-compliant application, which hooks into the Android platform. The developer has to “just” fill the skeleton. Based on these experiences, we derived the following concrete recommendations to avoid violations of module decomposition rules.

**Recommendation 1:** Generate skeleton code templates if modules should be built using the standardized interfaces of reusable frameworks.

This recommendation is targeted at systems that are built using a reusable framework, where it is possible to precisely specify the structure of each module and how each module should be using the framework. For example, the QQ demo application of the CFS captures the general decomposition structure of any CFS-compliant software module, which can be built using the APIs of the CFS core framework. Since the documentation of QQ contains example code snippets for each routine, the developers are expected to adapt these code snippets to their own needs, without changing the decomposition structure shown in Figure 9-1. In our opinion, if developers had a simple way to generate application modules by instantiating the QQ template, the detected violations of common look-and-feel among modules could have been avoided.

**Recommendation 2:** Documentation of architectural rules is not enough, communication is at least as important as documentation.

In our experience, we observed that programmers were not even aware of the existence of architectural rules they should be following. Depending on the safety-criticality or
9.2 Recommendations for Avoiding Structural violations

other important quality properties we need architectural enforcement in order to check whether the architectural rules are obeyed.

9.2.2 Organize the build process symmetric to the software structure

Here, we discuss some impacts of the structure of the build process on the violations of architectural rules using the SNAS, the CFS, and the GMSEC systems described above.

Let us analyze the structure of the build process of the SNAS system that allowed some undesirable dependencies to be easily introduced without being caught by compilers. In the SNAS, the build process is managed using shell scripts. The dependencies among these scripts have to be extracted in order to understand the build process. To this end, we wrote a few scripts that automatically extracted dependencies among the several dozen build scripts of the SNAS. Here, a relevant portion of the build view is shown in Figure 9-3 (a).

![Figure 9-3: The build process of the SNAS.](image)

In Figure 9-3 (b), we observe that there is one jar file for each subsystem except for the shareclient subsystem, which triggered further analysis to determine the contents of each jar file. It turned out that both the mocclient and the oamclient jar files have a copy of the classes of the shareclient. In addition, all subsystems jar files have a copy of both the common and framework jar files (see Figure 9-4). Similarly, the dsdm subsystem jar contains a copy of the string utility classes defined in the sdif subsystem (see dsdm.jar).

Let us recall the package dependency diagram, shown in Figure 6-12, of the SNAS system. In Figure 6-12, we could notice that the dependency from the dsdm to the sdif subsystem, which was, in fact, due to the dsdm subsystem using the string utilities of the sdif subsystem. This concrete example shows that because of the “relaxed” build process, it is possible for developers to easily access any class from any other subsystem, without being caught by compilers.

![Figure 9-4: Contents of the jar file for each subsystem.](image)
One problem with the structure of this build process is that the notion of components is hardly present at the build level. For example, consider the `sam.jar` binary file, it contains all the compiled classes of the `framework` and `common` folders, therefore, there is nothing that can stop or make it difficult for a developer to introduce undesired back links from these two folders to the classes of the `sam` folder. As a consequence, developers can easily violate architectural rules accidentally without the violation ever being noticed externally by compilers or users, thus, we recommended architectural control and monitoring.

In Chapter 2, we have shown that the Core Flight Software (CFS) implementation follows specified dependency restriction rules. One exception was that some modules directly used a few memory related routines of the standard C library, instead of the OSAL functions which wrapped them. Note that those routines are supported in all OS types. Therefore, those modules were not caught by compilers. Nevertheless, the CFS team has removed those violations in the latest version. Similarly, in Chapter 5, we have also shown that the Goddard Mission Services Evolution Center (GMSEC) implementation also follows dependency restriction rules. There are some interesting and useful lessons to be learned from these two very high quality frameworks, both based on the publisher-subscriber architectural style. Here, we discuss the importance of organizing the build process reflecting the structure of the software system.

In the CFS case, from the build process point of view, all core modules are compiled and linked together into one executable, without any knowledge of the presence of application modules. Each application module is also compiled into a separate object file, independent of other application modules. Also the OS abstraction layer of the CFS has a separate build process, independent of core modules or application modules. Therefore, developers cannot easily introduce spurious dependencies without being caught at compile-time.

In the GMSEC case, we have shown that there are no static compile-time, back-links from the software bus to publishers or subscribers, as specified. From the build process point of view, all modules of the software bus are compiled and linked into one executable, without any knowledge of the presence of publishers or subscribers. Each publisher and subscriber is also compiled into a separate executable. Thus, the software bus, publishers, and subscribers can be started independent of each other. Therefore, developers cannot easily introduce undesired direct dependencies, between publishers and subscribers, without being caught by the linker at runtime.

From these case studies, we recommend the following in order to prevent or impede the introduction of spurious dependencies that violate architectural rules.

**Recommendation 3:** Try not to keep the build process too “relax and open”. Otherwise, developers can easily access components and methods they should not be accessing.

**Recommendation 4:** Organize the build process symmetrically to the structure of the software system. That is, follow the notion of components at the build-level, too.
9.2.3 Composition of Architectural Styles

Let us consider the implemented package dependency diagram of the TSAFE system, shown in Figure 9-5, where arrows denote code relations (e.g., import, call). The specified design is the client-server architectural style. From the names of the subsystems, we could also infer that this system is likely to be based on the client-server architectural style.

![Figure 9-5: Implemented package dependency of the TSAFE.](image)

However, we do see a spurious, compile-time, back-link from the server package to the client package, which is not necessarily desired in a strict client-server architectural style. From the build process point of view, the classes of clients and servers are compiled together and the whole system is basically one executable. Thus, the violation of the back-link from the server package to the client package is not visible to other developers and users.

We analyzed this back-link and found that the server needs to access the interface of clients, in order notify all registered clients using this interface. In fact, the TSAFE client-server architecture style is implemented using the Observer design pattern [88]. Therefore, the server needs to know the client. The key point to note here is that the TSAFE design is a composition of two design constructs, namely the client-server and the observer design pattern. As a consequence, the resulting design is not strictly client-server anymore. Thus, we cannot claim that the back-link from the server package to the client package is a violation of an architectural rule. Given the fact that the system is not a strict client-server, we can also argue that the names of the packages are also somewhat misleading, it would have been better to assign names such as observers and subjects as package names. This experience has given rise to the following recommendations:

**Recommendation 5:** When composing multiple architectural styles together, identify and specify architectural rules that are valid after composition. Equivalently, some of the architectural rules of each style could become invalid after composition, thus, it makes little sense to check invalid or incorrect architectural rules against its implementation.
**Recommendation 6:** Naming of folders matter a lot for not only understanding but also following the architectural rules. Thus, try to assign names to folders in such a way that they convey the underlying architectural intentions.

### 9.2.4 Abstraction of errors is crucial for avoiding layering violations

In a strict layered architecture, each layer is only allowed to use the layer below it at compile-time. Each layer is an abstraction of below layers, meaning that special concerns are implemented on top of general concerns. One non-trivial issue, which is often ignored in the existing literature and also in practice, is related to managing errors raised by lower-level layers. In the systems we have analyzed, some of them do abstract low-level error types into high-level error types, whereas other systems fail to do so, resulting in violations of layer bypassing and also the leakage of error handling strategies of lower-level layers to higher-level layers.

Let us consider the SNAS’ database abstraction layer, discussed in Section 6.5.6.1, which offers database independent, abstract interfaces for modules to use in its interaction with the database. We have shown that each subsystem of the SNAS has its own database abstraction layer, and all usages of database tables are indeed through this interface only. Furthermore, we have also shown that in one of the subsystems sdif (see Section 6.5.7.1) the exceptions thrown by the database are not abstracted into database independent error types. As a consequence, the high-level layers that use the database abstraction layer are also aware of database concerns, even though the design permitted them to be unaware of such concerns. This layering violation could have been avoided at the first place if database errors were converted into abstract error types by the database abstraction layer.

It is worth noting that error abstraction is not only limited to a database concern, but also other concerns. For example, in the CFS product line, discussed in Chapter 2 and in Chapter 3, error abstraction is needed for abstracting the communication errors returned by software connectors (e.g., sockets, queues). In the CFS, the software bus module is built on top of queues or sockets. The operating system abstraction layer (OSAL) offers OS independent abstract interfaces as well as alternative implementations of several functions of operating systems. In this structure, the error abstraction occurs in two-steps. In the first step, the OSAL layer converts the error codes of queues (or sockets) of different operating systems into OS independent error codes. In the second step, the software bus converts error codes returned by the OSAL into error codes independent of the actual software connector type, which is independent of a queue or a socket connector type. Thus, the application modules that are using the software bus are agonistic to the actual connector type. Otherwise, application modules will have undesired direct dependencies to a specific connector type and deal with unnecessary low-level error details, too.

These two concrete examples demonstrated the importance of the abstraction of errors in order to avoid layering violations before they occur at the first place. Based on these analyses, we recommend the following:

**Recommendation 7:** In a layered architecture, it is crucial that each layer converts its errors into high-level errors, which are meaningful to high-level layers. Error
abstraction enables the higher layer being independent of the concerns that lower layers deal with.

### 9.3 Recommendations for Avoiding Behavioral Violations

In this section, we offer some recommendations that hold potential for avoiding behavioral violations in the first place. Our recommendations are derived from two case studies, discussed in earlier chapters of the thesis where we detected behavioral violations using our analysis methods. We would also like to emphasize the fact that, in general, it is very difficult to avoid all behavioral violations in the first place. This is because, in the state-of-practice, precise formal specifications of components’ interface behaviors are rather unusual, even in highly critical NASA contexts we are grateful to work with. This lack of formal specifications often leads to the misinterpretation of interface behaviors. Having said that, in our opinion, formal specifications are not easy to use, create, and maintain because they require special training and nontrivial mathematical skills.

#### 9.3.1 Specify behavioral constraints for architectural styles

Consider the NASA’s GMSEC framework, discussed in Section 5.4, which is based on the publisher-subscriber architectural style. Let us recall the fact that the GMSEC has a software bus that allows applications to communicate indirectly with each other by publishing and/or subscribing to messages using the interfaces of the software bus. In addition, one appealing feature of the GMSEC architecture is that it contains an abstract interface of the software bus with several implementations of the same interface using APIs of different vendors. We have shown that there is a behavioral inconsistency among the different implementation of the same interface, which affects the plug-and-play of software components. More specifically, in one implementation of the interface, it is possible to subscribe to a specific message more than once without an intermediate unsubscribe of the same message. The inconsistency lies in the fact that other implementations of the interface do not allow consecutive subscription to the same message. This bug was caused by the lack of a behavioral specification of the architecture style. Even though the behavioral constraints of the publisher-subscriber style may be intuitively clear, we recommend at least some semi-formal (or even English) specification of behavioral constraints of styles. Devils hide in details. Thus, we should attempt to enumerate nitty-gritty of architectural styles even though their high-level concepts might appear to be crystal clear. Based on this experience, we recommend the following:

**Recommendation 8:** *We cannot assume that all behavioral constraints of well-known architectural styles will be consistently followed in the implementation. Therefore, we recommend a specification of behavioral constraints at least semi-formally or even in natural language.*

#### 9.3.2 Standardize the interfaces of software components

In practice, integration of software components developed by one or more teams is challenging and error-prone. Many of the integration problems are due to the lack of
precise understanding of the interfaces and differences between the protocols of software components. For example, the GMSEC team offered the following two figures, which we use here to explain the importance of applying interface-oriented architecture design in order to avoid behavioral problems.

Figure 9-6 shows the traditional approach of highly coupled component-to-component socket connection. The addition or removal of a component or interface will cause significant perturbations to the connections and data flow. Furthermore, the integration of these components can take significant effort because all components have their own protocols.

![Figure 9-6: Conventional design with socket connections.](image)

In contrast, Figure 9-7 shows the components interfacing to the GMSEC Message Bus, but not to each other in a layered approach. In this design, software components incorporate the standardized GMSEC API and use standardized messages to interface to the message bus. The underlying middleware takes care of message routing and communications management thereby simplifying the components and freeing them of integration responsibility and complexity.

Similar to the GMSEC software bus, the CFS product line also contains a software bus with standardized APIs, as discussed in Chapter 2 and Chapter 3. Both systems emphasize the importance of interface-oriented architecting, meaning that all software components are based on standardized interfaces. Therefore, many of the component integration problems could be avoided before they occur. Based on these experiences, we recommend the following:

**Recommendation 9:** Standardize the interfaces of software components so that they all share a common look-and-feel, and they can be integrated in a uniform way. Ideally, systems implement a framework where components can be plugged-in and plugged-out.
9.3.3 Introduce an authorization mechanism

Consider Ricoh’s MFP software prototype, discussed in Section 4.5, which is based on the pipe-and-filter architectural style. The system offers flexibility to the user, meaning that the user has access to a catalog of software components (i.e., filters) and can compose the available components in any order. The configuration of filters and pipes are stored by a configuration manager. At runtime, it is possible for a filter to obtain a direct reference to its neighbor filters using the configuration manager. Let us also recall the fact that each filter is internally decomposed into a collection of layers such as the GUI and processing layer. The interfaces of the GUI layer of each filter are intended to be activated in reaction to the user’s interaction with the system using the GUI panel. However, we have shown that there is no protection of the filter interfaces and that it therefore was possible to invoke methods provided by the GUI layers directly from other filters. We have, in fact, shown that an “evil” filter can successfully halt the whole system by illegally invoking methods of the GUI layer of its neighbor filter. This problem could have been avoided if there is some an authorization mechanism that would prevent an “evil” filter to illegally activate the GUI functions of its neighbor filter. Based on this experience, we recommend the following:

**Recommendation 10:** Introduce an authorization mechanism to prevent illegal usage of software components. This recommendation is especially targeted for systems that allow previously unknown components to work together at runtime.

9.4 Avoiding Testability Problems

In the previous chapters, we analyzed architectural principles that impede or facilitate testing of several real-world systems. Based on these experiences, in this section, we offer some recommendations to make testing easier in practice.
9.4.1 Develop a Database Abstraction Layer

Databases are prominent entities in many systems. From a testing point of view, special considerations have to be given for interactions with a database, because a) the necessary testing data must be populated in database tables, otherwise it may not be possible to traverse different paths of the system under testing; b) before executing each test case, all database tables needs to be cleaned and should be in an appropriate state for testing that particular scenario; c) access to the database is usually many orders of magnitude slower than accessing the data in RAM (random access memory), thus running the test cases will consume time. In Section 6.5.6.3, we have shown that the snif subsystem of the SNAS has a database abstraction layer that is used to interact with the database. The abstraction layer facilitates testing since it is possible to switch on and off all interactions to the database, allowing all database interactions to be redirected to dummy methods that populate in-memory fake data, without changing the source code. Based on this concrete example, we recommend the following:

**Recommendation 11:** For improved testability, design the database abstraction layer in such a way that it is possible to redirect interactions to the database to dummy methods that can populate fake data. See Section 6.5.6 for an in-depth discussion.

9.4.2 Develop a Operating System Abstraction Layer (OSAL)

If a module directly uses the interfaces of operating systems, it affects testing because a) the test cases of the module under test must be developed for each operating system; and b) every machine must have an installation of all operating systems, which is difficult to set-up and install especially embedded OS such as VxWorks, RTEMS, etc. In contrast, if the architecture has an abstraction of OS functions, then the module and its test cases can be made fully agnostic to the underlying OS. We can learn from the GMSEC and the CFS projects on how to abstract the underlying OS, and make testing easier.

Let us consider one of the views of the GMSEC’ implemented architecture, shown in Figure 9-8, where arrows denote call dependencies. This figure captures how the APIs related to multithreading of different OS types are abstracted. We can see that all call dependencies to thread libraries are only through the abstract interfaces defined in middle layer. Higher level layers use only the Framework APIs (e.g., start, enterMutex). The external threading functions pthread_create and pthread_mutex_lock are used for the POSIX standard, whereas CreateThread and WaitForSingleObject are corresponding symmetric functions for Windows. The GMSEC implementation uses conditional preprocessor to choose between Windows and POSIX alternatives.

One benefit of abstracting the OS functions is that the modules of the high-level layer can be tested by creating light-weight mock implementations of the abstracted functions. This capability is crucial especially for unit-testing purposes, because a) the goal of unit-testing is to test each module in isolation, assuming the correctness of the modules it uses, and b) mock implementations save testing time due to their “light-weight” nature in comparison to real implementations of OS functions.
Figure 9-8: The abstraction of thread APIs in the NASA’s GMSEC.

Figure 9-9 shows the OSAL in the CFS’ architecture. Basically, there is a common interface and alternative implementations for different OS types. At build time, the CFS can be compiled for a particular OS by selecting the appropriate OS type in the build configuration file.

As we discussed in the analysis of the CFS’ test architecture, this abstraction of OS functions facilitates unit testing because mock implementations of OS interfaces can be easily created, thus, modules can be unit tested without the need for having the special OS installed and running. Note also that as a positive side-effect of the OS abstraction layer, the unit test cases of the module under test are independent of the behavior of OS functions, making the maintenance of test cases easier; see Chapter 3 for more details.

Based on these experiences, we recommend the following:
**Recommendation 12:** For improved testability, abstract the necessary OS functions by creating the OS independent interfaces so that modules can be easily unit tested on different OS types using mock implementations.

### 9.4.3 Testability - Constructors and Static methods

In the object-oriented paradigm, constructors and static methods cannot be overridden. Therefore, if critical or heavy-weight operations, such as accessing a database, COTS, or a server, are performed within constructors and static methods, there is absolutely no possibility to override the default behavior into a dummy light-weight behavior for testing purposes. For example, in Section 6.5.7.2, we have shown that several classes of the sdif subsystem of the SNAS cannot be tested without the database being up and running, because within the constructors of those classes, instances of classes that will interact with the database are created. Based on this testability problem, we recommend the following:

**Recommendation 13:** For improved testability, avoid performing critical operations within constructors and static methods, because they both cannot be overridden with a light-weight dummy implementation.

### 9.4.4 Separate GUI concepts from core logic

The GUI is yet another prominent component of most software systems. From a testing point of view, the GUI offers some challenges because, in general, a) humans need to enter the input data and validate the response from the system; b) it is not easy, if not impossible, to store the input data and replay it when the system under test changes. To tackle these challenges, the architecture has to separate GUI concepts from the core logic. If they are not separated, then we cannot regularly test the core logic using an automated testing framework like the JUnit. For example, in Section 6.5.8.2, we offered an example design that has a low testability because the interface of the routine assumed that the input data will always come from the GUI. Based on this experience, we recommend the following:

**Recommendation 14:** For improved testability, the interfaces of the core logic should not take GUI objects as arguments, otherwise the core logic cannot be tested without the GUI being up and running.

### 9.4.5 Do not overuse abstract data types or objects

In several cases, we observed that the arguments of methods often overuse abstract data types, where simple or primitive data types (e.g., int, float) would be sufficient. From a testing point of view, if we want to test a method that takes objects (or abstract data types in procedure oriented programming) as arguments, we have to first create all objects and then their dependent objects. This makes testing very difficult and the test code becomes complex and difficult to maintain. Based on this experience, we recommend the following:
9.4 Avoiding Testability Problems

**Recommendation 15:** *For improved testability, if methods need only basic data types for their operations, avoid using abstract data types or objects as arguments of methods.*

### 9.4.6 Open some internal details of modules

We agree that information hiding is a good software engineering principle. However, from a testing pointing of view, architects have to make trade-offs between the ease-of-testing and information hiding. In Section Chapter 3, we have shown that the test program needs to access and manipulate some internal data structures in order to test the system for a particular scenario. Let us recall the concrete example of testing the behavior of the CFS when the user is trying to load more than the allowed limit of the number of shared libraries. The test program simply manipulated the data structure that stores the number of shared libraries that are currently loaded. This way the tester was able to quickly test the behavior of system without creating and loading several shared libraries. If the data structure was hidden internally to the modules, it would not have been easy for the test program to change the state of the system’s configuration to the desired state. If access to the internal data structure was not possible, the tester had to first create at least one more than the allowed limit of shared libraries and then the tester had to load each of the libraries to test this scenario. Thus, it is effort intensive and time-consuming to create all shared libraries and test this scenario. Based on this experience, we recommend the following:

**Recommendation 16:** *For improved testability, some of the internal details of modules have to be made public. However, as a trade-off, architectural rules can be defined to enforce the usage of internal details restricted to test programs, see Sections 3.7.2 and 3.7.3 for more details.*

### 9.4.7 Avoid reusing the same return code for different errors

We have observed that, in several cases, the return code of a method is overloaded for different scenarios. For example, in the CFS case study, we have shown a function that returns the same return code from different paths for different types of input errors, see Section 3.7.3. We have also shown that testing such functions is challenging because it is difficult to decide the actual path traversed by the given input. Based on this experience, we recommend the following:

**Recommendation 17:** *For improved testability, avoid using the same return code from different paths of a method.*

### 9.4.8 Assign each entry point of a task to a separate module

We have highlighted in Section 7.3.2.4 that several entry points to the runtime tasks of the CARA medical device software are defined in one C file. One consequence of such partition is that changes to code elements of this file require recompilation and retesting of all tasks using the newly compiled binary code. This time could be reduced if the entry functions as well as functions uniquely used by the entry functions are moved to separate files. Only the tasks that are affected due to recompilation need to be unit tested again. Because such partition structure does not allow independent compilation
of task-specific code from other tasks, it is impossible to produce a separate executable for each task to facilitate unit testing. This partition structure as such is not amenable to task-by-task verification of safety properties and difficult, if not impossible, to unit test and verify. Based on this experience, we recommend the following:

**Recommendation 18:** For improved testability, assign entry functions of tasks into separate modules so that each task can be unit tested independent of other tasks.

### 9.4.9 Abstract the connector details from components

We have highlighted in Section 7.3.2.6 that the runtime components (i.e., tasks) of the CARA have a hard binding to the connector type used for communication among the tasks. That is, each of the tasks is aware of the fact that it uses queues as connectors to send and receive messages to other tasks. This lack of abstraction of the connector type makes unit testing difficult because one has to create and initialize necessary queues to do testing. Instead, if the connector type is abstracted out of each task, then each task can be unit tested by using lightweight connectors, such as a buffer or a file. Based on this experience, we recommend the following:

**Recommendation 19:** For improved testability, abstract the connector type from components so that components can be unit tested using lightweight connectors.

### 9.5 Avoiding Performance Problems

In general, improving the performance of a system after its implementation is challenging. In this section, we offer some recommendations based on our architectural analysis projects, where we collected some architectural strategies that are useful to prevent or minimize performance problems.

#### 9.5.1 Apply the database connection pool design pattern

As explained in the SNAS case study, frequently creating and deleting connections to a database adds overhead. By using the connection pool design pattern, where a set of connections are created up-front and reused for several requests, the overhead can be reduced, see Section 6.5.6.4. Based on this experience, we recommend the following:

**Recommendation 20:** For improved performance, consider using the database connection pool design pattern if your system has to handle several requests to the database, especially frequent short-lived sessions.

#### 9.5.2 Apply the transfer object design pattern instead of RPCs

As explained in the SNAS case study, remote procedure calls are inherently slow. By using the transfer object design pattern, where objects are passed on the communication channel, the overhead can be reduced, see Section 6.5.5.4. Instead of making several remote procedure calls, clients can get necessary data in one method call, because servers can package the necessary data and send to clients in one object. Based on this experience, we recommend the following:
**Recommendation 21:** For improved performance, consider using the transfer object design pattern, instead of remote procedure calls.

### 9.5.3 Pay attention to threading models

As explained in the SNAS case study, several of the detected performance problems could have been avoided by choosing appropriate threading models. Especially, if the system has to dispatch events to listeners, the architect should decide whether or not the listeners will process the data in the same thread used for the notification. In Section 6.5.8.1 and Section 6.5.5.5.2, we have highlighted the fact that the detected performance risk was due to the usage of the same thread for notification of events as well for processing of the notified events. This threading model has the risk of affecting performance because listeners are notified sequentially one after another in a synchronous way. Based on this experience, we recommend the following:

**Recommendation 22:** Threading models have to be carefully defined and analyzed. Otherwise, there is a high risk of poor performance.

### 9.5.4 Early binding of variation points than late binding

By *early binding*, we mean before runtime (e.g., compile-time); by *late binding*, we mean during the execution of the system. If the system has variation points, at some point in time, they have to be resolved. From a performance point of view, we recommend early binding of variation points. Consider, for example, the CFS product line, where one variation point is related to the OS type. In the CFS product line, the OS variation point is resolved at link-time because users’ know before runtime the actual OS type the system will be running on. The architecture of the CFS has an abstract interface and alternative implementations for all supported OS types. The binding of the abstract interface to the actual implementation occurs at link-time, thus, there is no runtime overhead due to the OS variation point.

On the other hand, the SNAS system has a variation point that allows the developer to switch on and off all interactions to the database, and redirect all database interactions to dummy methods. However, this redirection to dummy methods occurs at runtime, see Section 6.5.6.3, which means there is an added overhead in resolving the variation point. We have also mentioned that this runtime overhead could be reduced by using standardized frameworks such as Spring [267] or Google’s Guice [122]. Based on these experiences, we recommend the following:

**Recommendation 23:** For improved performance, give preferences to the early binding of variation points, instead of late-binding.

### 9.6 Closing Remarks

In this chapter, we investigated how we can avoid architectural problems in the first place. We presented constructive recommendations to avoid a) the degeneration of the software architecture present in the source code, b) testability problems that impede software testing, c) maintenance problems that make the evolution of software hard, and d) performance problems that impact the usefulness and usability of software. We
presented a large body of practically applicable architectural knowledge, mined from our systematic study of two dozen industrial systems, that can be used for not only improving the quality of existing systems but also during the architectural design of new systems in a green field scenario. We stressed the importance of the abstractions of different commonly occurring elements, such as GUIs, Middleware, Databases, OSes, Hardware, and COTS, in order to make systems easy to test and maintain. Naturally, an architectural design with good and sustainable abstractions requires technical creativity and expertise, understanding the organization and business relevance, and being able to predict the future needs. Abstractions require investment, but given their qualitative benefits that are discussed in this thesis, we believe it is definitely worth considering before taking “shortcuts”. Thus, organizations need to evaluate the business relevance of our recommendations related to abstractions for their business context before applying the recommendations into practice.
Chapter 10
Reverse Engineering Tool Suites

The goal of this chapter is to describe the various reverse engineering tools that are being developed by us as part of this work, and are mentioned in previous chapters. We can roughly classify reverse engineering tools into two high-level categories, namely the Extraction or Data Collection Tools and Analysis Tools. The analysis tools facilitate the abstraction and the presentation phases of reverse engineering as proposed by Krikhaar [169], for example.

10.1 Extraction Tools

In reverse engineering, often the first step is extracting data from existing artifacts [37]. As discussed earlier, real-world systems are often both large and complex and therefore we need tools to facilitate extraction of data for analysis. We can classify extraction tools into two categories, namely static and dynamic data extraction tools, as explained below.

10.1.1 Tools for Extraction of Static Data

By static data, we mean all types of data that can be collected without running the system under analysis. We extract static data from the directory structure and as well from the source code of the system under analysis, as described below.

10.1.1.1 Extracting data from directory structures

For several of the reverse engineering activities, we have come across the need to extract data such as extensions of file names, the hierarchy (a.k.a. Part_of) relation between directories and sub-directories or files, and the number of lines in each file. For this, we have gladly reused many of the extraction scripts of Krikhaar [169], written in Perl. We have ported and/or customized those scripts to several scripting languages as well as to the Java language.

Here, we placed those scripts which might be of interest for others. Each script prints output to the standard display so that we can compose scripts in many different ways.

10.1.1.1.1 Extracting all extensions of files

A filename extension is a suffix to the name of a computer file applied to indicate the encoding convention (file format) of its contents [80]. The exact definition, giving the criteria for deciding what part of the file name is its extension, belongs to the rules of the specific filesystem used; usually the extension is the substring which follows the last occurrence, if any, of the dot character (e.g., txt is the extension of the filename readme.txt, html the extension of mysite.index.html).
#!/usr/bin/sh

# Purpose: Extract all extensions of files
# Input 1: Directory to search
# Output: List of extensions

# Use "find" to list all files and then use "awk" to extract file extensions
find $1 -type f | awk -F . '{print $NF}' | sort | uniq -c

This script also has the capability to identify extensions of files including a) those files that have more than one extension and b) those files that do not have any extension at all (e.g., makefile). If a file has no extension, then this script prints the full file name.

My supervisor, Prof. Dr. Chris Verhoef, offered the following two variants. The first variant locates files with more than one extension. The second variant locates files with no extension.

10.1.1.1.2 Extracting multiple extensions of files

#!/usr/bin/sh

# Purpose: Extract multiple extensions of files
# Input 1: Directory to search
# Output: List of multiple extensions

# Use "find" to list all files and then use "awk" to match more than one "."
Find $1 -type f | awk -F/ '{if (match($NF,/.+/)>0) print $0}'

10.1.1.1.3 Extracting files with no extensions

#!/usr/bin/sh

# Purpose: Extract files with no extensions
# Input 1: Directory to search
# Output: List of files with no extensions

# Use "find" to list all files and then use "awk" to match zero "."
Find $1 -type f | awk -F/ '{if (match($NF,/.\)/)==0) print $0}'

10.1.1.1.4 Extracting the hierarchy relation

#!/usr/bin/sh

# Purpose: Extract the hierarchy relation
# Input: Directory to search
# Output: <Parent Dir> <Child dir>

# Use find to extract all directories and files and then use the sed tool to extract the hierarchy
find $1 | sed 's/\(.+/\)/\(.+\)/\1 \2/g'

If we run the above script, it shall print the hierarchy relation of our input directory, similar to the below output:
10.1.1.5 Counting the number of lines per file

```bash
#!/usr/bin/sh
# Purpose: Count the number of lines for given files
# Input 1: Directory to search
# Output : <number of lines> <file name>

#Use find to extract all files and then use word count (wc)
find $1 -type f | xargs wc -l
```

If we run the above script, it shall print the sorted number of lines of each file within our input directory, similar to the below output:

```
100 /sys/client/model.java
200 /sys/client/view.java
300 /sys/server/logic.java

600 total
```

10.1.1.1.6 Counting the frequency of words in an input stream

The following awk script will compute the frequency of each word (a.k.a. a token) in the given input stream. We select tokens that are identifiers as defined in modern computer programming languages. This awk script leverages the fact that we are allowed to use a string as an index of an array in awk.

```awk
# Prints the frequency of tokens in the given input stream
{
  # Get rid of non-alphanumeric characters
gsub( /[a-zA-Z_0-9]*/, " ")

  # Select tokens that are identifiers
  for(i=1; i <= NF; i++)
    if($i ~ /^[A-Za-z][A-Za-z0-9]*$/) freq[$i]++
}

END {
  sort = "sort -k 2 -n r"
  for (word in freq)
    printf "%s\t%d\n", word, freq[word] | sort
close(sort)
}
```
Let us assume this script was saved as `word_freq.awk`. In order to run this script, we need to pass an input stream. For example, if we have to count the frequency of each word present in our script, we run the script as follows:

```
cat word_freq.awk | awk -f word_freq.awk | more
```

Often, commonly used stop-words such as “a”, “the”, “i”, and so on emerge as the high frequency words. This effect can be avoided by filtering such stop words as well as the words of length one from the output of the above script.

### 10.1.1.2 Stripping C/C++/Java style comments

In order to remove comments present in the source code of the system under analysis, we used a deterministic finite automaton (DFA). Our DFA can handle both multi-line and single line comments. We adapted the DFA presented in [254] to handle comments that might be present within quoted strings. In contrast to [254], this DFA is capable of handling the escape character (i.e., “\”) that might be embedded within quoted strings, see Figure 10-1.

![Figure 10-1: An automaton for stripping C/C++/Java style comments.](image)

The Java program that implements this DFA for stripping source code comments is provided below. The program is based on the same seven states that the DFA consists of. The program accepts input from standard input and prints output to standard output.

```java
/*************************************************************************
* Compilation:  javac CommentStripper.java
* Execution:    java CommentStripper < source.c
* Reads in a source program and removes all of the comments using a 7
```
* state deterministic finite automaton.
*
*************************************************************************/

public class CommentStripper {
    public static void main(String[] args) {
        final int CODE = 0;  // parsing normal code
        final int SLASH = 1; // found a leading '/'
        final int BLOCK = 2; // in a block c-style comment
        final int LINE = 3;  // in a line comment
        final int STAR = 4;  // found a trailing * in a block comment
        final int QUOTE = 5; // in a quote
        final int ESCAPE = 6; // found the escape character (i.e., \)

        int state = CODE;  // current state
        final char EOL = System.getProperty("line.separator").charAt(0); // End-of-line constant
        final int EOF = -1; // End-of-file constant
        int whatever;       // current read char

        while((whatever = StdInput.readChar()) != EOF) {
            char c = (char)whatever;
            switch(state) {
                case CODE:
                    if (c == '"') { state = QUOTE; System.out.print(c); }
                    else if (c == '/') { state = SLASH; }
                    else { System.out.print(c); }
                    break;
                case SLASH:
                    if (c == '*') { state = BLOCK; }
                    else if (c == '/') { state = LINE; System.out.print("/"); }
                    else { state = CODE; System.out.print("/"); }
                    break;
                case BLOCK:
                    if (c == '*') { state = STAR; }
                    break;
                case STAR:
                    if (c == '/') { state = CODE; System.out.print(" "); }
                    else if (c == '*') { state = STAR; }
                    else { state = BLOCK; }
                    break;
                case LINE:
                    if (c == EOL) { state = CODE; System.out.println(); }
                    break;
                case QUOTE:
                    if (c == '"') { state = CODE; System.out.print(c); }
                    else if (c == '\\') { state = ESCAPE; System.out.print(c); }
                    else { state = ESCAPE; System.out.print(c); }
                    break;
                case ESCAPE:
                    if (c == QUOTE) { state = CODE; System.out.print(c); }
                    break;
            }
        }
    }
}
10.1.1.3 Extraction of code relations from source code

In order to extract data from source code, we need parsers for each programming language. In our work, we developed a tool called FACE for the C/C++ language. FACE was written in Java. For analyzing systems based on other programming languages, we used COTS, for example, the Understand tool [284] offers parsers for several programming languages.

Here, we briefly explain the FACE tool using the below sample code snippet, stored in main.c.

```c
#include<stdio.h>
void run() {
    printf(" I am running ...");
}
void main() {
    printf(" Start running ...");
    run();
    printf(" Done!");
}
```

The user interface of the FACE is similar to that of other familiar C compilers such as GCC. This makes it easier to embed FACE within Makefiles. For a given C file, it extracts the set of relations (e.g., Include, Call) and places each relation in a separate file so that we could use query languages such Relation Partition Algebra (RPA) to query the extracted relations [169].

To run the FACE tool, the user has to pass include paths and preprocessor symbols for each C file, similar to C compilers.

```
FACE -I <include path> -D <Define Symbol> <Source File 1> <Source File 2> ... 
```

If we run the FACE tool for the above main.c example, the extracted `Call` relation will be:

```python
main printf 2
main run 1
```
run printf 1

Each tuple of the extracted Call relation is in the format: <caller> <callee> <weight>. For example, main calls printf twice meaning that the weight is 2. In RPA, this relation is called a multi-relation.

Similarly, the Include relation for the above main.c example, under DOS, will be:

<absolute path>\main.c stdio.h

10.1.1.3.1 Overview of the FACE parser design

The FACE tool is based on the open source Eclipse CDT parser which has the capability to build an abstract syntax tree for each input C file. Furthermore, the CDT parser offers a call-back capability that enables us to register for notifications of events that are generated during the traversal of the abstract syntax tree. For example, the user can register for call-backs when parsing events are generated by the CDT parser. Parsing events could be, for example, the parser is entering the body of a C function, the parser is handling function references (i.e., calls), the parser is entering a header file, the parser is exiting a header file, etc. Because of these novel capabilities offered by the CDT parser, we decided not to develop our own parser. To this end, we developed a tool that wraps the CDT parser by reusing the call-back capability offered by the CDT parser.

Here, we explain the key interfaces and classes of the CDT parser (CDT core 4.0.3) and offer an example call-back program for extracting the Call relation from source code.

// This interface is part of Eclipse CDT core 4.0.3

class ISourceElementRequestor {

    public boolean acceptProblem( IProblem problem );
    public void acceptMacro( IASTMacro macro );
    public void acceptVariable( IASTVariable variable );
    public void acceptFunctionDeclaration( IASTFunction function );
    public void acceptASMDefinition( IASTASMDefinition asmDefinition );
    public void acceptTypedefDeclaration( IASTTypedefDeclaration typedef );
    public void enterFunctionBody( IASTFunction function );
    public void exitFunctionBody( IASTFunction function );
    public void enterInclusion( IASTInclusion inclusion );
    public void enterClassSpecifier( IASTClassSpecifier classSpecification );
    public void enterTypedefDefinition( IASTTypedefDefinition typedef );
    public void enterMethodDeclaration( IASTMethod method );
    public void enterMethodBody( IASTMethod method );
    public void exitMethodBody( IASTMethod method );
    public void acceptField( IASTField field );
    public void acceptClassReference( IASTClassReference reference );
    public void acceptTypedefReference( IASTTypedefReference reference );
    public void acceptNamespaceReference( IASTNamespaceReference reference );
    public void acceptEnumerationReference( IASTEnumerationReference reference );


The users of the CDT parser can plug-in their own implementations of the above interface to the parser. Fortunately, the CDT parser also has a default implementation (class StructuralParseCallback) of the above interface, which implies that users can override the default behavior of selected methods according to their own needs.

For example, the below code snippet explains how we can extract the Call relation between functions of a given input file. Note that we inherit from the base class of Eclipse and override only the methods related to parsing of functions. The interested reader could develop programs similar to the below snippet for extracting other relations such as the Include relation between files, the Inheritance relation between classes, etc.

```java
public void acceptVariableReference(IASTVariableReference reference);
public void acceptFunctionReference(IASTFunctionReference reference);
public void acceptFieldReference(IASTFieldReference reference);
public void acceptMethodReference(IASTMethodReference reference);
public void acceptEnumeratorReference(IASTEnumeratorReference reference);
public void acceptParameterReference(IASTParameterReference reference);
public void acceptFriendDeclaration(IASTDeclaration declaration);
public void exitTemplateDeclaration(IASTTemplateDeclaration declaration);
public void exitTemplateSpecialization(IASTTemplateSpecialization specialization);
public void exitTemplateExplicitInstantiation(IASTTemplateInstantiation instantiation);
public void exitLinkageSpecification(IASTLinkageSpecification linkageSpec);
public void exitClassSpecifier(IASTClassSpecifier classSpecification);
public void exitNamespaceDefinition(IASTNamespaceDefinition namespaceDefinition);
public void exitInclusion(IASTInclusion inclusion);
public void exitCompilationUnit(IASTCompilationUnit compilationUnit);

// Purpose: Extract Call Relation between functions of the given input file
// Output: <caller function> <callee function> on the standard output
import java.io.File;
import java.util.LinkedHashMap;
import org.eclipse.cdt.core.parser.IParser;
import org.eclipse.cdt.core.parser.IProblem;
import org.eclipse.cdt.core.parser.IScanner;
import org.eclipse.cdt.core.parser.ScannerInfo;
import org.eclipse.cdt.core.parser.ast.IASTFunction;
import org.eclipse.cdt.internal.core.parser.InternalParserUtil;
import org.eclipse.cdt.internal.core.parser.StructuralParseCallback;
```
The public methods of this class are called at runtime by the Eclipse CDT parser class CallGraphExtractor extends StructuralParseCallback {

// file to be parsed
private String inputFileName = null;

// name of the function that is currently entered
private String currentlyEnteredFunction = null;

public CallGraphExtractor(File inputFileName) {
    this.inputFileName = inputFileName.getAbsolutePath();
}

public void acceptFunctionReference(IASTFunctionReference reference) {
    // print <caller> <callee> pairs
    if (currentlyEnteredFunction != null)
        System.out.println(currentlyEnteredFunction + " " + reference.getName());
}

public void enterFunctionBody(IASTFunction function) {
    currentlyEnteredFunction = function.getName();
}

public void exitFunctionBody(IASTFunction function) {
    currentlyEnteredFunction = null;
}
}

In order to run the CallGraphExtractor class, we need to use two more interfaces, namely the IScanner and IParser of the CDT parser. The code snippet shown below can be used to run the call-back class CallGraphExtractor and extract the Call relation for the given input c file with include paths and appropriate preprocessor symbols for instantiation, if any.

import java.io.File;
import java.io.OutputStream;
import java.util.Map;
import org.eclipse.cdt.core.parser.IParser;
import org.eclipse.cdt.core.parser.IScanner;
import org.eclipse.cdt.core.parser.ParserFactory;
import org.eclipse.cdt.core.parser.ParserLanguage;
import org.eclipse.cdt.core.parser.ParserMode;
import org.eclipse.cdt.core.parser.ScannerInfo;
import org.eclipse.cdt.internal.core.parser.InternalParserUtil;
public class CallGraphExtractorTest {

public static void main(String[] args) {

    /* Our command line argument parser is for extracting input arguments
     * such as the include path, preprocessor symbols, etc. */
    CommandLineArgsParser myCmdLineArgsParser = new CommandLineArgsParser(args);
    String inputFile = myCmdLineArgsParser.getInputFile();
    String[] includePath = myCmdLineArgsParser.getIncludePath();

    /* Preprocessor symbols are stored as key value pairs (e.g., Language = Dutch) */
    Map preprocessorSymbols = myCmdLineArgsParser.getPreprocessorSymbols();

    /* IScanner and IParser are part of the Eclipse CDT core parser */
    IScanner myIScanner;
    IParser myIParser;

    /* CallGraphExtractor is our plug-in to the Eclipse CDT parser */
    CallGraphExtractor myCallGraphExtractor = new CallGraphExtractor(inputFile);

    /* Create a Scanner by passing the input file to parse with include paths and preprocessor
     * symbols. Note also that we need to pass our CallGraphExtractor instance to the CDT parser for
     * call-backs to work */
    myIScanner = ParserFactory.createScanner(InternalParserUtil.createFileReader(inputFile),
                                               inputFile,
                                               new ScannerInfo(preprocessorSymbols, includePath),
                                               ParserMode.COMPLETE_PARSE,
                                               ParserLanguage.C,
                                               myCallGraphExtractor,
                                               null, null);

    /* Create a Parser */
    myIParser = ParserFactory.createParser(myIScanner, myCallGraphExtractor,
                                            ParserMode.COMPLETE_PARSE,
                                            ParserLanguage.C, null);

    /* Start parsing ... */
    if (myIParser.parse() == true) {
        System.out.println("Parse successful for file: " + inputFile.getAbsolutePath());
    } else {
        System.out.println("Parse unsuccessful for file:" + inputFile.getAbsolutePath());
    }
}
There are several practical peculiarities we need to be able to handle for extracting code relations. For example, we need to be able to detect missing header files which may not be present in the source code archive offered by customers. For this purpose, we have developed simple, yet useful, search scripts that check whether or not the source code archive has all necessary header files. Sometimes missing header files can be freely downloaded from the Internet. Otherwise, we have to request a copy of necessary header files from customers. If this is an audit situation, missing header files is a finding to report to customers.

Another practical issue is that the source code archive may not be clean in the sense that some of files may be exact duplicates (e.g., x_temp.c, x_v1.c, x_v2.c) which should not be included in data collection. For this purpose, we developed tools for detecting duplicates of files, as explained in the next section.

10.1.1.4 Building an index for search and similarity analysis

As mentioned in Chapter 5 and Chapter 6, we developed a language independent tool for searching and calculating text-based similarity among software artifacts. Our tool leverages Lucene’s Java APIs for building an index of software artifacts [189]. Informally, an index is similar to the index we see in text books, in that we extract words (a.k.a. terms) from software artifacts. Lucene is a powerful framework for building such an index. It has a built-in collection of text analyzers, which extract words from input files in different ways. For example, the lower case analyzer converts all words into lower case during the construction of an index; the stemming analyzer converts all words into a base format so that players and player are treated as one word. The user can select the text analyzer of interest. It is also possible to combine different text analyzers in a pipe-and-filter style in such a way that the output of one text analyzer can be used by another text analyzer.

One appealing aspect of Lucene is that the user can develop his own text analyzer and plug it in to its framework. At runtime, using call-backs, Lucene will call the user’s text analyzer. We took advantage of this plug-in mechanism and developed a simple, yet powerful, text analyzer for source code. Our text analyzer is independent of the grammar of programming languages. Thus, it is used for collecting data even from makefiles, configuration files, requirements, readme, etc. Our text analyzer recognizes camel casing (e.g., startServer, start_server) often used in source code. For example, startServer (or start_server) will be split into start, Server, startServer (or start_server).

In order to run our indexer tool, the user has to specify the directory to parse, the list of file extensions, and the list of stop words (e.g., a, an, the, etc.), which will be excluded during parsing. The output of the indexer tool is a binary representation of the index as prescribed by Lucene. In Section 10.2, we explain how the extracted index can be used to search and calculate text-based similarity among indexed artifacts.

10.1.2 Tools for Extraction of Dynamic Data

By dynamic data, we mean data that is collected at runtime. We based our data collection on several existing technologies. For collecting data from a running C
system, we have developed tools using the APIs of Pin [230]. For collecting data from a running Java system, we have developed tools using the aspect-oriented programming paradigm, as shown in Section 4.4.3.

Here, we present some sample code, developed using Pin and AspectJ.

10.1.2.1 Runtime data collection using Pin

Pin performs in-memory instrumentation, which is a technique for injecting data collection code into “real” code in memory. Therefore, there is no need to change the build scripts, makefiles, etc. Pin provides APIs for instrumentation at different abstraction levels, from a single instruction to an entire binary module. It also supports callbacks for many events such as library loads, system calls, signals/exceptions and thread creation events.

Here, we present our Pin-based program that collects the call-graph at runtime and stores it to an output file. The user has to install pin libraries in order to compile and run our program. Note that this program is valid only for analyzing single-threaded programs. For multi-threaded programs, we have to collect the thread id of each function otherwise we cannot to extract a precise stack trace because multiple call events can interleave.

```c
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string.h>
#include <pthread.h>
#include "pin.H"

const char* cgOutputFile="callgraph.out";
FILE * out;
PINC_LOCK lock;

VOID onFunctionEnter(const char * funName)
{
    fprintf(out, "Enter %s\n", funName);
    fflush(out);
}

VOID onFunctionExit(const char * funName)
{
    fprintf(out, "Exit %s\n", funName);
    fflush(out);
}

VOID Routine(RTN rtn, VOID *v)
{
```

```c```
RTN_Open(rtn);

    // Insert a call at the entry point of a routine
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)onFunctionEnter, IARG_ADDRINT,
                   RTN_Name(rtn).c_str(), IARG_END);

    // Insert a call at the exit point of a routine
    RTN_InsertCall(rtn, IPOINT_AFTER, (AFUNPTR)onFunctionExit, IARG_ADDRINT,
                   RTN_Name(rtn).c_str(), IARG_END);

RTN_Close(rtn);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fclose(out);
}

/* ================================
* Main                                *
* =====================================================================
int main(int argc, char * argv[])
{
    // Open the file to write the call graph
    out = fopen(cgOutputFile, "w");

    // Initialize symbol table code, needed for rtn instrumentation
    PIN_InitSymbols();

    // Initialize pin
    PIN_Init(argc, argv);

    // Register Routine to be called to instrument rtn
    RTN_AddInstrumentFunction(Routine, 0);

    // Start the program
    PIN_StartProgram();

    return 0;
}

If we run the above Pin-based program, we shall get an output similar to the below output where Enter and Exit refer to entering and exiting a function, respectively.

Enter main
Enter run
Enter paint
Exit paint
Exit run
Exit main

10.1.2.2 Runtime data collection using AspectC

We also used AspectC for instrumenting systems based on the C language [12]. The Pin framework has some limitations on certain embedded hardware processors because it performs in-memory instrumentation. However, AspectC modifies the source code of the system under analysis by injecting data collection code into it. Thus, one disadvantage of AspectC is that we have to adjust the build process to use AspectC because it weaves aspect code only on the preprocessed code, that is, without any #include or ifdefs, etc.

Here, we include a sample AspectC program that was used to extract the time spent by functions that are called by the “execute” function, for example.

```c
#include <stdio.h>
#include <sys/timeb.h>
#include "logfile.h"

/* Print the time of functions called from the “execute” function */
before(): call($ $(...)) && infun("execute")
{
    struct timeb lTime;
    ftime(&lTime);
    long startTime = (long)(lTime.time*1000 + lTime.millitm);
    char output[100];
    /* Enter <func nam> <time> */
    sprintf(output, "Enter %s %lu", this->funcName, startTime);
    LOGFILEwrite(output);
}

/* Print the returned time of functions called from the “execute” function */
after(): call($ $(...)) && infun("execute")
{
    struct timeb lTime;
    ftime(&lTime);
    long startTime = (long)(lTime.time*1000 + lTime.millitm);
    char output[100];
    /* Exit <func nam> <time> */
    sprintf(output, "Exit %s %lu", this->funcName, startTime);
    LOGFILEwrite(output);
}

If we run the above aspect, its output should be similar to the below output. The third column of each line denotes the time in milliseconds. Thus, the duration for the “execute” function is 45-10 = 35 milliseconds.

Enter connect 10
Enter bind 12
Exit bind 16
Enter send 19
Exit send 35
Exit connect 45

10.1.2.3 Runtime data collection using AspectJ

In Chapter 4 and Chapter 5, we used AspectJ to collect runtime data [13]. Here, we include a sample AspectJ program that keeps track of classes which instantiate threads at runtime. Note that, in contrast to the above mentioned AspectC tool, AspectJ modifies the bytecode directly. Therefore, the source code of the system under analysis is never modified for data collection.

```java
import org.aspectj.lang.JoinPoint;

/**
 * Locate all classes that create threads
 *
 */
public aspect Thread_Creation extends TraceCollector {
    // Inject all calls to thread creation
    public pointcut callToThread() : call(java.lang.Thread..new(..));

    before() : callToThread() {
        printThreadCreation(thisJoinPoint);
    }
}

class TraceCollector {
    protected void printThreadCreation(JoinPoint jp) {
        // Print classes that matched this joinpoint.
        System.out.println(jp.getSourceLocation().getWithinType().getName());
    }
}
```

This small aspect program can be used to inject Java bytecodes in order to collect all classes that instantiate threads.

10.2 Analysis Tools

In this section, we briefly explain our tools that we use for analysis of the data extracted using the tools mentioned above.
10.2.1 Visualization of structural views using SAVE-LIGHT

The SAVE-LIGHT tool is used for visualizing hierarchical dependencies among modules. This is a lighter version of the SAVE tool mentioned in [187]. It is “LIGHT”, because in contrast to the SAVE tool, this tool does not require Eclipse and all necessary plug-ins to be installed and running, instead the SAVE-LIGHT runs in a web-browser and requires no installation. We have successfully installed the SAVE-LIGHT tool at a few of our customers’ site. One interesting aspect of this tool is that it is entirely decoupled from parsers and programming languages. Therefore, we can use any parser and feed the extracted data in a comma separated format (csv) format to the SAVE-LIGHT tool. We also used RPA’s lift and transitive closure operators on the extracted dependency data and exported the result to the SAVE-LIGHT tool for visualization and analysis.

![Figure 10-2: An example SAVE-LIGHT view.](image)

Figure 10-2 shows a sample output of the SAVE-LIGHT tool, where each box is a module (i.e., a directory), arrows denote dependencies between modules. If we click on a box, all its incoming and outgoing boxes are highlighted. If we click on “I”, it will show all modules, which are not in the current view, but use the children of the clicked box. Similarly, if we click on “O”, it will show all packages, used by the children of the clicked box. If we click on an edge between boxes, the tool shows all dependencies from the origin box to the target box, as shown in Table 10-1.

<table>
<thead>
<tr>
<th>Origin Folder</th>
<th>Origin File</th>
<th>Target Folder</th>
<th>Target File</th>
</tr>
</thead>
<tbody>
<tr>
<td>gov/nasa/gpf/cnas/...</td>
<td>EasConstants.java</td>
<td>gov/nasa/gpf/cnas/moclient/gui</td>
<td>ExpressionConstants.java</td>
</tr>
<tr>
<td>gov/nasa/gpf/cnas/...</td>
<td>SecurityWarningPanel.java</td>
<td>gov/nasa/gpf/cnas/moclient/gui</td>
<td>MainControlPanel.java</td>
</tr>
<tr>
<td>gov/nasa/gpf/cnas/...</td>
<td>ChangePasswordPanel.java</td>
<td>gov/nasa/gpf/cnas/moclient/gui</td>
<td>MainControlPanel.java</td>
</tr>
<tr>
<td>gov/nasa/gpf/cnas/...</td>
<td>PreferencePanel.java</td>
<td>gov/nasa/gpf/cnas/moclient/gui</td>
<td>NodeConstants.java</td>
</tr>
<tr>
<td>gov/nasa/gpf/cnas/...</td>
<td></td>
<td>gov/nasa/gpf/cnas/moclient/gui</td>
<td>MainControlPanel.java</td>
</tr>
</tbody>
</table>

Table 10-1: Dependencies from the origin folder to the target folder
“A” stands for abstract, which deserves further explanation. In many systems, we observed that, in the beginning of each source code file, there is one very high-level comment that summarizes the purpose of the file in one line. An example could be: “Abstract: This class connects to a remote client”. Because of the usefulness and abstract nature of such comments, we decided to pull these comments out of files and display them on a dependency diagram for analysis. For example, if we click on “A” of the gui package, the tool will display a table similar to Table 10-2. The SAVE-LIGHT tool has a predefined comma separated format (csv) format for importing such abstract information, which has to be, of course, extracted by parsing the source code of the system under analysis by considering the format used for defining an abstract.

Table 10-2: Snippet abstract information for files under gui

<table>
<thead>
<tr>
<th>File</th>
<th>Path</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>PassTheMessagesPanel.java</td>
<td>This class allows users to set which QAS messages to be forwarded automatically to EPS nodes.</td>
</tr>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>GraphicStatusIndicator.java</td>
<td>This class allows users to define orbital constraint limits.</td>
</tr>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>EDG3DSciC33ChannelDataFormatPanel.java</td>
<td>This class allows users to edit SAS Q Channel Data Format Parameters.</td>
</tr>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>EditIDRsPanel.java</td>
<td>This class allows users to download or modify a playback request.</td>
</tr>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>HelpDeskSelectionPanel.java</td>
<td>This class allows users to specify parameter limits for User Performance Data numeric values.</td>
</tr>
<tr>
<td>C:/gov/nasa/gphi/gphi/inoclient/gui/</td>
<td>PassTheMessagesPanel.java</td>
<td>This class allows users to set which QAS messages to be forwarded automatically to EPS pods.</td>
</tr>
</tbody>
</table>

10.2.2 Synchronized structure and behavior views using Dyn-SAVE

We developed the first version of the tool for visualizing sequence diagrams of the collected execution traces in [303]. Here, we give an overview of this tool using some screenshots. One appealing feature of this tool is the capability to synchronize both structural and behavioral views. By collapsing the hierarchies of a structural view, the sequence diagram can automatically be navigated at different levels of abstraction, as illustrated below.

![Figure 10-3: Structural dependency between two subsystems.](image-url)

A sample sequence diagram, visualized using the Dyn-SAVE, is shown in Figure 10-4. An appealing feature of the Dyn-SAVE tool is that we can synchronize both structural and sequence diagrams, meaning that if we expand, for example, the System box of
Figure 10-3, the sequence diagram will automatically react to this even and will also expand as shown below. This feature allows the user to navigate large traces because the user can use the hierarchy information to collapse the structural view, which also collapses the corresponding sequence diagram, as shown in Figure 10-5 and Figure 10-6.

![Component Sequence Diagram](image1)

Figure 10-4: Sequence diagram using Dyn-SAVE for Figure 10-3.

![Component Sequence Diagram](image2)

Figure 10-5: The view after expanding the “System” of Figure 10-3.
10.2.3 Specification and execution of Colored Petri nets

In Chapter 4 and Chapter 5, we used Colored Petri nets for discovering components connector views and sequence diagrams from execution traces. In Chapter 4, we used the Exspect COTS tool for specifying and executing Colored Petri nets \[1\]. Later, we developed our own specification language and a supporting tool using Java and SQL. In this tool, the user can specify a Colored Petri net using XML as a specification language. The syntax of the specification language and implementation details of our tool are well explained in a master’s thesis performed at Fraunhofer. We request the reader to read \[180\] for further details.

10.2.4 Searching for friends in a vector space

Here we briefly explain our tool for calculating text-based similarity among software artifacts. Our tool is being used in several architectural analysis projects.

The first step for calculating the degree of similarity among software artifacts is to build an index. In Section 10.1.1.4, we mentioned that we use Lucene framework for building an index. Lucene offers APIs for accessing the index. For example, we can programmatically query Lucene to list all files (a.k.a. documents) that contain a given word (a.k.a. term). In a similar way, we can also obtain the term frequency (tf) of a given term in a given document.

The second step is to build a term-doc matrix, as proposed by Sir Salton et al. \[249\]. We used Lucene’s APIs for building a term-doc matrix, see Figure 10-7.
In the existing literature, there are several formulas for assigning weights to the cells of a term-doc matrix. We list here a few formulas that are supported in our tool:

1. We can build a Boolean matrix with 1’s and 0’s: if a term is present in a document we can assign 1 to the cell, otherwise 0. The effect of this formula is that all terms of a document are treated equally important.

2. We can use the term-frequency (tf), which counts the number of times a term is present in a given document. The effect of this formula is that the terms with high tf are given more importance than those with low tf in a document.

3. We can use the inverse document frequency (idf), which is a logarithmic ratio of the number of documents in the index to the number of documents that contain a given term. Intuitively, it means that if a term is present in all documents then its weight will be zero. The effect of this formula is that if a term is present in many documents it is not so important.

4. We can combine tf and idf by just multiplying both weights; it is famously denoted as tf-idf. The effect of this formula is that if a term is present in only a few documents, by scaling it with its term frequency, the term becomes more important than other terms in the corresponding document.

In our tool, all the above four formulas are options for the user to choose. One of the important issues to note here is concerned with the memory requirements of a term-doc matrix. When we built a term-doc matrix for several real-world systems, we observed that the matrix becomes huge, and often cannot be stored in RAM, resulting in out-of-memory errors. However, we also observed that around 95% to 98% of the matrix cells were filled by zeros. Equivalently, a given term, in general, occurs only in a few documents. Therefore, we decided not to store zeros unnecessarily. Our implementation stores columns (or rows) of a term-doc matrix as a collection of sparse vectors with keys and corresponding values stored in two arrays. That is, each vector is implemented as a pair of arrays, in which the key array contains the row numbers where the value is non-zero, and the value array contains the value of the cell.
One practical benefit of this sparse vector representation is that we are able to scale to large matrices (e.g., 150,000 rows by 65000 columns), which means that the current implementation has been tested to handle software artifacts with 65000 files and 150 000 terms without sacrificing runtime performance. On the other hand, our implementation has become slightly complex because vector operations such as addition, multiplication, scalar or dot product are not straightforward due to the fact that we need to match-up positions of row (or column) number of pairs of vectors before performing operations on the values of vectors.

The third-step allows the user to search the term-doc matrix as follows. The main purpose of building a term-doc matrix is to allow the user to search for entities “similar” to the given entities. In our context, an entity can be either a term or a document. Our notion of similarity is based on co-occurrences. That is, if two terms occur together in several documents, then we consider them to be similar to each other; if two documents share many terms, then we consider them to be similar to each other. Here, we provide a geometric interpretation of similarity before introducing features of our tool.

Matrices can be interpreted not only algebraically but also geometrically. For example, we can interpret each column of a term-doc matrix with n rows and m columns as a geometric point in n dimensional space; each row is a point in m dimensional space, see Figure 10-8, for an example 3 x 3 term-doc matrix.

![Figure 10-8: Geometric views of a term-doc matrix.](image)

This geometric representation of a term-doc matrix convey us that if two documents do not share terms, then they will be orthogonal (i.e., 90 degrees) to each other; if two documents share a large number of terms, then they will be pointing in the same direction.

Elementary Linear Algebra offers algebraic formulas to compute the angle between two vectors \( v_1 \) and \( v_2 \) as follows [274]:

\[
\cos \theta = \frac{v_1 \cdot v_2}{|v_1||v_2|}
\]

In the numerator, the scalar product (a.k.a. the dot product) of given vectors are computed. In the denominator, the vectors are normalized. Note that, in our context, the range of \( \cos \theta \) is between 0 and 1, because cells of vectors are non-negative. When
$\cos \theta$ is close to 1, then the vectors representing the entities are almost identical, if it is close to 0 they are different.

We implemented this formula, which allows the user to search for entities similar to a given entity. For example, the user can query for terms similar to term 1, our tool will return a ranked list of all terms, in that the first ranked term has the highest cosine value with the query term.

In one of our case studies with Biofortis in Maryland, we had to analyze their system with respect to its compliance to the CFR Part 11 standard [48]. One part of this standard is related to requirements dealing with users’ account management. For example, requirements such as the following are part of the CFR Part 11 standard:

- The system should save the history of all passwords
- Passwords must be unique to users
- All login and logout must be recorded
- A new user must change his password first time
- Passwords must expire periodically

We used our vector space based search tool to automatically search for concepts related to users’ password management regulations. We can confirm in Figure 10-9 the system under analysis indeed contains the password concept and many of the requirements of the CFR Part 11 are implemented, without reading several source code files manually.

![Figure 10-9: Friends of the “password” concept.](image-url)
10.2.4.1 Searching using vector addition

In many cases, during our architectural analysis activities, we need the capability to search for entities that are friends of one or more entities. We implemented this capability using the standard vector addition operator. For example, the user can search for friends of the “password” or “login” term. In our implementation, we add the row vectors of “password” and “login” term stored in the term-doc matrix. Thus, our query is just one vector, which can be compared with all term vectors by using the $\cos \theta$ formula introduced above.

10.2.4.2 Searching using vector negation

In many cases, we have come across the need for “Not like these” capability. The motivation for this capability can be illustrated using a concrete example of the SNAS system. In Section 6.5.6.2, we discussed clones among files of the database abstraction layer due to the “boiler-plate” code needed for accessing a database. Figure 10-10 shows, for example, the top 20 friends of a DAO layer file DbSic.java where Sic stands for the status indicator code. We noticed that almost all friends of this file are also part of the DAO layer of the snif subsystem. Thus, if we are seeking friends of this file but not like other DAO files, we cannot easily find them. In order overcome this limitation, we implemented the NOT operator based on vector negation concept.

The user can search DbSic.java NOT DbRealtimeConnection.java. The effect is that a new query vector is automatically formed in such a way that it is closer to DbSic.java but orthogonal to DbRealtimeConnection.java. This means that the new query vector will also be almost orthogonal to the friends of
DbRealtimeConnection.java. As a consequence, it is expected that the search result will contain friends of DbSic.java but not like DbRealtimeConnection.java and its friends, which is indeed true as shown in Figure 10-11. We noticed that the NOT operator essentially removed all friends of DbSic.java from a database point of view. The search result now pulled the friends of DbSic.java from different parts of the SNAS system such as GUI models, panels, beans, and even XML data files. This would not have been possible without using the NOT operator. We had discussed this output with a SNAS developer, who agreed that this list is a good summary of files related to the SIC concept.

To implement the NOT operator we used the Gram-Schmidt orthonormalization process as proposed by Dominic Widdows [306]. It is worth noting that Widdows applies the NOT operator after reducing the dimension of the vector space. That is, Widdows transforms the term-doc matrix into a smaller matrix in such a way that the angle between column (or row) vectors is preserved to a large extent using random projection [146]. Geometrically, Widdows projects each vector into a low dimensional vector space in such a way that this transformation preserves angle between pairs of vectors to a large extent. In our implementation of the NOT operator, we do not perform such transformation, and work in the original vector space of the index. In the future, we plan to employ dimension reduction as performed in [306], [25], and [147] and compare the retrieval and ranking effectiveness without dimension reduction.
10.2.4.3 Visualizing all artifacts in two dimensional Euclidean space

By computing the angle between vectors, we can measure their degree of similarity, as introduced above. If there are, say m documents, we can compute a similarity matrix among all pairs of documents. For large systems, visualizing such a similarity matrix is a challenge. In our tool, we used a multi-dimensional scaling technique that helps us in assigning a geometrical point in a two dimensional Euclidean space for each document, using the similarity matrix as the input [30]. The geometrical coordinates of each document are assigned in such a way that if two documents have a high similarity in the similarity matrix, then they will be placed geometrical close together. One key benefit of assigning two dimensional coordinates to each document is that we can visualize all documents in one picture. At Fraunhofer, the VQI tool was developed for visualizing data points in general. We successfully reused this general capability to visualize all source code files in one picture, as shown in Figure 10-12. We assigned colors to files based on their root folder in the hierarchy of directory structures. All documents which belong to the same root folder were assigned the same color. In this example, we can visualize clusters of files in a two dimensional Euclidean space.

Figure 10-12: Visualizing all documents in two dimensions.

10.3 Closing Remarks

In this chapter, we offered a brief overview of several tools we developed to support reverse engineering. Our tools help in both extraction and analysis of data from
software artifacts, including static and dynamic data. First, we presented several of our extraction scripts and programs, which could be reused by others. Second, we presented our analysis tools for visualizing structural and behaviors views. Finally, we presented our tools for searching the code base to automatically build concept models and finding files “similar” to a given list of files. Our search tool is strongly rooted in Linear Algebra. We have implemented vector additions and negations for supporting sophisticated search of large code bases. Our future work prospects include the development of an analyst workbench, which will allow an analyst to plug and play different reverse engineering tools on demand. The “ToolBus” architectural style could be explored further to integrate our chain of tools in an indirect way by sending and receive messages on the bus [22].
Chapter 11
Epilogue

In the introduction, we discussed the importance of software architectures for developing, testing, and evolving high-quality software-based systems. Throughout this thesis we have highlighted architectural issues and challenges faced in industry. We introduced two scenarios which are the central focus of the thesis. The first scenario deals with systems that went through a software architectural design phase, but for which it is difficult to analyze whether or not the specified architectural rules are followed by its implementation. The second scenario deals with systems that did not go through an explicit architectural phase, making it difficult to see the implicit architecture that is hidden in source code, and to evaluate quality properties such as testability, performance, and maintainability. We also discussed how we developed an approach in response to the need to analyze these two types of systems, and how we applied the method to several real-world industrial systems. In this Epilogue, we revisit the research questions outlined in the introduction and draw conclusions and discuss open issues, which could be explored for future research.

11.1 Quality by Design Instead of Quality by Tests

Our goal was to offer recommendations so that quality can be built-in during architecture design, instead of being tested-in. Based on architectural analysis of several real-world systems, we developed a large body of architectural knowledge. We presented a list of recommendations that are useful to prevent architectural problems such as violations of architectural rules, low testability, performance, and maintainability risks. In Chapter 9, we investigated the first research question:

- RQ1: How can we avoid problems such as architectural violations at the source code level, testability, performance, and maintainability risks in the very first place?

We emphasize the importance of organizing the build process in such a way that it reflects the modular structure of the system under analysis. “Open and relax” build processes invite violations of architectural rules. In order to avoid behavioral violations, we recommend interface-oriented architectures so that all components plug-in into a framework in the same way. In addition, we demonstrated that by standardizing the interfaces of components their look-and-feel can be improved. This also facilitates maintenance because developers can easily get familiar with different components of the system, because of standardized interfaces and interaction protocols in place.

For improved testability, we recommend a collection of practically applicable principles, such as a database abstraction layer, an OS abstraction layer, a GUI abstraction layer, and making some internal details public, avoid performing critical operations inside constructors and static methods because they cannot be overridden,
and avoid using too much encapsulation because it is difficult to create a chain of objects for testing purposes. We also recommend the usage of architectural rules as a trade-off for compromising engineering principles for improved testability.

In general, engineers appear to be not so attracted by maintenance risks, but we found that they pay a lot of attention when we talked about testability risks in their architectures. It is worth noting that several of the architectural recommendations for improved testability implicitly offer recommendations for improved maintainability, too. As demonstrated in this thesis, for example, separation of concerns not only facilitates testing but also program understanding and maintenance activities. Thus, in practice we use testability principles as drivers for building maintainable systems.

For improved performance, we recommend a collection of practically applicable principles, such as the usage of transfer object design pattern to overcome limitations of remote procedure calls, the usage of the reactor design pattern to handle multiple clients accessing a server in the client-server architectural style, and the usage of the connection pool design pattern for accessing a database, and paying attention to threading models.

One open issue is that our recommendations did not cover other quality properties such as security and usability. More work is needed to populate a rich collection of practically applicable recommendations which can be eventually be evolved into a handbook of architectural knowledge base for developing high-quality software-based systems.

11.2 Compliance with Specified Architectural Design Rules

The high-level research question of the first scenario:

RQ2: How can we analyze that the implementation conforms to the specified architecture?

In this thesis, we refined this question and focused on structural and behavioral constraints of architectural styles. That is, from the style definition we derived a set of constraints and applied both static and dynamic analysis to check whether or not the implementation follows architectural styles. This led us to the following questions:

- RQ2.1: How can we statically analyze that the specified structural rules of architectural styles are followed by the implementation?
- RQ2.2: How can we dynamically analyze that the specified behavioral rules of architectural styles are followed by the implementation?
- RQ2.3: How can we combine static and dynamic analyses for compliance checking of static and behavioral rules of architectural styles?

We addressed these questions in Chapter 2, Chapter 4, and Chapter 5. We developed an elaborated method for analyzing architectural rules derived from architectural styles. In addition to statically analyzing structural constraints of architectural styles, our method has the capability to be deployed at runtime for monitoring a running system and
checking whether or not software components, which can even be plugged-in and plugged-out at runtime, follow behavioral constraints.

In Chapter 2, we concentrated on RQ2.1 in detail and investigated the applicability of the method for analyzing the structural constraints of the publisher-subscriber style. That is, we evaluated our method by analyzing the NASA’s CFS product line implementation, which is based on the publisher-subscriber style. Our approach was based on static analysis of the CFS source code. We extracted module views statically and compared them to the specified structural constraints of the publisher-subscriber style. It is worth noting that the CFS continuously undergoes rigorous code reviews and testing. Nevertheless, using our method, we detected a few structural violations that escaped reviews as well as testing. The detected violations were reported to the CFS team and were subsequently acknowledged and fixed by the team. Our conclusion was that the CFS is a well-engineered system, and one of the reasons for very few violations of architectural rules is the fact that its build process is organized in a component-oriented way. Therefore, it is not easy for developers to introduce violations.

In Chapter 4, we concentrated on RQ2.2 in detail and investigated the applicability of the method on the pipe-and-filter style. That is, we evaluated our method by analyzing the Ricoh’s MFP prototype implementation, which is based on the pipe-and-filter style. Our approach was based on dynamic analysis of the MFP. That is, we ran the MFP in a monitoring mode – with probes injected using aspect-oriented programming – and collected runtime data and verified whether or not the running system satisfies the specified behavioral constraints of the pipe-and-filter style, which was modeled using Colored Petri nets as the underlying formal language. It is worth noting that MFPs allow software components to be plugged-in and plugged-out at runtime. Therefore, it was critical to develop an approach to monitor and analyze behavioral constraints while the system was running because off-line analysis of runtime data would have led to too late detection of misbehaving components and removing them to avoid further problems.

In Chapter 5, we concentrated on RQ2.1, RQ2.2, and RQ2.3 in detail and investigated the applicability of the method for analyzing the structural and behavioral constraints of the publisher-subscriber style. That is, we evaluated our method by analyzing the implementation of NASA’s GMSEC product line, which is based on the publisher-subscriber style. From the style definition, we derived a set of reusable analysis questions and showed that the questions related to the structural constraints of the style were answerable using static analysis, whereas the questions related to the behavioral constraints of the style were answerable using dynamic analysis. We combined static analysis with dynamic analysis in order to instrument the system at the right locations to avoid collecting too much runtime data and minimizing its overhead. We applied Colored Petri nets to formally model the constraints of the publisher-subscriber style. In this endeavor, we detected a high-priority bug caused by the violation of behavioral constraints of the style. This bug was accepted by the GMSEC team and was fixed by them.

From a method point of view, in Chapter 2, Chapter 4, and Chapter 5, we applied and promoted the idea of “architectural style driven reverse engineering”.
We combined static and dynamic analysis for efficiently collecting relevant runtime data. By combining the two types of analysis, we were not flooded with megabytes of runtime data. We used aspect oriented programming to inject code for collecting runtime data. We used Colored Petri nets (CP-nets) as the formal language to precisely model the constraints of architectural styles. We fed the collected runtime data into our CP-nets which recognized pre-planned constrains, as shown in Chapter 4 and Chapter 5. We constructed CP-nets for recognizing the pipe-and-filter style and the publisher-subscriber architectural style of the running system, and checking the constraints of the styles. This set-up allowed us to detect violations of architectural rules.

We agree that there are other architectural styles in the existing literature which were not covered in this thesis. However, we obtained evidence to support that our experiences could be used for an analysis of other architectural styles. This is because there are some common properties among styles as discussed below.

One common property among all architectural styles is that each of them has a collection of pre-defined structural and behavioral constraints. From definitions of styles, we could identify the types and roles of each components and connectors, and derive constraints on them. Using static analysis strategies, as explained in Chapter 5 and Chapter 6, we could bridge the abstraction gap between the elements of architectural styles and the code elements of the system under analysis. Once the code elements are located, we could instrument them and collect minimal and necessary runtime data to analyze the behavioral constraints of the style.

Another common property among all architectural styles is that their runtime traces are highly likely to be inter-leaved. For example, if we consider the client-server architectural style, when a client is in the process of creating a connection to the server, the server can be in the state of serving other clients [205]. Thus, the traces are inter-leaved. We have highlighted this fact in Chapter 4 and Chapter 5 and noted that the traces of the pipe-and-filter style and the publisher-subscriber style are also interleaved. We elaborated that CP-nets are capable of recognizing pre-planned constraints in interleaved traces. Thus, we believe CP-nets are useful for an analysis of other styles, too. It is also worth noting that the construction of CP-nets is an investment. However, CP-nets are reusable for all systems that are based on the same architectural style and implemented in the same way. This is often the case in the context of software product lines.

One key take-away point here is that “architectural style driven reverse engineering” enables the reuse of analysis questions, techniques, and technical infrastructures for all systems based on the same architectural style, as illustrated in this thesis using case studies. In addition, we can focus the reverse engineering activities to architectural constraints of styles, and ignoring or filtering out irrelevant information. As a consequence, reverse engineering approaches based on architectural styles are capable of scaling to large systems.

11.2.1 Open issues on verification of architectural rules

We identified a number of issues related to the verification of architectural rules, of which we will mention the most prominent ones which require further research.
11.2 Compliance with Specified Architectural Design Rules

The first issue is related to parsing of source code. As discussed in several chapters of the thesis, parsing of real-world systems is a painful, yet unavoidable task for reverse engineering. To overcome this pain, currently we are evaluating parser technologies that are easy to customize for different language dialects, as proposed in [173], [174], [164], [159], and [35].

The second issue is related to a safe plug-and-play in the context of systems based on software components. The issue is related to the problem: *Do equivalent components behave identically?* This problem is of high interest for all systems whose architecture allows plug-and-play of components, similar to the CFS as well as the GMSEC. We explain the issue using the CFS and the GMSEC examples: In the CFS case, the OS abstraction layer has several implementations of the same OS independent abstract interface, where each implementation of the interface corresponds to one OS type. Users can select one preferred implementation of the OS abstraction interface corresponding to the OS of their interest. Similarly, in the GMSEC case, the middleware abstraction layer has several implementation of the middleware-vendor independent abstract interface, where each implementation implements the interface using one specific vendor’s APIs. Users can select one preferred implementation of the middleware abstraction corresponding to the vendor of interest.

For a safe plug-and-play of software components, users must be convinced that the behavior of alternative implementations of the same abstract interface is indeed compatible. In this thesis, we approached this problem using dynamic analysis. In Chapter 5, using our dynamic analysis approach, we detected a high priority bug which was due to an inconsistency among the different implementations of the middleware abstraction layer. That is, one implementation of the publisher-subscriber interface allowed the user to subscribe to the same message without an intermediate unsubscribe, whereas the other implementation did not allow this behavior. One inherent limitation of any dynamic analysis based approach is that we cannot easily generalize and draw conclusions on scenarios that were not executed [70]. Therefore, we cannot fully depend on dynamic analysis to show that all components which implement the same interface indeed exhibit equal behavior.

At the time of finalizing the thesis, we developed a lightweight technique that helped us in statically extracting a data model from each implementation function of abstracted interfaces. This data model is a suite of multi-relations between each function and its return codes with the weight represents the count of the number of times each return code is returned from the function. We wrote relational queries that helped us in automatically detecting deviations such as missing return codes or inconsistent return codes among different implementations of the same interface. We applied this technique on the NASA OSAL layer, which abstracts different OS types (see Chapter 2 and Chapter 3), as well as on the NASA GMSEC’s Middleware abstraction layer (see Chapter 5), which abstracts different middleware types. We were surprised that this light-weight heuristics technique statically found several inconsistencies among different wrappers that wrap the underlying OS (in the OSAL case) and middleware (in the GMSEC case).

It is worth noting that these detected inconsistencies due return codes were mostly related to error handling. For example, the function `OS_BinSemCreate`, which is responsible for creating a binary semaphore, returns `OS_ERROR` under Linux but the
corresponding RTEMS implementation returns OS_SEM_FAILURE, see Table 11-1. We checked that these two return codes are two different integer constants. Therefore, these two implementations are not compatible to each other. Similarly, we also found differences in the return codes among the GMSEC wrappers for different middleware technologies. For example, the Subscribe method of the WebSphere middleware wrapper returns GMSEC_INVALID_CONNECTION if the connection is not valid, but the GMSEC’s own proprietary middleware wrapper based on sockets returns GMSEC_OTHER_ERROR if the connection is not valid when the user calls the Subscribe method.

Table 11-1: Some behavioral inconsistencies among the OS wrappers

<table>
<thead>
<tr>
<th>Filename</th>
<th>Function name</th>
<th>Linux Return code</th>
<th>RTEMS Return code</th>
<th>VxWorks6 Return code</th>
</tr>
</thead>
<tbody>
<tr>
<td>osapi.c</td>
<td>OS_BinSemCreate</td>
<td>OS_ERROR</td>
<td>OS_SEM_FAILURE</td>
<td>OS_SEM_FAILURE</td>
</tr>
<tr>
<td>osapi.c</td>
<td>OS_BinSemTimedWait</td>
<td>OS_SEM_FAILURE</td>
<td>OS_SEM_FAILURE</td>
<td></td>
</tr>
<tr>
<td>osapi.c</td>
<td>OS_CountSemCreate</td>
<td>OS_ERROR</td>
<td>OS_SEM_FAILURE</td>
<td></td>
</tr>
<tr>
<td>osapi.c</td>
<td>OS_CountSemTimedWait</td>
<td>OS_SEM_FAILURE</td>
<td>OS_SEM_FAILURE</td>
<td></td>
</tr>
</tbody>
</table>

In both cases, the developers acknowledged these surprising findings, and included these inconsistencies among return codes of wrappers into their issue-tracking systems. One consequence is that such behavioral discrepancies among different wrappers might lead to subtle failures if one switches from one wrapper to another. These types of subtle failures are difficult to detect during testing simply because it is difficult to put the system into the desired state to simulate the errors that are discussed above.

We believe this problem requires further research. We suggest one more possible direction for solving this problem. By using the ideas of symbolic execution (e.g., [155] and [44]) we could extract a logic formula (a.k.a. model) for each implementation of the interface. We may be able to compare the collection of logic formulas in order to prove or disapprove the equivalence of behavior of several implementation of the same interface as follows. If there is an input that satisfies one of the formulas and simultaneously violates the other formulas, then we can conclude that different implementations of the same interface are not consistent. If there is no such input, then we can conclude that all implementations of the same interface are equivalent. We also need a method to extract resource constraints (e.g., memory, number of files opened, etc.) and timing aspects of different implementations of the same interface. In many operating systems, this data is already collected at runtime. For example, under Windows, the task manager collects the number of processes, threads, the number of bytes written to files, sockets, etc. We believe this data could be analyzed to draw conclusions on resource consumption of different implementations of the same interface.

The third issue is related to the completeness of dynamic analysis. By completeness, we mean making sure that all instrumented code elements, used for runtime data collection, are activated at least once. In our case studies, we read the systems’ user manual, requirements document, and existing test cases, if any. Based on the acquired knowledge and test cases, we triggered the system. This takes effort because we have to
read documents and gain some domain-level understanding. It would be useful if we can construct a test suite that will cover all instrumented code at least once. Thus, an open issue is: Under what set of scenarios should we be running the system to cover all injected code? At this point, we imagine that some advanced static analysis could help us in addressing this issue.

The fourth issue is related to resolving the architectural violations that were detected in the implementation of the system under analysis. In this thesis, we have not discussed techniques for resolving detected violations automatically. Some of our project collaborators have expressed interest in evaluating and applying the state-of-art software renovation technologies that hold potential for resolving the detected architectural violations automatically [167], [36], [38], and [298]. We propose to investigate this topic in collaboration with researchers at the VU University Amsterdam.

11.3 Discovery of Software Architectures

In the second scenario, we focused our research on discovering the software architecture, which is only implicitly present in the source code of the system under analysis because no architectural design phase was explicitly conducted. This scenario has given rise to the following question:

- RQ3: How can we efficiently discover software architectures and analyze quality properties, in particular, testability, performance, and maintainability, without reviewing inhibitive many source code files?

We motivated the fact that, in our experience, even well-engineered systems have a “spaghetti” like structure, just because all concerns such as persistence, error handling, logging, licensing, etc. are part of the reverse engineering model of the system under analysis. Therefore, we developed a method for analyzing the implemented system with respect to one concern at a time so that we see the hidden “lasagna” and analyze its quality properties such as testability, performance, and maintainability. In addition, several of those systems also had a large amount of test code, whose architecture is also implicitly present in the test code. Our task was to make the architecture explicit, evaluate quality properties, and identify quality risks.

Thus, we refined the above question as follows:

- RQ3.1: How can we analyze the systems’ implementation and discover architectural views for various concerns, such as persistence, GUI, OS variants, etc.?
- RQ3.2: How can we identify architectural design decisions that facilitate or impede testing?
- RQ3.3: How can we identify implemented architectural design decisions that attribute to performance risks?
- RQ3.4: How can we assess the maintainability of the test code?
In Chapter 6, we covered RQ3.1, RQ3.2, and RQ3.3 in detail. We developed an elaborated method for discovering the software architecture of the system under analysis. We presented a four-dimensional model for architecture discovery and analysis of quality properties, see Section 6.4. We showed that our method is flexible, in that analysts can decide, based on their goals, the concern of interest and relevant quality properties to evaluate. The premise of our method is that architecture decisions are inspired and influenced by the external entities that the software system makes use of. Examples of such external entities are COTS components and the programming language libraries. Traces of these architecture decisions can thus be found in the implemented software and manifest in how the software system uses such external entities. We developed a knowledge base of external entities and discovered the implemented software architecture using dependencies to external entities by just reviewing a minimal set of files.

In Chapter 6, using our knowledge-based reverse engineering method on the relatively large (~600 KLOC) NASA Space Network Access System (SNAS) system, we explained how the independent analyst discovered several architectural insights by reviewing less than 4% of the 1578 source files. Examples of architectural insights are a) that the implemented architecture of the SNAS is based on a distributed client-server architectural style, b) that the distributed subsystems exchange data by sending and receiving objects using the transfer object design pattern [5], c) that each subsystem of the SNAS has a dedicated layer for handling the persistence concern, and d) that the GUI subsystem is based on an event-driven architecture. In addition, with the help of our knowledge base, several architecturally relevant performance related constructs were discovered including the usage of a) a database connection pool design pattern in order to overcome the performance overhead of frequently creating and deleting database connections [10], b) the reactor design pattern in order to reduce the overhead of frequently creating and deleting threads for each client connection in a client-server architectural style [252]. Some testability problems due to a weak separation of GUI concepts with core logic were also discovered as well as some performance risks due to threading models. The analysis, detected problems, potential risks as well as concrete solutions and risk mitigation strategies were reported to the SNAS team.

In Chapter 3, we covered RQ3.2 and RQ3.4 in detail. We presented how a suite of reusable questions can be used for assessing the unit test code from a maintainability point of view. We analyzed the characteristics of architectural decisions that impede or facilitate unit testing. Our key findings are that a) programming to abstract interfaces facilitate unit testing because light-weight mock implementations could be bound to the interfaces of dependent modules, thereby, the module under test can be independently tested, assuming the correctness of dependent modules, b) if a function returns the same return code for different input scenarios, its test program becomes complex since it is difficult to know which path was indeed taken by the given input, c) some internal details of modules have to be made public for improved testability, d) a dense graph of module dependencies does not imply poor design quality and low testability, and e) for improved maintainability of test code, it is important to define an architecture for testing, too. That is, for example, there should be a clear strategy on how to unit test each module. Ideally, the test architecture reflects the software architecture of the system under test.
To sum up, we obtained evidence to support the claim that testing can be facilitated by explicitly taking care of testing related issues at the software architecture level. We have highlighted the fact that by applying basic software engineering principles as well as by opening up some internal details of modules to public, it is possible to unit test each module without running the whole system, and also without access to special operating systems, hardware, databases and GUIs. We recommended verification of architectural rules as a trade-off for relaxing some software engineering principles so that if other modules misuse such “open points”, we could detect them. See Sections 3.7 and 6.5 for more details on an in-depth analysis of testability and architectures.

11.3.1 Open Issues on Software Architecture Discovery

We identified some open issues of our architecture discovery and analysis method, which are discussed below as potential topics for further research.

The first issue is related to requirements and architectures. If the system under analysis has no traceability links in place, connecting the requirements to the extracted architecture is a non-trivial challenge. We need such capability for requirements-oriented reasoning, if, for example, we have to evaluate how the implementation separates various domain concepts. Thus, we believe this issue deserves further research. The ideas of the similarity tool and analysis of words used in source code, as discussed in Chapter 10, could be explored further for solving this issue. We could exploit hard-coded strings used in log statements, constants, or in exceptions, and compare them with texts present in requirements document. The ideas of extracting business logic from source code could be explored further for modernization of software systems. Modernization could be, for example, migration of a system to modern frameworks and/or programming languages [296].

The second issue is related to evaluating the testability of web-based systems. In general, web-based systems have some special challenges such as variability due to different types of browsers, distributed computation on web browsers, web servers, application servers, databases, and/or back-end servers. Web-based systems are often implemented using several languages, frameworks, etc. In addition, there is a challenge of security (e.g., SQL injections) and privacy issues as well as managing several user sessions and transactions. Given this complex nature of web-based systems, we need further research on understanding and characterizing architectural design decisions that facilitate or impede unit testing of individual modules.

The third issue is related to the quantitative models of performance. In Chapter 6, we offered some insights on architectural decisions that influence performance. We believe quantitative models based on the discovered architecture would help in systematically evaluate performance. For example, in the SNAS architecture, the implementation of the transfer object design pattern used several communication ports to transfer various types of objects in order to minimize the waiting time of objects on communication channels before being picked-up and processed. The challenge is on how to model this architecture and perform simulation studies to predict several parameters, for example, what is average waiting time of an object if we had only one port instead of four ports. What-if analysis could be performed if we could develop analytical performance models of software architectures. This issue deserves further research in our opinion.
11.4 Organizational Aspects

Our third research area was related to the understanding of architectural aspects and how they influence the implemented architecture. To this end, we formulated the following questions:

- **RQ4.1**: How are the implemented architectural decisions related to business goals of organizations?
- **RQ4.2**: Can we use the work assignment relation to identify and reason about architectural issues that impact understandability and maintainability of systems?
- **RQ4.3**: Can we use the work assignment relation between developers and files to understand the modular structure of the implementation?

In Chapter 2 and Chapter 5, we covered RQ4.1. Both studies revealed that software architectures bridge the business goals and technical activities of organizations. For example, in Section 2.3.2 and in Section 5.4.6, we explained how high-level business goals were addressed by selecting the appropriate architectural style. In both the CFS and GMSEC case study, even non-technical stakeholders such as project managers and product leads, are well aware of the importance of key architectural decisions, such as a software bus, the publisher-subscriber architectural style, a OS abstraction layer, a middlelayer abstraction layer, for achieving the business goals of the organization. Thus, given the importance of architectures in fulfilling the business goals of the organization, we recommend organizations to check compliance with architectural rules in the implementation of the system under analysis.

In Chapter 6, we covered RQ4.2. This study revealed that we can use the work assignment relation between developers and the files they worked on to reason about architectural issues that affect understandability and maintainability. For example, the sdif subsystem of the SNAS has its own database interaction architecture, its own logging strategy, and its own utilities. We analyzed the developers-files relation and found that there is hardly any overlap between the developers of the sdif subsystem and other subsystems. Further discussions with the development organization confirmed that this subsystem was developed by a different development crew from a different contractor. This offered some potential reasons for differences in common look-and-feel of the way the subsystem is architected. If we had not analyzed this developers-files relation, we would not have obtained this insight. This study also indicates that if different subsystems have to use the same concern, for example, a database, the architecture must define standardized interfaces for all teams to follow. Otherwise, differences in look-and-feel are inevitable, thus affecting maintainability.

In Chapter 8, we covered RQ4.3. We developed a simple process for reconstructing an architectural view, called ownership architecture, which captures the relationship between the developers and the subsystems they worked on. We applied the proposed process on two “clone-and-own” variants of Hitachi’s engine control systems. We identified who worked on which part of the ECS. This knowledge is useful for organizing the interviews and asking the right questions to the right people, amongst
11.4 Organizational Aspects

other purposes. We identified the owners/experts at different levels of abstraction (e.g., subsystem, component level). We validated our approach by discussing with project managers. This study indicates that the ownership architecture is a good indicator of the implemented architecture of an existing product. That is, using developers-files relation we could identify potential subsystems of the system. When developers are shared between subsystems, the study indicates that dependencies exist between such subsystems. We observed the same characteristics even if subsystems are communicating indirectly using intermediate connectors, such as sockets or queues. However, systems based on the publisher-subscriber architectural style let developers work on their modules independently to a large extent, because the integration of components is taken care by the software bus.

11.4.1 Open issues related to organizational aspects

We identified some open issues, which might be of interest for future research.

The first issue is related to the understanding the characteristics of architectural violations in the context of multi-site development. Most of our work focused on analyzing systems developed in only one site. Thus, we are not able to characterize the nature of architectural violations of multi-site development. More work is needed in this direction.

The second issue is related to the cost and benefit of investing in architecture-based reuse. In our study, we have not discussed the economic benefits in investing in flexible architectures. In [106], we introduced a simple, yet powerful, economic model in the context of a product line. This model did not include architectural aspects because it was intended to be a conceptual economic model to discuss with management. We believe this model could be explored further, in conjunction with emerging results (e.g., [226] and [73]), by taking into consideration the investment in creating flexible architectures and reusable components.

The third issue is related to understanding the relationship between quality problems and organizational collaboration models, especially in the context of software reuse. In [163], we reported that response time issues in a software component developed for reuse was mainly due to the lack of an appropriate collaboration model between the teams that reuse the components and the team that developed the component. In general, teams that develop reusable components cannot predict how their components will be used by other teams. In our opinion, there should a simple collaboration model that facilitates exchange of feedback, test cases, performance data, etc. among different teams. We need to explore further to better understand the relationship between quality issues and organizational structure and collaboration models, similar to the work reported in [213].

The fourth issue is related to identification of subsystems by combining source code dependencies with developers-files dependencies. There is a lot of research on identifying potential experts using source code dependencies. However, very little research explored the benefits of developers-files relation for discovery of architecture as well as reasoning about detected architectural issues. Thus, it would be interesting to combine both source code dependencies with the developer-files relation for reverse engineering purposes.
Summary

In this thesis, we developed a *practically inspired* approach for architectural analysis of implemented industrial systems. As discussed in different chapters of the thesis, we followed the “industry-as-laboratory” approach by working closely with customers. Essentially, we followed to a large extent the principles of the “Action Research” model, where change is the success. By working closely with customers, we attempted to identify and solve “real problems” of their interest. We provided evidence that change has happened, in that customers or collaborators have been offered methods, tools, and design lessons that made impact on the quality of software products.

Based on several real-world endeavors, it is our position that software architectural design is a challenging job. Architects need reference material of proven architectural best practices so that they can build systems that are testable, meet performance requirements, are maintainable, and so forth. Our main goal was to come up with a large body of practically relevant architectural knowledge by systematically reverse architecting a pool of industrial systems and derive an array of recommendations so that quality can be designed in the first place. To efficiently and effectively analyze real-world systems, we followed an architecture-centric approach, meaning that we discovered the implemented software architecture and analyzed its testability, performance, and maintainability. Thus, we developed the Architecture Discovery and Analysis Method (ADAM). To this end, we enumerate the list of contributions of the thesis.

1. We proposed a method for analyzing the structural and behavioral constraints of the specified architectural style with respect to its implementation. The core idea of the method is that architectural styles offer vocabularies and constraints on the types of components and connectors, thus, we could derive rules from styles. We formalized styles using Colored Petri nets as the executable formal language. At runtime, we fed the collected runtime events to CP-nets to discover component-connector views, sequence diagrams, and analyze various constraints of styles. For details, the reader is referred to Chapter 2, Chapter 4, and Chapter 5.

2. We proposed a method for analyzing the architecture of unit test code from a maintainability point of view. We discussed architectural decisions that facilitate or impede unit testing. We offered a list of reusable questions to assess the maintainability of unit test code. We showed that these questions are simple, yet effective, to review unit test code. For details, the reader is referred to Chapter 3.
3. We proposed a method for discovering the software architecture from the implementation, and analyzing quality risks, in particular, risks related to performance, testability (with emphasis on unit testing), and maintainability. The core idea of the method is that architectural decisions of implemented systems are inspired and influenced by dependencies to external entities (e.g., COTS, frameworks, programming language libraries), which can be explored systematically for discovering software architectures and quality risks, hidden deep in the source code. Our method helps us in improving our understanding of relationships between software architectures and testing. For details, the reader is referred to Chapter 6 and Chapter 7.

4. We proposed a method for reverse architecting abstract runtime structures from the source code. We showed that our method was used to reason about testability at the architecture-level of safety critical medical device software. For details, the reader is referred to Chapter 7.

5. We proposed a method for analyzing organizational aspects of the system under analysis and their influence on implemented software architectures. Our method helps us in improving our understanding of relationships between software architectures and organizational aspects. For details, the reader is referred to Chapter 8.

6. We proposed generally applicable recommendations for quality by design instead of quality by tests. Based on architectural analysis of several systems, we derived an array of generally applicable recommendations so that quality can be built-in instead of being tested-in. Our recommendations characterize a) how to avoid architectural violations in the first place, b) how to facilitate testing by explicitly addressing testing related issues during the design of software architectures, and c) how to reduce performance risks by leveraging performance-oriented design patterns. For details, the reader is referred to Chapter 9.

7. We developed a suite of reverse engineering tools. These tools contribute to a) data extraction from implementation either statically from the source code or dynamically at runtime, and b) data abstraction and visualization of the collected fine-grained data. For details, the reader is referred to Chapter 10.

8. In Chapter 11, we revisited the research questions, which were formulated in the introduction, and discussed how we addressed them and also listed related open issues that are of interest for future research.

A good painter knows when to stop painting.
**Samenvatting**

In dit proefschrift hebben we, door de praktijk geïnspireerd, een aanpak voor software architectuur analyse uitgevoerd op industriële systemen. De aanpak volgt de filosofie van "industry-as-laboratory”. Door het toepassen van de beginselen van “Action Research” konden we in samenwerking met de industrie, echte problemen identificeren en de oplossingen valideren. In dit proefschrift hebben we aangetoond dat deze aanpak heeft geleid tot tools en design leermomenten die de kwaliteit van deze software producten heeft verbeterd.

Het ontwerpen van software architecturen is een uitdagende activiteit. Architecten hebben behoefte aan referentie-materiaal van bewezen best architecture practices, zodat ze hun eigen systemen kunnen toetsen dat ze voldoen aan kwaliteitsseisen, zoals onderhoudbaarheid, testbaarheid en runtime-performance. Ons belangrijkste doel was om te komen tot een aantal praktische en relevante architectuur feiten, door systematisch, uit bestaande industriële systemen, een reeks aanbevelingen af te leiden. Om bestaande real-world systemen efficiënt en effectief te analyseren, volgden we een benadering waarin de software architectuur centraal stond. Dat betekende dat we de geïmplementeerde software architectuur geanalyseerd hebben op zijn testbaarheid, performance en onderhoudbaarheid. Zo ontwikkelden we de Architectuur Discovery and Analysis Method (ADAM).

1. In dit proefschrift hebben we een analyse methode voorgesteld om structuur en gedrag vanuit de implementatie te herkennen en vast te leggen. Door het maken van een goede vocabulaire en beperkingen (constraints) op types van componenten en connectoren te leggen, konden we regels voor architectuur stijlen afleiden. We hebben deze stijlen geformaliseerd met behulp van gekleurde Petri netten. Tijdens de werking van het systeem hebben we gebeurtenissen (events) verzameld om, met behulp van deze Petrinetten, component-connector views, sequence diagrammen en verschillende architectuur stijlen te herleiden. Dit staat beschreven in hoofdstuk 2, hoofdstuk 4 en hoofdstuk 5.

2. Wij hebben een methode voorgesteld om onderhoudbaarheid te bepalen door unit test code te analyseren. We bespraken architectuur beslissingen die unit test code eenvoudiger te maken. Wij hebben een vragenlijst ontwikkeld om de onderhoudbaarheid van unit test code te bepalen. Verder toonden we aan dat deze vragen eenvoudig, edoch effectief, zijn om unit test code te bekijken. Voor meer details wordt de lezer verwezen naar hoofdstuk 3.
3. Wij hebben een methode voorgesteld om de implementatie te analyseren om de kwaliteit aspecten van de software architectuur te beschrijven. In het bijzonder de volgende kwaliteiten, met bijbehorende risico's: performance, testbaarheid (met nadruk op unit testen) en onderhoudbaarheid. De kerngedachte bestaat uit het feit dat architectuur beslissingen van systemen zijn geïnspireerd en zijn beïnvloed door afhankelijkheden met externe entiteiten (zoals Commercial Off The Shelf (COTS) componenten, frameworks en libraries). Door deze afhankelijkheden systematisch te onderzoeken kan de software architectuur en kunnen de kwaliteit risico's die diep zijn verborgen toch worden ontdekt. Onze methode helpt bij het verbeteren van ons begrip voor de relatie tussen software-architecturen en testen. Voor meer details wordt de lezer verwezen naar hoofdstuk 6 en hoofdstuk 7.

4. Wij hebben een methode voorgesteld om uit de source code de runtime structuur van software architectuur te halen. We toonden aan dat deze methode goed is om, op architectuur niveau, te redeneren over de veiligheid van kritische medische software systemen. Voor meer details wordt de lezer verwezen naar hoofdstuk 7.

5. Wij hebben een methode voorgesteld om (bedrijfs-) organisatie aspecten te analyseren die invloed hebben op de software architectuur. Onze methode ondersteunt het verbeteren van de relaties tussen software-architecturen en organisatorische aspecten. Voor meer details wordt de lezer verwezen naar hoofdstuk 8.

6. We hebben algemeen toepasbare aanbevelingen gegeven om de kwaliteit tijdens design te verbeteren in plaats van kwaliteit te meten tijdens test. Gebaseerd op architectuur analyse van de diverse praktijk systemen, hebben we een reeks van algemeen geldende aanbevelingen ontdekt die de kwaliteit reeds tijdens ontwerp realiseert, in plaats van achteraan in het proces tijdens het testen. Onze aanbevelingen zijn te karakteriseren als a) hoe architectuur overtredingen te voorkomen in de eerste plaats, b) hoe het testen te vergemakkelijken door het testen expliciet te ontwerpen en c) hoe de performance risico's te verminderen door gebruik te maken performance-georiënteerde design patterns. Voor meer details wordt de lezer verwezen naar hoofdstuk 9.

7. We hebben een suite van reverse engineering tools ontwikkeld. Deze tools dragen bij aan a) extractie van gegevens tijdens de werking van het systeem, hetzij statisch uit de source code of dynamisch tijdens runtime, en b) abstractie en visualisatie van de verzamelde gegevens. Voor meer details wordt de lezer verwezen naar hoofdstuk 10.

8. In Hoofdstuk 11 hebben we opnieuw naar onze onderzoeksvragen gekeken vanuit het perspectief van gedane onderzoek. De nog openstaande kwesties van belang voor toekomstig onderzoek zijn hierin ook beschreven.
Een goede schilder weet wanneer hij moet stoppen met schilderen.
Appendix A
Catalog of Analyzed Systems

As mentioned earlier, this thesis evolved from a series of architectural analysis projects of industrial-strength systems. Here, we list all the systems and briefly characterize each of them, so that the reader can get an overview of the nature of architectural analyses that were done by us, and the kinds of architectural issues typically found in these real-world industrial projects.

First, by analyzing several systems we learned the types of architectural problems organizations face for which they need help. By addressing the right set of architectural problems, we improved our architecture reconstruction approach based on feedback from our customers. Second, we also helped our customers by sharing best architectural practices followed in other projects. Third, we were able to build a knowledge base and collect and compare different ways systems implement the same architectural concept. This knowledge base was used by analysts for analyzing other systems in an efficient way, and also for sharing architectural best practices between projects. For example, one of the projects had an elegant way to implement an OS abstraction layer (OSAL). We shared this valuable design knowledge with other projects which did not have a clear separation of OS concerns from other concerns. Fourth, we were able to collect architectural principles and build processes that make it difficult to introduce architectural violations in the first place. All these endeavors enabled us to better understand relationships between software architectures and quality properties such as testability, performance, and maintainability.

We were actively involved in all projects listed below. These projects were performed by us from the year 2003 onwards. All tables support the fact that the proposed Architecture Discovery and Analysis Method (ADAM) has been successfully applied to and evolved from many industrial systems in various domains, including aerospace, automobiles, finance, office appliances, and medical information systems.
Table A-1: Analysis of MarketMaker's software product line

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>MarketMaker’s Software Product Line for web-based stock market systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goals were to:</td>
</tr>
<tr>
<td></td>
<td>1. Summarize the evolution of different versions of the product line.</td>
</tr>
<tr>
<td></td>
<td>2. Organize evolution monitoring workshops with the development team so that they understand what parts are changed and why they are changed.</td>
</tr>
<tr>
<td></td>
<td>3. Perform runtime analyses to locate issues with the response time of variants of the product line.</td>
</tr>
<tr>
<td>Key Results</td>
<td>This product line was implemented in Java. The first two goals were achieved using static analysis, in particular using object-oriented metrics. The third goal was achieved using the dynamic analysis, by instrumentation of the byte-code of Java classes and collecting and analyzing timing data in the production environment. In this project, we developed an environment for analyzing “delta” between versions, and techniques for dynamic analysis of production software.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2003-2004</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to [115], [89], and [105] for details.</td>
</tr>
</tbody>
</table>
Table A- 2: Migration of the engine control systems to a product line

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Hitachi’s Engine Control System (ECS).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>Hitachi’s ECS business unit had several versions of stand-alone ECS systems. The goal was to assess the potential of merging several existing systems into a software product line.</td>
</tr>
<tr>
<td>Key Results</td>
<td>All versions were implemented in C with some assembly code. We extracted module-views from the source code of each version, and analyzed source code clones among different versions at an architectural-level. We also performed an analysis of organizational aspects such as structure of teams, and owners of existing components. Based on these analyses, we defined a migration strategy to a product line. In this project, we developed a method for analyzing the existing “clone-and-own” versions and migrating to a product line.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2005-2006</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to Chapter 8 or [107], [315], [316], and [317] for details.</td>
</tr>
</tbody>
</table>
Table A- 3: Performance analysis of Ricoh’s reusable component

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Ricoh’s reusable user interface component (UIC).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The UIC component supports four-line LCD, VGA and WVGA output panels with function keys and touch screens as input devices. UIC was used in many products at Ricoh. The products’ teams reported that the UIC has performance problems. The UIC team, however, refused to take responsibility and remove the problems. Our goal was analyze the UIC’s performance properties and help resolving this issue.</td>
</tr>
<tr>
<td>Key Results</td>
<td>UIC was implemented in C++. We performed instrumentation of UIC’s source code and ran it on the emulation product as well as two product instances that use the UIC. Based on the collected response time data, we were able to locate performance problems in the UIC component. In this project, we gained hands-on experiences with the performance analysis of reusable software components.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2005-2006</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to [163] for details.</td>
</tr>
</tbody>
</table>
Table A-4: Migration of the digital cameras to a product line

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Digital Camera.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goal of the analysis was to compare the implemented architectures of different versions of the software used in cameras, and propose a strategy for introducing a software product line using existing artifacts.</td>
</tr>
<tr>
<td>Key Results</td>
<td>All versions were written in C++. We extracted module views of each version. We also measured code cloning among different versions and the usage of conditional preprocessor constructs. Based on this analysis, we proposed an incremental migration strategy to a product line. In this project, we gained experiences of analyzing “clone-and-own” variants and architectural challenges due to variants.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2005-2006</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A- 5: Architectural analysis of Ricoh’s MFPs

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Ricoh’s Multi-Function Peripherals (MFPs).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>Ricoh has developed a MFP product line using the flexible pipe-and-filter architectural style, which enables its customers and vendors to customize the machine to a large extent. One can even introduce new software components into the architecture. The goal of the analysis was to analyze whether or not (previously unknown) components can work together safely at runtime by following the constraints of the pipe-and-filter style.</td>
</tr>
<tr>
<td>Key Results</td>
<td>Most of the MFPs were implemented in Java, with some legacy and booting code implemented in C. We developed an elaborated approach for monitoring the running MFP and checked whether or not each components followed the architectural constraints of the pipe-and-filter style. An appealing aspect of this approach is that behavioral violations can be detected at runtime itself, which is important for these types of systems, because we have to remove software components that do not follow constraints of the architectural style at runtime. In this project, we developed a method for architectural analysis of systems based on the pipe-and-filter architectural style.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2006-2007</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to Chapter 4 or [94] and [95] for details.</td>
</tr>
</tbody>
</table>
Table A- 6: Architectural analysis of Testo's software product line

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Testo’s climate and temperature measurement product line.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goal of the analysis was to review the quality of the implementation of the product line.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The source code was written in C. We extracted module dependencies and measured structural complexity using metrics. We organized workshops with the developers of the product line and presented our findings. In this project, we gained experiences with variability and testing issues of a software product line at the code level.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2005-2006</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to [96] for details.</td>
</tr>
</tbody>
</table>
Table A- 7: Architectural analysis of variability issues of farming trucks

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Software for farming trucks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>One of the leading manufacturers of trucks wanted us to analyze the product line potential of existing versions of their software, used in farm trucks.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The source code was written in C++. We extracted module dependencies and measured the usage of conditional preprocessor constructs. We organized workshops with the developers and presented a migration strategy to a software product line. In this project, we gained experiences with variability issues at the code level.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2006</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A-8: Architectural analysis of an electrical appliances system

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Electrical measurement embedded systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goals were to:</td>
</tr>
<tr>
<td></td>
<td>1. Document the implemented module view and report architectural violations with respect to the specified module view.</td>
</tr>
<tr>
<td></td>
<td>2. Locate routines that affected the system’s response time for scenarios reported by the end-users of the product.</td>
</tr>
<tr>
<td>Key Results</td>
<td>This system is fully implemented in C. The first goal was achieved by extracting module dependencies from the source code. In order to achieve the second goal, we used dynamic analysis by monitoring the running system that was given to us. We were able to run the embedded system and collected traces were analyzed by visualizing sequence diagrams using our tool-suite. In this project, we gained experiences with dynamic analysis of a resource constrained embedded system using aspect-oriented runtime data collection techniques.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2007</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to [303] and [92] for details.</td>
</tr>
</tbody>
</table>
Table A- 9: Architectural analysis of Cranefoot

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Cranefoot – an open source Pedigree visualization framework.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>Our customer BioFortis wanted to evaluate the technical feasibility of integrating Cranefoot into their medical information system LabMatrix. To this end, they asked us to review the Cranefoot source code and analyze whether or not it implements 20 necessary requirements already. If these requirements were not implemented, how difficult it would be to introduce them into Cranefoot’s source code.</td>
</tr>
<tr>
<td>Key Results</td>
<td>This system is fully implemented in C++. We extracted module dependencies using the source code of Cranefoot. We documented the resulted modular structure and performed change-impact analysis with respect to the given 20 requirements. We found that one key requirement was not possible to implement without drastically changing the software architecture. All other requirements were possible to implement without changing the existing structure. In this project, we gained experiences of using architectures for impact analysis, and also licensing issues we need to be aware of when integrating open source into “closed source”.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2008</td>
</tr>
<tr>
<td>References</td>
<td>Our architecture documentation and analyses results are released as part of the Cranefoot’s distribution [91].</td>
</tr>
</tbody>
</table>
Table A-10: Architectural analysis of Madeline

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Madeline - an open source Pedigree visualization framework.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The same as the above Cranefoot scenario.</td>
</tr>
<tr>
<td>Key Results</td>
<td>This system is fully implemented in C++. We repeated the same process followed for the evaluation of Cranefoot. We found that many of the 20 requirements were already implemented in Madeline; the developer of Madeline has offered to implement the missing requirements. In this project, we gained further experiences of using architectures for impact analysis, and related licensing issues.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2008</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to [93] for details.</td>
</tr>
</tbody>
</table>
Table A-11: Architectural analysis of the NASA’s CFS

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>The NASA’s Core Flight Software Product Line (CFS).</th>
</tr>
</thead>
</table>
| Analysis Goals     | The CFS team has developed a flight software product line based on flexible the publisher-subscriber architectural style, which enables its customers to plug-and-play software components even at runtime. The goals were:  
  1. Analyze the implemented architecture of the CFS product line and locate deviations to the specified architectural rules.  
  2. Analyze the architecture of the CFS’ unit testing framework, and suggest improvement recommendations. |
| Key Results         | The CFS was fully implemented in C. To achieve the first goal, we analyzed the CFS statically and extracted its key architectural concepts such as the publisher-subscriber style, its OS abstraction layer (OSAL). In addition, we detected some violations of the OSAL and other architectural rules. Our analysis of the unit testing tests highlighted those architectural decisions that made unit testing complex to understand and evolve. In this project, we developed a method for analyzing architectural rules of software product lines, and also methods for analyzing the unit test code and an understanding of architectural decisions that impede or support unit testing. |
| Year of Analysis    | 2008 – Present (i.e., active during 2011) |
| References          | Please refer to Chapter 2 (or [97]) and Chapter 3 (or [101] and [102]) for details. |
Table A- 12: Architectural analysis of the NASA’s GMSEC

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>The NASA’s GMSEC Product Line.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The GMSEC team has developed a ground software product line based on flexible the publisher-subscriber architectural style, which enables its customers and vendors to plug-and-play software components. The goal was to analyze whether or not the implementation follows both from the structure and behavioral constraints of the style.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The core of the GMSEC is implemented in C++. However, other language bindings such as for C, Java, and Perl are also provided. We analyzed the GMSEC using static and dynamic analyses because the architecture is based on the publisher-subscriber style, which is difficult to fully analyze statically. We discovered its middleware abstraction layer, and the runtime architectural views such as component-connector views and sequence diagrams. Furthermore, we identified some high-priority bugs due to the violations of behavioral constraints of the same API for different middleware vendors’ used for implementing the software bus. In this project, we developed a method for architectural analysis of systems based on the publisher-subscriber style.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009 – Present (i.e., active during 2011)</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to Chapter 5 or [103] for details.</td>
</tr>
</tbody>
</table>
Table A- 13: Risk analysis of the NASA’s space network

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>The NASA’s Space Network (SN).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goals of the analysis were to document and deliver module dependency diagrams and identify risky code elements, which are difficult to test, for example. This was done partly as a help provided to the new contractors who are overtaking the parts of the SN from other contractors.</td>
</tr>
<tr>
<td>Key Results</td>
<td>It is worth noting that the SN is not a single system. It is made of systems of several systems that were compiled, deployed, and executed on different machines across different NASA centers. Thus, it is made of several languages such as Ada, Fortran, C, C++, Java, and SQL. There were around 10 Million Lines of code to analyze. This was a very huge project for us. In this project, we gained experiences with reverse engineering of large systems.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009 – 2010</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A-14: Architectural analysis of the NASA’s SNAS system

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>The NASA’s Space Network Access System (SNAS).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The SNAS is the front-end of the Space Network system used by the NASA. The goals of the analysis were to document the implemented architecture and to identify risks due to performance and low-testability.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The SNAS is implemented in Java and in SQL. We extracted SNAS’s module dependencies and component-connector views. We detected performance, testability, and error-handling issues which are currently being addressed by the development team. In this project, we developed a method for analyzing the implemented system with respect to concerns, and pinpoint testability and performance risks by only reviewing a minimal set of source code files.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009 – 2010</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to Chapter 6 or [99] for details.</td>
</tr>
</tbody>
</table>
**Table A- 15: Architectural analysis of CCIS**

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>The TSS Sweden - Cold Chain Information System (CCIS).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>Our customer TSS, Sweden, wanted us to review the implementation of two versions of CCIS with respect to the design quality of the implementation. The goal was perform an independent review and document the finding so that the results can be shown to the external quality control auditors.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The CCIS has two versions, namely the online and offline versions. As the name suggests, the online version is a web-based system implemented using Java, JavaScript, Stripes, and Hibernate. The Offline version is implemented in Java. We extracted module dependencies and analyzed how the implementation handles several concerns such as the GUI, Persistence, interactions with hardware sensors and thermometers.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009 – 2010</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A-16: Introducing reverse engineering to software architects

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>A Web-based Medical Information Management System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The goal of the analysis was to introduce software reverse engineering methods and tools to the architects of this system. To this end, the customer offered us to use one of their web-based systems to explain how reverse engineering can help them in analyses of design quality in general.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The system was fully implemented in Java, J2EE, Spring, Hibernate, and Ajax. We extracted module dependencies from the implementation and identified issues such as the bypass of logging wrappers, mixture of database concerns with business concepts, and inappropriate error or exception handling. In this project, we gained experiences with teaching of reverse engineering methods and tools to software architects.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009 – 2010</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A- 17: Architectural analysis of Archivex

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>Archivex – Laboratory Inventory Management System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The main and only developer left this start-up company in Baltimore. Our goals were a) review the source code quality in general, and b) report to the CEO of the company on the current status of the system so that contract obligations with the developer can be settled appropriately.</td>
</tr>
<tr>
<td>Key Results</td>
<td>The key results were that the developer had spent a lot of effort in infrastructure code, thus, very limited user-visible features were developed. It turned out that several new technologies were used in this project, for example, Google’s GWT framework, Apache’s Derby Database, and Jetty Web server. We reported that these new technologies pose risk because it may not be easy to find new people to take over this project. Based on our recommendations, the management changed the recent versions of this project to use “traditional” web-technologies, which meet the needs of this project. In this project, we gained experiences with several new technologies and their implications to reverse engineering.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2009</td>
</tr>
<tr>
<td>References</td>
<td>Not published due to confidentiality reasons.</td>
</tr>
</tbody>
</table>
Table A-18: Introducing reverse engineering at the US FDA

<table>
<thead>
<tr>
<th>Name of the system</th>
<th>CARA at the US Food and Drug Administration (FDA).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Goals</td>
<td>The primary goal of the FDA’s division of Electrical and Software Engineering is to help manufacturers get the software right the first time so that the FDA has fewer adverse events to investigate. To the extent that they sometimes fail, the FDA always has a need for forensic engineering, but that's not the primary focus of the FDA. To this end, we are in the process of training software engineers our reverse engineering methods and tools, so that they can extract architectural knowledge from the implementation of the system they are investigating.</td>
</tr>
<tr>
<td>Key Results</td>
<td>We are working on architectural analyses of the implementations of medical device software. We extracted module dependencies as well runtime inter-task dependencies. We detected testability problems due to lack of separation of computation from communication concerns.</td>
</tr>
<tr>
<td>Year of Analysis</td>
<td>2010 – Present (i.e., active during 2011)</td>
</tr>
<tr>
<td>References</td>
<td>Please refer to Chapter 7 or [100] and [98] for details.</td>
</tr>
</tbody>
</table>
Appendix B
Relation Partition Algebra (RPA)

RPA offers set and relational algebraic operators for formalizing and executing mathematical formulas. In this thesis, we use RPA for discovering module and runtime views from source code. Here, we present an overview of RPA concepts. The reader is referred to [169] for a detailed presentation of RPA.

Set: A set is a collection of unique elements where the elements can be of any type. For example, a set can be a collection of functions, procedures or classes defined in a software system.

Union of Sets: Let A and B be two sets. The union of A and B is the set of elements that are present in either A or B. For example, if A = {1, 3, 5} and B = {2, 4, 6} then the union of A and B is {1, 3, 5, 2, 4, 6}.

Intersection of Sets: Let A and B be two sets. The intersection of A and B is the set of elements that are present in both A and B. For example, if A = {1, 2, 3, 5} and B = {2, 4, 6} then the intersection of A and B is {2}.

Binary relation: A binary relation is a set of ordered pairs of elements (a.k.a. 2-tuples). For our purposes, the order means that the first element of the pair is dependent on or part of the second element (e.g., if the elements are functions and function f calls function g then this fact would be encoded as <f, g>). Hereafter, we simply refer a binary relation as a relation.

Domain: The left column of a relation is called the domain set. For example, if Call = {<main, f>, <f, g>, <g, h>} then the domain of the Call relation is the set {main, f, g}.

Range: The right column of a relation is called the range set. For example, if Call = {<main, f>, <g, h>} then the range of the Call relation is the set {f, h}.

Carrier: The carrier of a relation is the union of its domain and range sets. For example, if Call = {<main, f>, <f, g>, <g, h>} then the carrier of the Call relation is the set {main, f, g, h}.

Union of Relations: The union of two relations is formed by taking all tuples of two relations. For example, if Call = {<main, f>, <f, g>} and Call2 = {<f, g>, <g, h>} then the union of these two relations is the relation {<main, f>, <f, g>, <g, h>}.

Intersection of Relations: The intersection of two relations is formed by taking all tuples that are present in both relations. For example, if Call = {<main, f>, <f, g>} and Call2 = {<f, g>, <g, h>} then the intersection of these two relations is the relation {<f, g>}.
Converse: The converse of a relation is formed by swapping its domain and range. For example, if Call = \{<main, f>, <f, g>, <g, h>\} then the converse of the Call relation is the relation \{<f, main>, <g, f>, <h, g>\}.

Transitive Closure: The transitive closure of a relation is formed by performing a reachability analysis of the domain of a relation. This is done by taking each value in the domain and identifying all values in the range that can be reached directly and indirectly through other tuples. New tuples are added for each discovered reachability pair. For example, if the Call relation of a program is Call = \{<main, run>, <run, execute>, <execute, print>\} then the transitive closure of the call relation is the union of Call and the relation \{<main, print>, <main, execute>, <run, print>\} because the print function is reachable from both the main and the run functions, and execute is reachable from main.

Composition: The composition of two relations produces a new relation by performing a join of the first relation’s range with the domain of the second relation. For example, if Call = \{<main, f>, <f, g>\} and Call2 = \{<f, g>, <g, h>\} then the composition of these two relations is the relation \{<main, g>, <f, h>\}.

Lifting: Given a relation \(R\) and a part-of relation \(P\) we can construct a new relation \(Q\) by lifting \(R\) using \(P\). For example, if \(R = \{<\text{main}, f>, <f, g>\}\) and \(P = \{<\text{main}, \text{main.c}>, <\text{f, f.c}>, <\text{g, g.c}>\}\) then \(Q = \{<\text{main.c, f.c}>, <\text{f.c, g.c}>\}\). In this example, the lift operator helped us in extracting the dependencies between files using the call dependencies between functions \(R\) and the hierarchical relation between the functions and the files they were defined in \(Q\).

Domain Lifting: Given a relation \(R\) and a part-of relation \(P\) we can construct a new relation \(Q\) by lifting the domain of \(R\) using \(P\). For example, if \(R = \{<\text{main}, f>, <f, g>\}\) and \(P = \{<\text{main, main.c}>, <\text{f, f.c}>, <\text{g, g.c}>\}\) then \(Q = \{<\text{main.c, f}, <\text{f, c, g}>\}\). In this example, the domain lift operator helped us in extracting the dependencies from files to functions using the call dependencies between functions \(R\) and the hierarchical relation between the functions and the files they were defined in \(Q\).

Range Lifting: Given a relation \(R\) and a part-of relation \(P\) we can construct a new relation \(Q\) by lifting the range of \(R\) using \(P\). For example, if \(R = \{<\text{main}, f>, <f, g>\}\) and \(P = \{<\text{main, main.c}>, <\text{f, f.c}>, <\text{g, g.c}>\}\) then \(Q = \{<\text{main, f.c}, <\text{f, c, g}>\}\). In this example, the range lift operator helped us in extracting the dependencies from functions to files using the call dependencies between functions \(R\) and the hierarchical relation between the functions and the files they were defined in \(Q\).

Domain Restriction: The domain restriction operator restricts the domain of a relation to a given set. For example, if Call = \{<main, h>, <main, g>, <f, h>, <g, h>\} and \(D = \{\text{main, f}\}\) then the domain restriction of Call on \(D\) is the relation \{<main, h>, <main, g>, <f, h>\}.

Range Restriction: The range restriction operator restricts the range of a relation to a given set. For example, if Call = \{<main, h>, <main, g>, <f, i>, <g, h>\} and \(R =
\{h, i\} then the range restriction of Call on R is the relation \{<main, h>, <f, i>, <g, h>\}.

**Carrier Restriction**: The carrier restriction operator restricts the carrier of a relation to a given set. For example, if Call = \{<main, h>, <main, g>, <f, i>, <g, h>\} and C = \{main, i, g\} then the carrier restriction of Call on C is the relation \{<main, g>\}.

**Top or Root Elements**: The top or root elements of a relation are the set of elements that are in the domain but not in the range. For example, if Call = \{<main, h>, <h, g>, <run, h>\} then the top elements of Call is the set \{main, run\}.

**Bottom or Leaf Elements**: The bottom or leaf elements of a relation are the set of elements that are in the range but not in the domain. For example, if Call = \{<main, h>, <h, g>, <run, h>\} then the bottom elements of Call is the set \{g\}.

**Left Image**: The left image of an element returns the subset of the domain of a relation where the range value is matching the given element. For example, if Call = \{<main, h>, <h, g>, <run, h>\} then the left image of the element h on Call is the set \{main, run\}.

**Right Image**: The right image of an element returns the subset of the range of a relation where the domain value is matching the given element. For example, if Call = \{<main, h>, <h, g>, <run, h>, <run, i>\} then the right image of the element run on Call is the set \{h, i\}.

**Multi-set**: A multi-set is a weighted set, meaning that each element of the set has a positive integer weight. For example, A = \{<a, 1>, <b, 2>, <c, 3>\}. Naturally, a multi-set is also a relation with its range as a set of positive integers only.

**Multi-relation**: A multi-relation is a weighted relation, meaning that each tuple of the relation has a positive integer weight. For example, Call = \{<main, run, 3>, <main, stop, 1>\} denotes that the main function calls the run function 3 times and the stop function one time.

Figure B-1 summarizes the list of RPA operators and notations that are used in the thesis.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Union of two relations or sets</td>
</tr>
<tr>
<td>∩</td>
<td>Intersection of two relations or sets</td>
</tr>
<tr>
<td>o</td>
<td>Composition of relations A and B</td>
</tr>
<tr>
<td>A⁻¹</td>
<td>Converse of the relation A</td>
</tr>
<tr>
<td>A*</td>
<td>Transitive closure of the relation A</td>
</tr>
<tr>
<td>A(dom)</td>
<td>Restrict the domain of the relation A to the set S</td>
</tr>
<tr>
<td>A(ran)</td>
<td>Restrict the range of the relation A to the set S</td>
</tr>
<tr>
<td>A(car)</td>
<td>Restrict the carrier of the relation A to the set S</td>
</tr>
<tr>
<td>T(A)</td>
<td>Set of top or root elements of the relation A</td>
</tr>
<tr>
<td>↑(A)</td>
<td>Set of bottom or leaf elements of the relation A</td>
</tr>
<tr>
<td>A.x</td>
<td>The left image of the element x</td>
</tr>
<tr>
<td>x.A</td>
<td>The right image of the element x</td>
</tr>
<tr>
<td>A↑P</td>
<td>Lift the relation A using the relation P</td>
</tr>
<tr>
<td>A(dom) P</td>
<td>Lift the domain of the relation A using the relation P</td>
</tr>
<tr>
<td>A(ran) P</td>
<td>Lift the range of the relation A using the relation P</td>
</tr>
<tr>
<td>[A]</td>
<td>Convert the set/relation A to a multi-set/relation</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Figure B-1: The RPA operators used in the thesis.
Appendix C
Colored Petri Net (CP-net)

In this thesis, we use Colored Petri nets (CP-nets) for analyzing runtime events of software systems. Here, we present an overview of CP-nets using straightforward and intuitive examples. For a detailed discussion of CP-nets, we refer the reader to [139] and [1].

**Petri net**: A Petri net consists of places, transitions, and directed arcs [229], [209], and [228]. Arcs run between places and transitions, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. In Figure C-1, P1, P2, P3, and P4 are places, and T1 and T2 are transitions. The dot within a place is called a token. P1, P3, and P4 have one token. Places may contain any non-negative number of tokens. A distribution of tokens over the places of a net is called a “marking”.

![Sample Petri net](image)

Figure C-1: Sample Petri net.

A transition of a Petri net may fire whenever there are tokens at all input places; when it fires, it consumes these tokens, and places tokens at all output places (see Figure C-2). In Figure C-1, T1 can fire, but T2 cannot fire because P2 has no tokens. A firing is atomic and therefore a single non-interruptible step. In general, the execution of Petri nets is nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire.
**Colored Petri Net:** A colored Petri net (CP-net) extends the basic version of a Petri net by introducing the notion of colors or data types for places. In basic Petri nets, tokens do not have any structure, that is, they do not hold data. In a CP-net, tokens are data holders according to data type definitions. Every place in a CP-net is assigned a data type, and every token inside a place is an instance of the corresponding type. Places with the same data type are assigned the same color. Every transition of a CP-net contains a precondition (e.g., a conditional expression) based on input places. If the precondition is true then the transition will fire, and tokens from the input places that satisfy the precondition will be consumed.

![Figure C-2: Petri net after firing the transition T1.](image)

**Figure C-2:** Petri net after firing the transition T1.

**Figure C-3** shows a CP-net that computes the total salary of all employees. Each token in the `Account` place represents an employee. The `Total_Salary` place has one token with an integer value zero. Whenever the `Add salary` transition fires it consumes one token from its input places, namely the `Account` and `Total_Salary`. The transition also adds the salary of the consumed token to the total salary. The token stored in the `Total_Salary` place is updated with the latest total. After three steps, the `Total_Salary` place has the sum of salaries of all employees, see Figure C-4.

![Figure C-3: CP-net for computing the total salary.](image)

**Figure C-3:** CP-net for computing the total salary.

```xml
<emp_id="E1", yearly_salary=200000, currency="dollar">
<emp_id="E2", yearly_salary=100000, currency="dollar">
<emp_id="E3", yearly_salary=250000, currency="dollar">
Add salary
Total_Salary

Total_Salary = Total_Salary + Account.yearly_salary
```

```xml
Account
<emp_id="E2", yearly_salary=100000, currency="dollar">
Total_Salary
<emp_id="E3", yearly_salary=250000, currency="dollar">
Add salary 0
Total_Salary = Total_Salary + Account.yearly_salary
```
Suppose we want to find “rich” employees who earn at least 250000 dollars annually. We formalize this constraint as the precondition of the transition, see Figure C- 5. If an input token satisfies this precondition, the action statement will be executed. As the result, the token will be moved to the output place Rich_Account.

The Filter Rich Acct transition will fire only once because the precondition is satisfied by only one employee. At the end of the transition, we are left with two employees in the Account place and one employee in the Rich_Account place, see Figure C- 6.
Hierarchical Colored Petri net: The CP-net models of real world systems often contain a vast amount of places and transitions, making it very hard to analyze, visualize, maintain, and reuse parts of them in other contexts. Moreover, putting all places and transitions at the same level does not reflect the structure of the system being modeled. Hierarchical CP-nets help us to organize CP-nets like a tree. In a hierarchical CP-net, in addition to places and transitions, we can also embed CP-nets as elements. These CP-nets are called sub-CP-nets.

Figure C-6: CP-net after filtering rich accounts.
Bibliography


Booch, G. http://www.handbookofsoftwarearchitecture.com


[47] CFR - Code of Federal Regulations Title 21, United States FDA.


[145] Juice Framework. [http://google-juice.net/](http://google-juice.net/)


[197] mCRL2. [http://www.mcrl2.org](http://www.mcrl2.org)


[266] SPIN Model Checking Tool. [http://www.spinroot.com](http://www.spinroot.com)


[287] US Food and Drug Administration. [http://www.fda.gov/AboutFDA/default.htm](http://www.fda.gov/AboutFDA/default.htm)


Curriculum Vitae

In 1993, Dharmalingam Ganesan received his secondary school certificate at the American College Higher Secondary School in Madurai, India. That same year Dharmar started to study Bachelor of Science (B.Sc.) in Mathematics at American College in Madurai. Dharmar obtained his B.Sc. degree with distinction in 1996. That same year Dharmar started to study Master of Science (M.Sc.) in Mathematics and Computer Science at College of Engineering, Anna University, Chennai, India. Dharmar obtained his M.Sc. degree with distinction in 1998. That same year Dharmar started to study Master of Technology (M.Tech.) in Computer Science and Technology at Jawaharlal Nehru University (JNU), New Delhi, India. Dharmar obtained his M.Tech. degree with distinction in 2000.

In 2000, Dharmar joined IBM Global Services in Bangalore, India as a Software Engineer. Dharmar moved to Philips Research in Bangalore and worked there as a Research Engineer from July 2000 to June 2001. Dharmar got a job offer to work as a Software Engineer for emagine Gmbh, a subsidiary of Deutsche Bank in Frankfurt, Germany. Dharmar worked at emagine Gmbh from July 2001 to December 2002. In January 2003, Dharmar joined the Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany, as a Research Scientist. In January 2008, Dharmar moved to the Fraunhofer Center for Experimental Software Engineering (CESE), Maryland, USA, and employed as a Research Scientist.