TOWARDS CONTROLLING SOFTWARE ARCHITECTURE EROSION THROUGH RUNTIME CONFORMANCE MONITORING

Lakshitha de Silva

A Thesis Submitted for the Degree of PhD at the University of St Andrews

2014

Full metadata for this item is available in Research@StAndrews:FullText at:
http://research-repository.st-andrews.ac.uk/

Please use this identifier to cite or link to this item:
http://hdl.handle.net/10023/5220

This item is protected by original copyright
Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring

Lakshitha de Silva

This thesis is submitted in partial fulfilment for the degree of Doctor of Philosophy at the University of St Andrews

May 2014
Abstract

The software architecture of a system is often used to guide and constrain its implementation. While the code structure of an initial implementation is likely to conform to its intended architecture, its dynamic properties cannot always be fully checked until deployment. Routine maintenance and changing requirements can also lead to a deployed system deviating from this architecture over time. Dynamic architecture conformance checking plays an important part in ensuring that software architectures and corresponding implementations stay consistent with one another throughout the software lifecycle. However, runtime conformance checking strategies often force changes to the software, demand tight coupling between the monitoring framework and application, impact performance, require manual intervention, and lack flexibility and extensibility, affecting their viability in practice. This thesis presents a dynamic conformance checking framework called PANDArch framework, which aims to address these issues. PANDArch is designed to be automated, pluggable, non-intrusive, performance-centric, extensible and tolerant of incomplete specifications. The thesis describes the concept and design principles behind PANDArch, and its current implementation, which uses an architecture description language to specify architectures and Java as the target language. The framework is evaluated using three open source software products of different types. The results suggest that dynamic architectural conformance checking with the proposed features may be a viable option in practice.
Acknowledgements

This PhD has been a quest. It was a journey I enjoyed for most part. But at times it has also been an emotional nightmare filled with frustration, fear, desperation and heart break. Then again, the people who were around me made this journey all worth while. For that, they have my heart.

First and foremost, my deep felt gratitude goes to my supervisor Dharini Balasubramaniam. I wouldn’t have done this without your guidance, patience, support and the immense commitment to seeing me through. One couldn’t ask for a better supervisor, and it is a privilege working with you. I am also grateful to my secondary supervisors Simon Dobson and Steve Linton for sharing their wisdom.

I thank the wonderful people at #stacs. This is an amazing school to study, not only because of the unimaginable forms of cake served during tea or the regular bbq parties, but also because of the genuine support you always provide students.

My PhD was funded through a studentship from the Scottish Informatics and Computer Science Alliance (SICSA), whom I wish to thank, not only for their financial generosity, but also for their vehement and varied support towards uplifting computer science research in Scotland.

I wish to thank the folk at Structure101 for affording me a free license to their great product. It was immensely helpful in my work.

To my pub-crawling, bop-loving, curry-crazy gang and numerous office mates who made me smile when I needed to; thank you for keeping me sane.

I am enormously thankful to my wife Disha who has given me abundance of love and support despite my obnoxious moods, and to my family for continuously believing in me. You all are my pillars of strength.

My eternal gratitude goes to my parents, who were rather surprised when I undertook this endeavour. I never would have come this far without your encouragement, understanding, patience and love. Thank you for giving me the strength to believe in my dreams and the freedom to pursue them. This one is for you.
Declaration

Candidate’s Declarations

I, Lakshitha de Silva, hereby certify that this thesis, which is approximately 48,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student and as a candidate for the degree of Doctor of Philosophy in September, 2009; the higher study for which this is a record was carried out in the University of St Andrews between 2009 and 2013.

Date:

Signature of candidate:

Supervisor’s Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date:

Signature of supervisor:
Permission for Electronic Publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis:

Access to printed copy and electronic publication of thesis through the University of St Andrews.

Date:

Signature of candidate:

Signature of supervisor:
To Ammi and Thaththi.

To Aunty Indrani and Uncle Leicester.
1 Introduction
   1.1 Overview ......................................................... 1
   1.2 Terminology .................................................... 5
   1.3 Hypotheses ...................................................... 6
   1.4 Motivation ....................................................... 7
       1.4.1 The need for Pluggability ................................. 7
       1.4.2 The need for Non-intrusiveness ........................... 7
       1.4.3 The need for Customisability ............................... 8
   1.5 Main Contributions ............................................. 9
   1.6 Thesis Organisation ........................................... 11
   1.7 Publications .................................................. 12

2 Background ......................................................... 13
   2.1 Introduction ................................................... 13
   2.2 Classification .................................................. 15
   2.3 Process-oriented Architecture Compliance .................... 17
       2.3.1 Architecture Design Documentation ....................... 18
       2.3.2 Architecture Analysis ........................................ 19
       2.3.3 Architecture Conformance Monitoring ..................... 19
       2.3.4 Dependency Analysis ......................................... 20
       2.3.5 Discussion .................................................. 21
   2.4 Architecture Evolution Management ............................ 23
   2.5 Architecture Design Enforcement ............................. 25
       2.5.1 Code Generation .............................................. 25
       2.5.2 Architecture Patterns ....................................... 26
       2.5.3 Architecture Frameworks ..................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.4</td>
<td>Discussion</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Architecture to Implementation Linkage</td>
<td>29</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Discussion</td>
<td>30</td>
</tr>
<tr>
<td>2.7</td>
<td>Self-adaptation</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Architecture Restoration</td>
<td>34</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Architecture Recovery</td>
<td>34</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Architecture Discovery</td>
<td>36</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Architecture Reconciliation</td>
<td>36</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Discussion</td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td>Conclusions</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Strategy</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture Conformance Checking</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Static and Dynamic Conformance Checking</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Top-down vs. Bottom-up Checking</td>
<td>43</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Automated vs Manual Checking</td>
<td>43</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Modelling Software Architectures</td>
<td>44</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Mapping Architecture to Implementation</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Conceptual Foundation</td>
<td>46</td>
</tr>
<tr>
<td>3.3.1</td>
<td>First Principles</td>
<td>46</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The Architecture Environment</td>
<td>48</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Architecture Conformance Rules</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Proposed Approach</td>
<td>53</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Conformance Monitoring Process</td>
<td>53</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Specifying Architectures for Conformance Checking</td>
<td>54</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Linking Architecture to Implementation</td>
<td>55</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Mining Implementation Data</td>
<td>56</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Using the Framework in the Software Development Life Cycle</td>
<td>57</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusions</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>Architecture and Design</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Architectural Design Decisions</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>The Java Specific Architecture of the Framework</td>
<td>65</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Logical Layers</td>
<td>66</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Component Decomposition</td>
<td>70</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Physical Deployment</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>Choice of Modelling Notation</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation of the Design</td>
<td>74</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusions</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>Implementation</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Shared Components</td>
<td>78</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 A simple example of architecture erosion. ........................................ 5

2.1 A classification of available techniques for controlling architecture erosion. ... 15

3.1 The architectural moulding concept .................................................... 48

3.2 Architecture specification driving both the AE and system implementation. ...... 49

3.3 Usage scenarios of the architecture environment. The host is a different instance in each case .......................................................... 50

3.4 Process for checking architecture conformance in the proposed framework. ....... 53

3.5 Tagging architecture elements with mappings to implementation ................ 55

3.6 The event-driven process used for gathering implementation data. ................ 57

4.1 The general architecture of PANDArch showing the two-stage conformance evaluation pipeline. Three evaluators are used for checking conformance in the target application with the runtime evaluator, in this case, configured with three runtime monitors to receive events arriving from three separate processes of the executing application. In addition to the monitors and evaluators, the architecture also includes two distiller components for extracting properties from the architecture specification of the target application. The extracted conformance rules are fed to all three evaluators, although for improved clarity, the diagram depicts only the runtime evaluator accepting information from the distillers. .................................. 63

4.2 The layered architecture used in the Java implementation of the PANDArch framework. ................................................................. 66

4.3 The component decomposition view of the PANDArch architecture. This diagram shows the main components available in the current implementation of the framework and their interactions ................................................. 71

4.4 Deployment architecture of the PANDArch framework .............................. 73

5.1 The interface-oriented design used for implementing generic events. Note that the notation used for specifying the template parameter is Java specific. ................. 80

5.2 The interface-oriented design used for implementing monitor components. Note that the notation used for specifying the template parameter is Java specific ..................... 82

5.3 The internal operations of the Java runtime monitor .................................. 85

5.4 Use of instrumentation probes to capture method-entry events in the Java agent implementation. The shaded pseudo-bytecode instructions in the two application classes represent the injected the probes. In the current implementation of the Java runtime monitor, only method-entry events captured using probes. The technique, however, can be used for certain types other event as well ............................. 89
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>The interface-oriented design used for implementing evaluator components. Note that the notation used for specifying template parameters is Java specific.</td>
<td>94</td>
</tr>
<tr>
<td>5.6</td>
<td>The interface-oriented design used for implementing conformance rules. Note that the notation used for specifying template parameters is Java specific.</td>
<td>95</td>
</tr>
<tr>
<td>5.7</td>
<td>The interface oriented design used for implementing rules generator (i.e. distiller) components. Note that the notation used for specifying template parameters is Java specific.</td>
<td>98</td>
</tr>
<tr>
<td>6.1</td>
<td>The recovered component architecture of Jackrabbit.</td>
<td>121</td>
</tr>
<tr>
<td>6.2</td>
<td>The component dependency matrix for the recovered Jackrabbit architecture.</td>
<td>122</td>
</tr>
<tr>
<td>6.3</td>
<td>Graph showing the performance of Jackrabbit as the count of architecture violations increases.</td>
<td>129</td>
</tr>
<tr>
<td>6.4</td>
<td>The recovered component architecture of jEdit.</td>
<td>131</td>
</tr>
<tr>
<td>6.5</td>
<td>The component dependency matrices for the recovered jEdit architecture.</td>
<td>132</td>
</tr>
<tr>
<td>6.6</td>
<td>The recovered component architecture of Hibernate.</td>
<td>140</td>
</tr>
<tr>
<td>6.7</td>
<td>The component dependency matrices for the recovered Hibernate architecture.</td>
<td>141</td>
</tr>
<tr>
<td>6.8</td>
<td>Grouped component links for evaluating the PANDArch with Hibernate.</td>
<td>145</td>
</tr>
<tr>
<td>6.9</td>
<td>Performance of Hibernate as the number of architecture violations increases.</td>
<td>147</td>
</tr>
<tr>
<td>A.1</td>
<td>The Grasp conceptual model formalised in UML.</td>
<td>173</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Controlling architecture erosion with process-oriented strategies .................. 21
2.2 Controlling architecture erosion with design enforcement strategies ............... 27
2.3 Controlling architecture erosion with architecture restoration strategies .......... 38

4.1 Currently implemented components in the framework and possible future components. 72
4.2 The evaluation of the design of PANDArch against guiding principles (part 1) ....... 74
4.3 The evaluation of the design of PANDArch against guiding principles (part 2) ....... 75

5.1 Java namespace assignments for representing layers in the PANDArch architecture. 78
5.2 Implementation descriptors supported by the framework. ........................... 79
5.3 Java namespace assignments for implementing monitor components ................. 81
5.4 The list of event types supported by runtime monitors ............................... 84
5.5 Java namespace assignments for implementing evaluator components .............. 93
5.6 List of conformance rules currently implemented in PANDArch .................... 96
5.7 Java namespace assignments for implementing rules generator (i.e. distiller) ....... 98
5.8 Mapping configurations supported by the Grasp rules generator ...................... 101
5.9 Mapping configurations supported by the Grasp rules generator ...................... 101
5.10 Command line arguments for launching PANDArch ................................. 102
5.11 The evaluation of the implementation of PANDArch against guiding principles (part 1) 103
5.12 The evaluation of the implementation of PANDArch against guiding principles (part 2) 104

6.1 System specification of the environment employed for executing evaluation tests. 113
6.2 Test cases for verifying the EvalNamespace rule ........................................ 115
6.3 Test cases for verifying the EvalType rule .............................................. 115
6.4 Test cases for verifying the EvalProvidesInterface rule .............................. 116
6.5 Test cases for verifying the EvalRequiresInterface rule .............................. 117
6.6 Test cases for verifying the EvalLinkInteraction rule ............................... 118
6.7 Summarised results of unit testing ..................................................... 119
6.8 Jackrabbit deployment configuration used for evaluating the framework .......... 120
6.9 A comparison of performance impact between the use of JDI and instrumentation probes to capture runtime events. Execution times are in microseconds ....... 128
6.10 The MD5 digests of the Jackrabbit code verifying the non-intrusiveness of PANDArch 128
6.11 jEdit deployment configuration used for evaluating the framework ................. 130
6.12 Test scenarios used for evaluating the framework with jEdit ....................... 135
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.13</td>
<td>The MD5 digests of the jEdit code verifying non-intrusiveness during conformance checking.</td>
<td>138</td>
</tr>
<tr>
<td>6.14</td>
<td>Hibernate deployment configuration used for framework evaluation.</td>
<td>139</td>
</tr>
<tr>
<td>6.15</td>
<td>Performance results from executing the Hibernate test harness with PANDArch. The harness outwith PANDArch is run once, with the execution time from this test used as the baseline for comparing others. All times are in milliseconds.</td>
<td>146</td>
</tr>
<tr>
<td>6.16</td>
<td>The MD5 digests of the Jackrabbit code verifying the non-intrusiveness of PANDArch.</td>
<td>147</td>
</tr>
<tr>
<td>A.1</td>
<td>Reserved keywords in Grasp ADL</td>
<td>170</td>
</tr>
<tr>
<td>A.2</td>
<td>Operators in Grasp ADL</td>
<td>172</td>
</tr>
</tbody>
</table>
## Code Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The grammar specification of the agent protocol.</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Using Grasp to specify architecture and implementation mappings</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Grasp specification of the recovered Jackrabbit architecture.</td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>Grasp specification of the recovered jEdit architecture.</td>
<td>133</td>
</tr>
<tr>
<td>6.3</td>
<td>Grasp specification of the recovered Hibernate architecture.</td>
<td>142</td>
</tr>
<tr>
<td>A.1</td>
<td>Grasp specification of a partial EFT system architecture.</td>
<td>174</td>
</tr>
</tbody>
</table>
ACRONYMS

ADL architecture description language
ATAM Architecture Trade-off Analysis Method
CbC construction-by-configuration
CP-net Coloured Petri net
DSM dependency structure matrix
EBNF Extended Backus-Naur Form
IDE integrated development environment
JRE Java runtime environment
JVM Java virtual machine
MDA model driven architecture
MVC Model-View-Controller
SCM software configuration management/manager
SSAM Scenario-based Architecture Analysis Method
UML Unified Modelling Language
VM virtual machine
1.1 Overview

Software systems are under constant pressure to adapt to changing requirements, technologies and organisational landscapes. At the same time these systems must continue to deliver services to users at expected levels of performance. Often, modifications made to a software system over a period of time damage its integrity and violate its design principles. As a result, the system may exhibit a tendency towards diminishing returns as further enhancements are made. Such software is neither useful for its intended purpose nor is it economically viable to maintain. Eroded software often goes through a process of re-engineering, though this may not always yield the expected benefits. The alternative is to build a replacement system from scratch, which would require a sizeable investment. Moreover, software systems have become key assets in organisations that sell them and in those which use them. Ensuring that these systems perform as expected throughout their intended lifetime is vital for the sustainability of these organisations.

Software erosion is not a new concept. Parnas [1] argues that software aging is inevitable but nevertheless can be controlled or even reversed. He highlights the causes of software aging as obsolescence, incompetent maintenance engineering work and effects of residual bugs in long running systems. However, later works in this area such as those carried out by Huang et al. [2] and Grottke et al. [3] define software aging as the gradual degradation of performance in executing software processes due to changes in the runtime state (e.g. memory leaks). This thesis regards erosion as the overall deterioration of the engineering quality (see Section 1.2 on terminology) of a software system during its evolution, and often a contributory factor to the kind of software aging studied by Huang et al. and Grottke et al.
The impact of erosion is profound when the damage affects the architecture of a software system. Software architecture [4, 5] establishes a crucial foundation for the systematic development and evolution of software and forms a cycle of influence with the organisation to which the system belongs [6]. It provides a high level model of the structure and behaviour of a system in terms of its constituent elements and their interactions with one another as well as with their operating environment. Architecture also encompasses rationale, which forms the basis for the reasoning and intent of its designers [4].

This thesis focuses on architecture erosion which plays a large role in accelerating software erosion. Architecture erosion is often the result of system modifications that disregard its fundamental architecture properties. Although a single violation is unlikely to have an adverse effect on the system, the accumulation of violations over time can eventually create a mismatch between the implemented software and its architecture. The effects of architecture erosion tend to be systemwide, and therefore harder to rectify than other forms of software degeneration such as inferior code quality which can be corrected through software refactoring techniques [7].

The deterioration of software systems over time has been widely discussed since the late '60s debate on the “software crisis” [8, 9]. Evolving software systems gradually become more complex and harder to maintain unless deliberate attempts are made to reduce this complexity [10]. At the same time, these software systems have to be continually upgraded to adapt to changing domain models, accommodate new user requirements and maintain acceptable levels of performance [10]. Therefore, complexity becomes a necessary evil to prevent software from becoming obsolete too soon.

Complexity, however, makes it harder to understand and change a design, leading to programmers making engineering decisions that damage the architectural integrity of the system. Eventually, the accumulation of architectural violations can make the software completely untenable. Lack of rigorous design documentation and poor understanding of design fundamentals make a complex system even harder to maintain [1]. Furthermore, software architectures that have not been designed to accommodate change tend to erode sooner [1].

Architecture erosion can also result from contemporary software engineering practices. Architectural mismatches can arise in component-based software engineering (CBSE) due to assumptions that reusable components make about their host environment [11]. New challenges in CBSE like trust, re-configurability and dependability create enormous demands on the architectures of evolving software systems [12]. In addition, modern iterative software development processes (such as agile programming methods) may cause the occurrence of architecture erosion sooner rather than later because they place less emphasis on upfront architectural design [13].
Industry case studies point towards widespread occurrence of architecture erosion in real-world software. Eick et al. [14] present a study of a large, 15-year old telecommunication software system developed in C/C++. They derive a set of indices from change request data to measure the extent of erosion and its impact on the system. Although termed code decay, module level changes may have architectural level impact. The study shows a clear relationship between erosion and an increased effort to implement changes, an increased number of induced defects during changes, increased coupling and reduced modularity. In another commonly cited example of architecture erosion, Godfrey and Lee [15] describe their analysis of the extracted architectures of the Mozilla web browser (which subsequently evolved into Firefox) and the VIM text editor. Both these software products showed a large number of undesirable interdependencies among their core subsystems. In fact, the badly eroded architecture of Mozilla caused significant delays in the release of the product and forced developers to rewrite some of its core modules from scratch [13, 16]. Other similar findings have been reported on popular open source projects such as FindBugs [17], Ant [18] and version 2.4 of the Linux kernel [13].

The impact of architecture erosion is far reaching and has an associated cost. In the worst case an eroded software system requires complete re-development. Even if a system does not become unusable, erosion makes software more susceptible to defects, incurs high maintenance costs, could degrades performance and lead to more erosion. Consequently, the system may lose its value, usefulness, technical dominance and market share, as experienced by the Mozilla web browser. Mozilla lost its leading position and a large portion of the Internet browser market to Microsoft’s Internet Explorer mostly due to its inability to introduce new features on time [16].

The thesis investigates the possibility of employing architecture conformance monitoring within a larger agenda for controlling architecture erosion through effective and practical measures. The proposed approach introduces the notion of an "architecture environment" within which a software implementation is allowed to develop and evolve. This notion is governed by three fundamental principles. An architecture environment is – 1) pluggable, so that it can easily be attached to and detached from the software, 2) non-intrusive, because it does not modify the source code or the compiled code of the software, and 3) customised, because the properties of the environment are not the same for every software system. An architecture specification of a software system serves as the basis for generating its environment, which thereafter functions as a virtual enclosure for guiding the software implementation towards architecture conformance. The generated environment is composed of the necessary architecture properties and the required tools for checking conformance. From a conceptual standpoint, an architecture environment monitors both static and dynamic conformance and is therefore able to guide an implementation to conform to the architecture during both development and execution.
Software architectures are often described using multiple views, each representing a different perspective of the system under consideration [19, 20]. A given architecture view conveys specific information about either the structure of the system, its internal behaviour or the interactions with its environment. While every view can potentially contribute towards controlling erosion, this thesis examines the use of runtime structural views to achieve the said objective. The runtime view of a system is primarily a specification of the architecturally significant structural elements that exist during system execution. A runtime view may also depict communication constraints between these runtime structures. As runtime architecture properties have considerable influence on the performance, usability, and quality-of-service aspects of the software, ensuring their conformance in the implementation is important.

The work described in the thesis explores the feasibility of runtime conformance monitoring using runtime architectural views as the basis. In order to achieve this objective, the proposed approach expects the architecture specification to provide information that associate properties in the architecture to the code implementing those properties. The exercise of mapping architectural constructs to implementation is known as bridging the abstraction gap, and plays a crucial role in architecture conformance monitoring.

The thesis shows that, by comparing implementation metadata found in runtime events against rules derived from a runtime view of the architecture, violations of some architecture properties in the software can be detected. Runtime events occur due to changing state in executing program code. An example of a runtime event is a simple method invocation or a more elaborate object creation in an object-oriented program. For these events to be useful in architecture conformance checking, they need to carry information about their source, such as method or class names. Compiled native code generally does not have this information unless the compiler has included runtime type information. Virtual machines such as the Java virtual machine (JVM), on the other hand, generate runtime events tagged with rich type information often useful for carrying out runtime architecture conformance checking.

The notion of the architecture environment establishes the conceptual underpinnings for the design and implementation of the runtime conformance checking framework described in this thesis. The framework, which itself has been implemented in Java, is able to check conformance of Java software by inspecting their runtime events. It employs the Grasp [21, 22] architecture description language (ADL) for reading architecture specifications and their implementation mappings. However, the framework is designed to be extensible for supporting other types of event sources as well as architectures specified in different notations. This extensibility is further strengthened by introducing measures in the design to represent events and related metadata in a platform neutral manner.
The thesis takes a two-phase approach for evaluating the Java implementation of the runtime conformance monitoring framework. The primary conformance checking elements of the framework are first unit-tested using a simple purpose-built Grasp architecture and its corresponding Java implementation. In the second phase, the framework is evaluated against three well regarded open source software products of substantial size and complexity. Experiments are designed to gauge both the effective detection of conformance violations and performance implications during conformance checking on each of the test subjects. Their results are analysed to determine whether the framework has achieved its core objectives and is a viable approach for architecture conformance monitoring.

1.2 Terminology

This thesis defines architecture erosion as the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture. The implemented architecture is the high-level design that has been realised in the program code and other implementation artefacts. The term concrete architecture [23] also refers to the implemented architecture. The intended architecture, which is also known as the conceptual architecture [23] or the planned architecture [24], is the outcome of the architecture design process. The divergence itself is not caused by malicious human actions (though it could be intentional), but rather by routine maintenance and enhancement work typical for an evolving software system. An example of this notion of architecture erosion is diagrammatically shown in Figure 1.1. The boxes in the two diagrams depict components in the architecture, while lines represent component containment. For instance, the root component S in Figure 1.1(a) consists of components M₁, M₂ and M₃, while component M₄ is contained within M₂.

![Diagram of architecture erosion](image-url)

**Figure 1.1:** A simple example of architecture erosion.
Architecture erosion leads to the gradual deterioration of the *engineering quality* of software systems. In this thesis, engineering quality is defined as a subsumption of architectural integrity (i.e. completeness, correctness and consistency), conformance to *quality attribute* requirements [6], adoption of sound software engineering principles (e.g. modularity, low coupling, etc.), and adherence to architecture rationale and corresponding design decisions. The loss of engineering quality makes software hard to adapt [4] and, as a result, may become less useful or obsolete sooner than expected.

The engineering quality of a system may not always equate to the quality of system performance. A well-performing system may have a badly eroded architecture. However, such a system is extremely fragile and has a high risk of breaking down whenever modifications are made. Similarly, a well-engineered system may not perform as expected because its architecture has not been designed to cater to user requirements. An architecture incompatible with user requirements is not considered an eroded architecture. However, such an architecture erodes faster if developers resort to ad-hoc repairs to satisfy requirements instead of addressing fundamental design issues. On the other hand, it is possible to progressively improve a flawed architecture with carefully crafted upgrades to the system. The topic of correcting design flaws in the intended architecture during or after implementation is not within the scope of this thesis.

### 1.3 Hypotheses

The thesis proposes the following hypotheses.

1. The conformance of static and dynamic architecture properties of a software system can be checked by containing the implementation within a pluggable, non-intrusive and customised architecture environment derived from an architecture specification of the software. It is possible to build a software framework that reifies the notion of an architecture environment, has the aforementioned attributes and performs architecture conformance checking.

2. Useful implementation details required for checking conformance of some static and dynamic architecture properties of a software system can be extracted from the instrumentation and debugging data provided by the hosting virtual machine (VM) during the execution of that software.

3. Architecture conformance of a software system can be verified during execution without impairing the functionality or the testability of the software.
1.4 Motivation

The motivation behind the three primary drivers of the approach are discussed below.

1.4.1 The need for Pluggability

As discussed in detail later in Chapter 2, approaches that link architecture models to implementations for the purpose of monitoring architecture conformance rely on runtime frameworks. These monitoring frameworks impose an additional layer of overhead on software that is often built around industry standard infrastructure such as middleware and persistence frameworks, potentially increasing deployment complexity and affecting system performance. In order to avoid these drawbacks, an application being monitored by a conformance framework should also be able to operate outwith that framework. Therefore, a framework that can be easily attached to the application for checking architecture conformance as part of a software testing process, and thereafter be detached prior to deployment should encourage stronger adoption.

1.4.2 The need for Non-intrusiveness

Mainstream programming languages such as Java, C# or C++ are not sufficiently expressive to represent architectural concepts. Hence, programs written in these languages are unable to interpret the architectural designs and properties they implement. To overcome this problem, techniques that link architecture with implementation inject special code or annotations into source programs. These injected probes provide feedback to a monitoring framework that checks the conformance of the running software against the architecture. However, a programmer could intentionally or accidentally remove these probes rendering the conformance monitoring process ineffective. In addition, altering source code has the potential to cause performance and security issues in the software. Therefore, an effective architecture conformance assurance framework should follow a strategy that refrains from intruding or altering the source code of the subject software.

A non-intrusive strategy also extends well towards software developed through construction-by-configuration (CbC) [25]. Developers following a CbC process assemble together components they have either developed in-house or procured elsewhere. The source code for these components may not always be available and/or their internal behaviour not well documented. Therefore, injecting probes at the source level into these components runs the risk of creating undesirable side effects or even breaking their functionality altogether.
1.4.3 The need for Customisability

Every software system has its own architecture. While architecture patterns, styles and other best practices are often applied in the design of a software, the architecture of a given system has unique properties. These properties not only reflect the primary requirements of the system, but they also embody the various design trade-offs made in order to satisfy any contending requirements. Hence a conformance monitoring approach that depends on some universal set of architecture properties is unlikely to be an effective tool for controlling erosion. A conformance monitor should instead derive properties from the architecture specific to the software under consideration for checking its conformance.
1.5 Main Contributions

The key contributions of this thesis are listed below.

1. A comprehensive survey of currently available approaches, developed in academia as well as in industry, for controlling erosion, and a classification framework for categorising these approaches. Other comparable surveys are smaller in scope and do not classify the different techniques similar to the survey presented in this thesis.

2. The conceptualisation, design and implementation of an architecture conformance monitoring strategy that is driven by the architecture specification of the software being monitored. Architecture properties for validating the implementation for architecture conformance are accepted only from this specification.

3. The conceptualisation of a simple and extensible mechanism for mapping architecture elements to their corresponding programming constructs. This mapping strategy forms a cornerstone in the architecture conformance monitoring approach described in the thesis. In the implemented framework, mapping details and the architecture model of the subject software are described within a single Grasp specification. Furthermore, the mapping strategy allows the conformance checker to apply tolerance for incomplete areas in the architecture specifications.

4. The conceptualisation of conformance rules. Representing the foundation of the conformance checker, these rules contain primitive constraints derived from architecture properties and their mappings to the implementation, which can be validated against static or executing code. Conformance rules are independent of the architecture notation or modelling paradigm from which they are extracted, and the framework can be extended by adding conformance rules to support new architecture concepts.

5. The conceptualisation of a generalised event-driven mechanism for mining implementation metadata. Although in its present implementation the framework uses only runtime events to monitor conformance, the overall strategy and the core framework design employs events to collect all forms of implementation details including static code properties.

6. The concept of genericising the representation of events and implementation metadata for evaluating conformance rules. This design strategy makes conformance rules independent of any platform-specific or technology-specific event types and metadata extracted from an implementation.
7. The design of a **two-stage processing pipeline** for implementing the conformance checker. Monitor components in the first stage collect events from various platform specific event sources, and translate them to their generic counterparts. These generic events are consumed by evaluator components in the second stage of the pipeline, which use them for checking conformance rules and, in turn, detecting conformance violations. The design allows multiple pairs of monitors and evaluators to participate simultaneously in single conformance checking session.

8. The implementation of **non-persistent, stateless and high-performance instrumentation probes** to generate runtime events having useful information for monitoring architecture conformance of Java applications. Tests show that the performance impact of these probes on the monitored application is significantly lower than that of the standard Java event capturing technology.

9. The design and implementation of a runtime architecture conformance monitoring framework that is **pluggable**, whereby the subject software is attached to the framework only for the duration of checking conformance. To check conformance of the software, it needs to be launched through the framework. Once the conformance checking has been completed, the software operates in the usual manner with no links to the framework.

10. The design and implementation of a runtime architecture conformance monitoring framework that does not alter the source code or the on-disk compiled code of the subject software. The framework does not require access to the source code of the software during conformance checking, nor does it require the software to be recompiled afterwards.

11. The design and implementation of a runtime architecture conformance monitoring framework that parses the architecture specification of the subject software to automatically generate conformance rules required for checking architecture conformance. While the design can be extended to support any suitable architecture notation, the current framework implementation is capable of processing Grasp architecture models.

12. The design and implementation of a runtime architecture conformance monitoring framework that automatically detects and notifies conformance violations on-the-fly of executing Java software. In addition, some conformance violations are notified immediately after the subject application has terminated execution.
1.6 Thesis Organisation

The remainder of the thesis is organised as follows.

Chapter 2 presents a comprehensive background survey of available approaches for controlling architecture erosion. These approaches are discussed within the context of a classification framework developed specifically for this purpose. An analysis of the efficacy and industry adoption of each strategy in the classification framework are provided.

Chapter 3 introduces the overall strategy adopted for building the architecture conformance monitoring framework. The chapter covers the conceptual foundations upon which the monitoring framework is designed and implemented, along with the core principles that guide all aspects of the framework, from design to evaluation.

Chapter 4 details the architecture and the high-level design of the conformance monitoring framework. While the design in general is platform independent, some parts are intended for monitoring conformance of Java applications. The chapter also discusses the modelling notation, i.e. the Grasp ADL, employed for specifying architectures.

Chapter 5 discusses the implementation of the framework in Java. Significant emphasis is placed on discussing the techniques adopted for minimising the performance impact on the software being monitored.

Chapter 6 presents the evaluation of the framework. Unit tests for verifying that the framework produces the expected output along with experiments for evaluating its effectiveness and performance using three open source software products are discussed in this chapter.

Chapter 7 presents a critical assessment of the work covered in the thesis. In particular, the framework is assessed against its core principles to determine how well the objectives have been achieved.

Chapter 8 summarises the thesis and discusses possibilities for future work.

Appendix A presents a specification of the Grasp architecture description language. Grasp is employed for specifying architectures and mappings their to implementation constructs as part of the conformance monitoring approach proposed in the thesis.
1.7 Publications

Peer reviewed publications

  
  (contributes to chapters 3, 4, 5 and 6)

  
  (contributes to chapters 1 and 2)

  
  (contributes to chapters 4 and 5 and Appendix A)

  
  (contributes to Appendix A)

Other publications

  
  (contributes to Appendix A, although the Grasp language has significantly evolved since its original specification)
Numerous approaches have been proposed over the years either to prevent architecture erosion or to detect and restore eroded architectures. This chapter presents a survey of these approaches, which include techniques, tools and processes. They are classified primarily into three generic categories that attempt to minimise, prevent and repair architecture erosion. Within these broad categories, each approach is further broken down to reflect the high-level strategies adopted to tackle erosion. Some of these strategies in turn contain sub-categories under which survey results are presented. The merits and weaknesses of each strategy are discussed, with the argument that no single strategy can address the problem of erosion at present. The chapter concludes by presenting a case for further work in developing a holistic and practical approach for controlling architecture erosion.

2.1 Introduction

Architecture erosion and its effects are widely discussed in literature. Perry and Wolf [4] differentiate *architecture erosion* from *architecture drift* as follows: erosion results from violating architectural principles while drift is caused by insensitivity to the architecture. As the underlying causes for both are treated equally, the thesis does not consider this difference for the purpose of the survey. Additionally, the notion of software architecture erosion is discussed using a number of different terms such as architectural degeneration [24], software erosion [18], design erosion [13], architectural decay [26], design decay [27], code decay [14, 28] and software entropy [29]. Although some of these terms imply that erosion occurs at different levels of abstraction (for instance code decay may potentially be considered insignificant at the architectural level), the underlying view in each discussion is that software degeneration is a consequence of changes that violate design principles.
Mechanisms for controlling architecture erosion have traditionally centred on architecture repair as evident from the large body of published work (e.g. [30, 31, 32]). Architecture repair typically involves using reverse engineering techniques to extract the implemented architecture from source artefacts (recovery), hypothesising its intended architecture (discovery) and applying fixes to the eroded parts of the implementation (reconciliation). Subsequent research in this area focuses more on erosion prevention schemes (e.g. Mae [33] and ArchJava [34]) and explores concepts from other areas of computer science such as artificial intelligence to increase the accuracy of architecture discovery and recovery techniques (e.g. Bayesian learning-based recovery [35]).

This chapter presents a classification of strategies for controlling architecture erosion and a survey of currently available approaches under each category in the classification. The remainder of the chapter is structured as follows. Section 2.2 introduces the classification scheme while sections 2.3 to 2.8 present the survey results under this classification. For each approach, evidence of adoption is provided where available, and a discussion of efficacy and cost-benefit analysis based on own experience and material from literature. Section 2.9 concludes the chapter with a summary of the current state-of-the-art and presents a justification for the work in this thesis.
2.2 Classification

Existing approaches for controlling architecture erosion are first classified into three broad categories depending on whether they attempt to *minimise*, *prevent* or *repair* erosion. Each of these categories contains one or more sub-categories based on the high-level strategies used to realise its goal. Some of these sub-categories are further divided indicating the specific strategy adopted. Figure 2.1 shows a graphical representation of this classification framework.

At the top-most level, the *minimise* approach contains those strategies which are targeted towards limiting the occurrence and impact of architecture erosion but may not be effective in eliminating it. On the other hand, strategies that fall into the *prevent* category aim to stop erosion. The third group of strategies attempt to *repair* the damage caused by erosion and reconcile the implementation with its architecture.

The second level of the classification framework lists specific strategies under each of the top level categories. These are briefly explained below.
CHAPTER 2. BACKGROUND

- **Minimise**
  - **Process-oriented Architecture Compliance** includes software engineering processes that ensure architecture compliance during development and maintenance activities.
  - **Architecture Evolution Management** covers methods available for managing the evolution of architecture specifications in parallel with implementation artefacts.
  - **Architecture Design Enforcement** incorporates methods and tools available for transforming architectural models into implementation.

- **Prevent**
  - **Architecture to Implementation Linkage** includes mechanisms that associate or embed architectural models within source code and support the ability to monitor architecture compliance at runtime.
  - **Self-adaptation** technologies enable systems to reconfigure themselves to align with their architectures after a change has been made to either the implementation or the runtime state.
  - **Architecture Conformance Monitoring** involves static and runtime conformance checking by comparing information retrieved from static analysis or execution-time events against a specification of the architecture. The work presented in this thesis comes under this category.

- **Repair**
  - **Architecture Recovery** involves extracting the implemented architecture from source code and other artefacts.
  - **Architecture Discovery** techniques are useful for eliciting the intended architecture from emergent system properties and other means in the absence of architecture documentation.
  - **Architecture Reconciliation** methods help reduce the gap between the implementation and the intended architecture of a software system.

As these three repair strategies can be effective when applied together, they are discussed under a common theme titled “Architecture Restoration” in Section 2.8.

The next seven sections present the survey results for each of the above strategies. Each section concludes with an analysis of the adoption and efficacy of the strategies discussed.
2.3 Process-oriented Architecture Compliance

Architecture compliance, vital for minimising architecture erosion, is often attempted through process-centric activities in software development. Literature identifies a number of reasons for architecture erosion that relate directly to human and organisational factors. Parnas [1] highlights inadequate design documentation, misunderstood design principles and poor developer training as key triggers of erosion. Similarly, Eick et al. [14] argue that vague requirements specifications, poor architectural designs and programmer variability, among other factors, lead to architecture erosion.

Checks for assuring architecture compliance during both development and maintenance are built into most software development processes such as the Rationale Unified Process [36, 37] and the Open Unified Process (part of the Eclipse Process Framework) [38]. These processes enforce architecture reviews to ensure that the architecture meets user requirements, design reviews to verify that designs adhere to architectural guidelines and code reviews to check that architectural principles are not violated in program code. Similarly, change requests should be approved by architects to ensure system updates are agreeable with the intended architecture.

To address the issue of programmer variability, software processes often include some form of skill-gap analysis to identify the training needs of new team members. Junior team members are also paired with senior developers as a scalable mentoring and review mechanism in large project teams. Some of these process activities are often supplemented by automation tools in order to increase productivity and reduce human-induced errors.

Software engineering processes incorporate the following strategies to enhance the effectiveness of controlling architecture erosion:

- Architecture Design Documentation,
- Architecture Analysis,
- Architecture Conformance Monitoring, and
- Dependency Analysis

The survey results are discussed under the above categories in sections 2.3.1 to 2.3.4.
2.3.1 Architecture Design Documentation

Addressing Parnas’ concern that poor design documentation may lead to software degeneration, mature software processes prescribe uniform and rigorous documenting of architecture specifications. For them to be useful in understanding the architecture, documented specifications have to be both sufficiently detailed and adequately abstract [20]. While architecture documentation does not necessarily require a process framework, this activity should ideally be formalised for ensuring that documentation artefacts are up to date and well catalogued.

Popular architecture documentation techniques focus primarily on recording system structure, its interactions with the operating environment and its transactions with the organisation. They use a number of views, viewpoints (i.e. a perspective of a view) and mappings among views to capture different aspects of an architecture. The 4+1 View Model [19] and Views and Beyond [20] provide foundations for view-based documentation of architecture while Bachmann et al. [39] describe a technique for documenting system behaviour using notations such as state charts, ROOMcharts and collaboration diagrams.

An architecture description language (ADL) [40] offers a precise notation with well-defined semantics for describing architectures. A number of ADLs have been developed over the years, each with its own characteristics and capabilities geared towards addressing certain architectural aspects. However, most ADLs are conceptually based on the structural architecture primitives of components, connectors, interfaces and configurations [40]. Examples of well-known ADLs include Acme [41], Darwin [42] and xADL [43] all of which model structural properties while Rapide [44] was developed to model dynamic properties of systems. Further discussions covering numerous ADLs can be found in a survey by Clements [45] and in a classification framework for ADLs by Medvidovic and Taylor [40].

The IEEE 1471-2000 standard (also known as ISO 42010), titled the Recommended Practice for Architectural Description of Software-Intensive Systems [46], provides a comprehensive set of guidelines for describing software architectures. This standard prescribes how an architecture should be documented and what should ideally be included in such a description. However, it was not the aim of IEEE 1471 to suggest an ideal architecture or a notation for specifying architectures.

A few techniques for documenting architectural design decisions and rationale can be found in the literature. Tyree and Akerman [47] suggest the use of document templates to systematically capture architecture design decisions and their rationale. A post-design approach for capturing design decisions is discussed by Harrison et al. [48] where patterns in the architecture are analysed to capture the rationale that motivated them.
2.3.2 Architecture Analysis

Architecture analysis methods formalise architecture evaluation and review. The Architecture Trade-off Analysis Method (ATAM) [49] facilitates evaluation of an architecture with respect to requirements, quality attributes, design decisions and design trade-offs. The outcome of an ATAM process identifies risks, sensitivity points and non-risks in the architecture. Industrial case studies (e.g. the Linux case study by Bowman et al. [50] and a study of the Space Station operations control software by Leitch and Strouli [51]) show that these risks and sensitivity points in architectures are more susceptible to erosion.

A number of other similar architecture analysis methods are available including the Scenario-based Architecture Analysis Method (SSAM) [52] and the Software Architecture Evaluation Model [53]. Additionally, a concern-based analysis method is introduced by Tekinerdogan et al. [54]. The technique employs dependency structure matrix (DSM) models of the architecture to derive important concerns using a mapping scheme. These concerns form the basis for further analysis and refactoring. Lindvall et al. [55] and Tvedt et al. [56] describe a process for evaluating the maintainability of an architecture. They propose examining the amount of inter-module coupling in the implemented architecture to reduce violations when further code changes are made. Finally, an analysis technique using Coloured Petri nets (CP-nets) for validating architecture quality attributes is proposed by Fukuzawa and Saeki [57].

2.3.3 Architecture Conformance Monitoring

An important process-oriented architecture compliance assurance strategy is performing audit checks on the implementation. A software process that includes regular conformance monitoring activities should be able to detect when an implementation diverges from its intended architecture.

With reflexion models [58][59], an extracted model of the implemented architecture is compared against a hypothetical model of the intended architecture using an intermediary manually created mapping scheme. The hypothetical architecture, in the absence of documented architecture specifications, is usually modelled by observing external system behaviour. The computed outcome is the reflexion model that identifies places in the implemented architecture where there are deviations or omissions from the hypothetical design.

Rosik et al. [60] present an adaptation of reflexion models called inverted reflexion models. This method presumes the availability of the intended architecture and therefore not hypothesised. A model of the implemented architecture is repeatedly refined and compared against the intended architecture. Thus, inverted reflexion models appear better suited for conformance monitoring while generic reflexion models are useful for architecture recovery.
Postma [61] introduces a method similar to reflexion models, but uses relationship constraints among architecturally significant modules to verify conformance between implementation and its intended architecture. The implemented architecture is derived from source code, but the mappings and constraints are generated by tools using formal models of the intended architecture and input from architects.

DiscoTect [62] [63] is an approach for extracting the architecture of a system by examining events that occur while the software executes. This technique dynamically collates runtime traces in order to identify architecturally relevant elements or behaviours, and looks up a mapping scheme described in a custom notation named DiscoSTEP. This mapping scheme is built with the knowledge of the intended architecture, and serves for bridging the abstraction gap. Once the process is complete, views of the the extracted architecture are manually examined to identify deviations from the intended architecture. A later work by Ganesan et al. [64] extends DiscoTect with a mapping scheme based on hierarchical CP-nets. The authors claim that CP-nets are more reusable and have better tool support than DiscoSTEP.

In addition to these methods, engineering processes can also utilise architecture recovery techniques discussed Section 2.8.2 for the purpose of conformance checking. A recovered implemented architecture can be manually verified against the documented intended architecture of the system, if one exists.

2.3.4 Dependency Analysis

A growing body of work exists for checking dependencies among modules and classes using implementation artefacts [65, 66, 67, 68, 69, 70, 71, 72]. Dependency analysis can expose violations of certain architecture constraints such as inter-module communication rules in a layered architecture, thus identifying points of potential erosion. Similar to conformance monitoring approaches, these tools enable the automation of architecture compliance assurance activities.

A method based on DSMs developed by Sangal et al. [73] allows dependencies between software modules (or subsystems) to be modelled using a matrix where columns of the matrix specify dependents and rows contain modules that are depended upon. An interesting capability of this technique is that simple design rules like “module X should call module Y” can be applied to the matrix and automatically validated against the implementation. The dependency chart in the matrix can be re-organised using partitioning algorithms to expose the “most used” modules and “most provided” modules which are often useful in understanding the layering constraints of an architecture.
SonarJ [66] uses XML specifications of the intended architecture to detect dependency violations during and after development. A sister product named Sotograph [67] performs architecture conformance validation using a repository of implementation data and monitors the structural evolution of architectural models between versions of a software product. Structure101 [68] provides a visual representation of module dependencies at various levels of abstractions of an implemented architecture allowing architects to identify where undesirable dependencies may have occurred.

A number of other tools such as Axivion Bauhaus [69], Klocwork Architect [70], Coverity [71] and JDepend [72] provide various forms of dependency analysis, quality metrics and visualisation support that are all useful in architecture compliance assurance. Finally, Huynh et al. [74] present a technique for compliance checking using a DSM representation of source artefacts and a DSM model of the intended architecture. This technique is targeted towards checking violations of modularity principles in architectural designs.

### 2.3.5 Discussion

Table 2.1 highlights the contribution of each strategy in the process-oriented category towards controlling architecture erosion.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Contribution towards controlling erosion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Design Documentation</td>
<td>▶ Records architecture design and rationale with the intent of disseminating architectural knowledge to a wider audience and provides a point of reference for developers throughout system evolution.</td>
</tr>
<tr>
<td>Architecture Analysis</td>
<td>▶ Uncovers weaknesses in the intended architecture, in particular, sensitive points which can be easily violated in the implementation.</td>
</tr>
<tr>
<td>Architecture Conformance Monitoring</td>
<td>▶ Establishes the means to verify whether the implementation is faithful to the intended architecture during both the development and subsequent maintenance phases of a system.</td>
</tr>
<tr>
<td>Dependency Analysis</td>
<td>▶ Exposes a common form of architecture violations by detecting dependencies among components in the implementation that break constraints in the intended architecture.</td>
</tr>
</tbody>
</table>

**Table 2.1:** Controlling architecture erosion with process-oriented strategies
Adoption: Only a few of the aforementioned techniques have been reasonably well adopted in the industry. In particular, architecture documentation (except for ADLs, which are not in widespread use) is included in most software processes due to its applicability across a wide range of projects. While documentation alone will not stop architecture erosion, process activities such as design and code reviews are able to identify architecture non-conformance. However, despite the availability of numerous dependency and static analysis tools, industry surveys indicate that these are not extensively used for checking architecture conformance at code level [75]. Similarly, evidence is not available to indicate the widespread use of conformance monitoring methods in industry. Some possible reasons for this general lack of adoption are highlighted in the evaluation section below.

Efficacy: These strategies are particularly effective for assuring compliance of certain non-functional attributes in architectures (e.g. adherence to standards or time-to-market). These methods are also useful in identifying early stages of architecture erosion where minor violations of architectural principles take place. This allows the implemented architecture to be kept in step with the intended architecture during initial system development. The effectiveness of most process strategies, however, is largely dependent on human factors. Process guidelines are often ignored and process checks bypassed in favour of achieving project deadlines or cutting costs. In such instances process strategies cannot be relied upon to control architecture erosion.

Evaluation: Using process methods seems a viable solution for minimising architecture erosion. Besides addressing erosion, the engineering rigour introduced by this class of techniques provides a comprehensive set of supplementary benefits such as reducing defects, increasing predictability and improving reliability of the software. A key advantage these strategies provide over others is simplicity. An organisation can use existing resources to implement most of these process techniques. However, there is an inherent resistance from developers to heavy engineering processes and process guidelines are often broken to satisfy project demands. The scaleability of process tasks with respect to architecture compliance is also a concern as core architectural knowledge is often retained within a small group of individuals. Rigorous process activity is also not cost effective during the maintenance phase of some systems when only a small team is retained. Therefore, while processes are necessary for the effective execution of software projects, they alone may not provide an adequate solution to control architecture erosion.
2.4 Architecture Evolution Management

Architecture erosion can be controlled by carefully managing the evolution of the architecture and the implementation as a whole. Often evolution management is performed with the help of software configuration management/manager (SCM) tools \[76\], also known as version control systems. A few methods have been developed by exploiting the capabilities of SCMs to control, trace and record changes made specifically to architectural models and their corresponding implementation. Systems capable of managing architecture evolution do much more than merely version architectural artefacts. They are able to semantically interpret architecture specifications. There are two classes of tools in this domain that are of interest. In the first category, the SCM plays the central role by facilitating the specification of architectures within itself. This is called SCM-centred architecture \[77\]. In more elaborate evolution management systems, the SCM becomes part of the architecture itself. This second category is termed architecture-centred SCM \[77\]. The specific techniques presented next come under one of these two categories.

Van der Hoek et al. \[78, 33\] introduce Mae, an architecture evolution environment based on architecture-centred SCM concepts. Mae uses a type system to model architectural primitives and version control is performed over type revisions and variants of a given type. Mae can also be coupled with an ADL for specifying architectures and is currently part of the ArchStudio \[79\] toolset. An important part of the architecture erosion problem that has been addressed by this tool is enabling the architecture itself to be updated in addition to managing changes to the implementation.

ArchEvol \[80\] is a SCM-centred tool that monitors the co-evolution of architecture specification and corresponding implementation. A key aspect of this approach is its use of existing tools. Although ArchEvol was implemented using ArchStudio, Eclipse \[81\] and Subversion \[82\], its conceptual model is not tied to these tools. ArchEvol allows architects and programmers to work in parallel on their respective domains of the system and takes responsibility for maintaining the consistency between architecture and program code. While this approach appears promising, its treatment of software architecture as consisting of only components and connectors might limit its usefulness in controlling erosion. Other types of architectural elements such as interfaces may also be points of erosion.

Finally, the Molhado \[83\] tool is somewhat similar to ArchEvol but can be considered an architecture-centred SCM tool with additional support for interface elements. This system combines SCM concepts such as version control with architectural principles into a unified model to help track unplanned changes to a system’s architecture or implementation.
Discussion

Techniques in this group attempt to minimise architecture erosion by managing changes to both the implementation and the architecture in parallel while ensuring they remain faithful to each other during the evolutionary process.

Adoption: There is no evidence of widespread adoption of these strategies in industry. However, a number of published academic works either extend the techniques discussed earlier (e.g. MolhadoArch [84] which expands capabilities of Molhado) or explore new avenues with the use of SCMs [85].

Efficacy: The effectiveness of these strategies in controlling architecture erosion is expected to be quite high. Their ability to monitor and control changes to an architecture specification and its implementation artefacts addresses a root cause of architecture erosion. By validating the implementation against the architecture after a change has been made to either artefact, evolution management techniques leave little room for the implementation to diverge from the intended architecture.

Evaluation: The use of evolution management techniques is not widespread despite the strong adoption of SCMs in industry [86]. A possible reason for the lack of adoption could be the advanced capabilities required by SCM tools to meaningfully monitor the evolution of architectural models. While this capability itself is not hard to implement, the myriad of languages and other methods employed to specify architectures makes evolution management practically ineffective. Furthermore, software systems using these techniques have to rely on an external entity, SCM tool, to ensure the implementation and the intended architecture do not diverge. In this arrangement the SCM tool has to be made aware of architectural rules, which creates a binding between the two, adding to the complexity of the engineering process. It is suggested that an architecture specification and its implementation should instead have a conformance checking mechanism that does not require the support of a third party.
2.5 Architecture Design Enforcement

Architecture design enforcement entails guiding the design and programming activities using formal or semi-formal architectural models. This approach, as exemplified by the model driven architecture (MDA) [87] paradigm, is another possible strategy for minimising erosion. Typically, architecture enforcement is established at the beginning of the implementation process where an architecture is transformed either directly into source code or Unified Modelling Language (UML) models that form the basis for implementation. Alternatively, the architecture of the system is elaborated using architecture patterns or frameworks that impose certain constraints on the implementation.

The following sub-categories are identified within this strategy.

- Code Generation
- Architecture Patterns
- Architecture Frameworks

These approaches are discussed in detail next.

2.5.1 Code Generation

Code generation tools, also known as generative technologies [88], are common in most integrated development environments (IDEs) available today. Generators use a specification of the architecture to produce code stubs, or skeleton classes in the case of an object-oriented language, that are representative of the architecture. However, the generated code does not produce useful services without additional work by a human programmer, even with more detailed design specifications such as UML models. Furthermore, generated code could subsequently diverge from the architectural models originally prescribed. To overcome this *round-tripping problem* [88], tools used for architecture modelling, code generation and source code editing should tightly integrate and share a protocol for communicating architecture properties with one another.

A few ADLs have associated tools to generate code from architecture specifications. The ArchStudio [89, 79] environment contains a code generator that produces template code from xADL [90] interface specifications and other architectural constructs which can thereafter be edited by a programmer. The tool has the capability to detect whether the edited code deviates from the architecture specification. Similarly, the C2 ADL toolkit [91] produces class hierarchies in Java, C++ and Ada, while Aesop [92] generates the same in only C++. The generated class hierarchies form the basis for concrete implementation [93], though unlike ArchStudio, C2 and
Aesop do not address the round tripping problem.

Stavrou and Papadopoulos [94] describe a methodology for generating executable code from architecture models represented in UML 2.0 notations. Component diagrams are suggested for modelling static structures and sequence diagrams for modelling behaviour. An accompanying tool generates Manifold code from these UML models. Manifold is a coordinating language for concurrent processes [95]. However, support for mainstream programming languages such as Java, C++ or C# is not discussed in this work.

2.5.2 Architecture Patterns

Similar to design patterns, architecture patterns [96] are well-tested and documented solutions to recurring architectural design problems. Besides the solution, architecture patterns record design decisions and their rationale along with the context in which the pattern is applicable. Patterns are also named and catalogued, allowing architecture knowledge to be communicated and shared with little ambiguity. An architecture designed incorporating patterns has two key advantages in terms of minimising architecture erosion. Firstly, patterns are robust solutions that have been tried and tested. The opportunities for a correctly applied architecture pattern to violate design principles are low, thereby reducing the possibility of erosion. Secondly, patterns communicate design decisions at a level appropriate for programmers implementing the code. This communication is vital during system evolution, as a common cause of erosion is unavailable or misunderstood design decisions.

A limitation of architecture patterns is that their rules can be easily broken, particularly when they are adapted for developing a specific system. Conflicting system requirements such as performance and maintainability often encourage programmers to make sub-optimal design decisions that disregard the constraints of an already applied architecture pattern.

Examples of common architecture patterns are Model-View-Controller (MVC) [97], Pipe and Filter [96], and the Blackboard model [98].

2.5.3 Architecture Frameworks

Architecture frameworks [99] (also known as architecture implementation frameworks [88]) consist of comprehensive collections of resources that facilitate the systematic development of software systems. Most frameworks provide some core functionality that can be either reused or extended to build applications with the support of accompanying tools, guidelines, test procedures and management processes. More advanced frameworks have ready-to-use generic services such as logging or communication infrastructure, while some others have been designed
to cater to a specific domain like e-commerce. Despite this variance, all architecture frameworks guide software engineering activities using well-defined rules. They are equipped with mechanisms to detect violations of these rules. These frameworks encompass mature architectural styles and best practices that encourage good design, enforce programming consistency and minimise programmer variability. In addition, they form the basis for architectural analysis and requirements traceability. Hence, architecture frameworks can play an important role in managing system evolution and minimising architecture erosion.

Similar to architecture patterns, architecture frameworks require significant amounts of configuration when used in real-world software projects. The generic nature of some of these frameworks leave room for misconfiguration, which can later become a source of architecture erosion.

Popular architecture frameworks such as Spring [100] and Struts [101] are called application frameworks since they offer a significant number of core services required to build industry-strength applications. Enterprise architecture frameworks consider organisational and business factors in addition to software architectures. Examples of these include the Open Group Architecture Framework [102], the US Department of Defense Architecture Framework [103] and the Zachman Framework for Enterprise Architecture [104].

2.5.4 Discussion

Key strengths of the three design enforcement strategies presented with respect to reducing architecture erosion are summarised in Table 2.2.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Contribution towards controlling erosion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Generation</td>
<td>▶ Transforms architectural design into program stubs that accurately capture properties of the intended architecture, thereby avoiding mistakes that can be made by humans when programming to an architecture.</td>
</tr>
<tr>
<td>Architecture Patterns</td>
<td>▶ Provide a robust, well-understood path for implementing the architecture while communicating key design decisions to lower level abstractions with minimum overhead, thus making programmers aware of architecture properties.</td>
</tr>
<tr>
<td>Architecture Frameworks</td>
<td>▶ Provide a comprehensive set of services to guide the implementation according to the intended architecture and may contain some capabilities to detect when the implementation diverges from the architecture.</td>
</tr>
</tbody>
</table>

Table 2.2: Controlling architecture erosion with design enforcement strategies
Adoption: The adoption of some of the techniques surveyed under this section is quite widespread in the industry, possibly due to their simplicity and applicability to mainstream software development. In particular, architecture patterns and frameworks are used extensively in commercial software development [105]. This is evident from the strong emphasis given to patterns and reference architectures by the Java and .NET developer communities [106,107]. On the contrary, code generation from architecture specifications appears poorly adopted, possibly due to ADLs and MDA not having significant take up in industry. Most IDEs are capable of generating code from intermediate design artefacts such as UML models, though these models usually do not capture a sufficient degree of architecturally meaningful information.

Efficacy: The effectiveness of design enforcement strategies in controlling architecture erosion depends largely on how precisely an architectural design can be transformed to a design of lower abstraction that can be implemented. If this model is inaccurate, then at this point the implementation has already diverged from the intended architecture. Therefore, architecture patterns and frameworks must be carefully selected to considering other system requirements during the architecting process. Similarly, code generation techniques can be effective only if the target language has sufficient constructs to support architectural abstractions.

Evaluation: Propagation and enforcement of architecture properties at source code level are fundamental in ensuring that the implementation does not deviate from the architecture. In this respect, architecture design enforcement techniques present an opportunity to address the problem of erosion effectively. However, a key practical issue faced by enforcement techniques is retaining the initial fidelity between the architecture and the implementation when either entity evolves. Of particular importance is adapting the implementation to changes made in the architecture itself, which may require undoing, for instance, some previously generated code or architecture patterns.

Another limitation of these techniques is that not all architecture properties and principles can be enforced by the current state-of-the-art. Industrial case studies show that these limitations in patterns and frameworks are intentionally exploited to satisfy competing architectural demands such as increasing performance or reducing time-to-market [16,108,109,110]. Such practices negate the contributions made by architecture styles and frameworks towards controlling erosion. Unless programming languages are able to model architectural concepts effectively, design enforcement strategies will not be completely effective in controlling erosion.
2.6 Architecture to Implementation Linkage

Significant research has been carried out to prevent architecture erosion by linking architecture specifications to the code that reifies those architectures. A key difference between linkage methods and architecture design enforcement techniques like code generation is that linkage approaches attempt to establish a continuous correlation between the architecture and the implementation. The association is often bi-directional, enabling both the architecture and the implementation to evolve independently while ensuring conformance rules are not broken in the process. Therefore, opportunities for the implementation to deviate from its architecture are effectively eliminated.

ArchJava [34], developed by Aldrich et al., unifies a formal architecture specification and its implementation in a single entity. Architectures specified using the formal notation of an ADL do not contain implementation details. ArchJava departs from this practice by allowing program code to reside within an architecture specification. An ArchJava specification can be thought of as an executable architecture due to embedded executable code. ArchJava notations allow modelling of structural and some dynamic properties of software architecture. Standard Java constructs are used within an ArchJava specification to implement the classes and methods from the architecture description. The ArchJava compiler parses the ArchJava specific syntax in the source files in order to validate architecture constraints imposed by the language. The standard Java compiler thereafter compiles the Java code embedded in the ArchJava specification.

The Archium platform [111] consists of an extension to Java, and a framework that checks runtime architecture properties. Archium was primarily designed to explicitly model architectural design decisions as part of an architecture specification and to ensure that the implementation conforms to these decisions. The Archium strategy treats software architecture as a composition of design decisions that could possibly “vaporise” if they were not adequately captured, leading to architecture erosion. Similar to ArchJava, the Archium compiler translates source specifications to Java code before invoking the Java compiler. The compiled binaries are executed together with the Archium framework that monitors the consistency of the runtime architecture of the program.

ArchWare [112, 113] provides a complete architecture-centric platform for developing evolvable software. The platform consists of an ADL for specifying executable architectures, an associated set of tools, a runtime monitoring infrastructure and a generic model for architecture-driven software processes. ArchWare aims to support active architectures, whereby the architectural model is the system implementation. This unification enables an executing system to be dynamically evolved at the architectural and implementation levels simultaneously. The
ArchWare platform is further enhanced by an architecture analysis language for validating dynamic and structural properties, and an architecture refinement language to minimise the abstraction gap between architectural models and the implementation. Altogether, the ArchWare suite accounts for a rigorous process for enacting architecture into implementation and subsequent evolution with the aim of preventing architecture erosion.

Researchers have long been investigating runtime evolution management as a possible approach for stopping architecture erosion. Early work on architecture-driven runtime software evolution was carried out by Oreizy et al. [114]. Although their approach primarily intends to address the issue of evolving executing systems that are too critical for maintenance shutdowns, the authors emphasise “controlling change to preserve system integrity”. A runtime platform called the Architecture Evolution Manager (AEM) plays the central role of propagating changes made to an architectural model to its corresponding implementation. The AEM ensures that architecture constraints are not violated in this process. This approach relies heavily on software connectors and implicit knowledge of the C2 architectural style, allowing components to be added, removed and updated without affecting the global runtime state of the system.

2.6.1 Discussion

Techniques that link architecture with implementation provide a strong mechanism to prevent architecture erosion by bonding the implementation to the architecture.

Adoption: Besides a few experimental applications [115] [116], the survey did not find evidence of widespread adoption of these techniques in the industry.

Efficacy: These technologies have the potential to be highly effective in preventing architecture erosion. The very fact that there are perpetual and explicit links between elements of the architecture and the code that realises the architecture enables both entities to evolve independently and still remain faithful to each other when the adaptation is complete. A change in the implementation that breaks an architecture rule will trigger an immediate reaction from the monitoring system. Similarly, an update to the architecture that is incompatible with the current implementation will be flagged up.

Evaluation: Unlike design enforcement techniques, methods that link architecture to implementation usually do not suffer from the round-trip problem. Runtime environments and supporting tools that accompany these approaches continuously monitor associations between the architecture and its implementation. Despite these benefits, a numbers of possibilities can be conjectured for the lack of adoption of these technologies.
An essential criterion for almost every strategy in this group is that a fairly complete architecture specification must be produced beforehand. Although upfront architectural design is a cornerstone in MDA, it conflicts with the more pragmatic iterative development techniques popular in industry. Ideally, programmers would like to begin work with a fairly accurate but incomplete architecture, while the architect continues to refine the architecture based on feedback given by them. Therefore, an architectural design process should support this iterative refinement loop, which may be difficult to reconcile with linkage strategies. These very arguments are highlighted in a case study of re-engineering a Java project into ArchJava [115].

Another weakness in techniques such as ArchJava and ArchWare is that they make it harder to differentiate the architecture from the implementation. This can cause difficulty in evolving the architecture itself. ArchJava mingles structural descriptions of the architecture with Java program code while ArchWare does not distinguish at all between architectural and implementation constructs. Separating program code from architecture specification is important for bringing any erosion control strategy into mainstream use.

The lack of adoption can also be attributed to the fact that these linkage and dynamic conformance checking approaches require the support of a runtime environment. ArchWare, Archium and AEM all require some form of infrastructure that is a permanent fixture, without which conformance checking is not possible. This will, however, add another layer of complexity to most non-trivial software applications that are already built upon middleware, application servers, persistence frameworks, etc. Both developers and users alike may resist the extra burden of integrating, testing and deploying additional runtime frameworks.

These runtime environments also inflict a significant performance penalty on target applications. While this may not be altogether avoidable, a conformance monitoring framework allowing a target application to opt-out, especially in a production environment, could encourage stronger adoption.
2.7 Self-adaptation

Self-organising or self-healing systems are capable of adapting to changing requirements and operating conditions [117] by themselves. As noted in Section 2.1, software architecture erosion is a consequence of regular system maintenance performed by humans disregarding architectural guidelines. Thus, erosion could possibly be eliminated if human intervention is replaced by a capability within the system that enables it to self-adapt with respect to complying with the architecture. Self-adaptive systems are typically based on a closed feedback loop consisting of sensors (or probes) that detect changes in the system’s environment or its internal state, comparators (or gauges) that evaluate the sensor output against pre-defined policies, and actuators that initiate a sequence of changes in the system.

Rainbow [118] is a self-adaptation framework that enables an executing system to be monitored against an architectural model. It is capable of performing limited automated repairs in the implementation if constraints in the architecture are violated. The monitoring data returned by probes in the executing system are transformed using gauges into information that can be related to the model, while the outcome from constraint evaluation is fed to a repair engine that initiates the necessary corrective measures in the live system.

Georgiadis et al. [119] propose a technique that is primarily geared towards distributed component-based systems where removing or adding components to a system can cause the system to break its architecture properties. In this approach, every component has a built-in configuration view of the architecture while a component manager updates this view based on a set of hardwired structural constraints. A component that enters or leaves the system causes other components to re-evaluate their bindings to satisfy these constraints. The system is supported by a runtime communication infrastructure allowing components to interact with one another. In a later work, Kramer and Magee [117] extend and generalise this component model to build an architecture framework for self-managed systems. They propose a tiered architectural style inspired by robotics for building autonomous software systems.

Vassev et al. [120] describe an approach for building runtime adaptive architectures by extending the Autonomic System Specification Language (ASSL) framework [121] developed by the same researchers. The framework uses the language element of ASSL to specify autonomic systems and provides a set of actions to facilitate self-modifiability of their architectures. It propagates self-initiated architectural changes to the implementation level through a runtime code generator and a code manager. It is hypothesised that this technique helps an ASSL-driven system to be hot-plugged with generated code that complies with its architecture.
2.7. SELF-ADAPTATION

Discussion
The contribution of self-adaptation strategies towards preventing architecture erosion relies on minimizing human interference in routine maintenance activities. These systems have a built-in knowledge base of the intended architecture to guide changes made to the implementation.

Adoption: These technologies have not been widely adopted in real-world systems, except in highly specialised applications such as unmanned space exploration vehicles where they become an important part of autonomous onboard systems maintenance processes [122]. There is, however, evidence of continuing academic research in the area of self-adaptation [123].

Efficacy: Self-adaptive techniques are expected to be effective in controlling erosion where regular updates to a system can be readily automated. If the majority of the changes in the target software system requires human intervention, the benefits of self-adaptation may not be significant. Furthermore, their effectiveness depends largely on the extent to which designers can anticipate beforehand of the possible changes a deployed system will go through during its lifetime.

Evaluation: Similar to linkage strategies in Section 2.6, self-adaptation technologies require an application to carry the burden of a runtime framework in order to benefit from autonomic adaptation. However, from the work that has been carried out in this area, it appears self-adaptation is more suitable for specific architectures such as distributed systems than for a broader range of applications. It also appears that a self-adaptation strategy works best with long-running software processes where system updates are performed while the software is executing and the architecture remains fairly constant during system evolution. Furthermore, as self-adaptation techniques have limited scope for changes they can perform, these methods are more applicable for correcting minor conformance violations.
2.8 Architecture Restoration

The approaches for controlling architecture erosion discussed in the previous categories are aimed at reduction and prevention. However, complete prevention is a difficult task and may not always be feasible. Industry case studies indicate that a majority of the attempts at controlling erosion are focused on repairing the damage after the fact [124]. Once erosion has been identified, the software re-engineering process involves recovering the implemented architecture, repairing the recovered architecture to conform to the intended architecture and reconciling the implementation with the repaired architecture. In addition, some repair techniques introduce an architecture discovery activity to establish a possible intended architecture from system requirements and use cases. Considering these different activities in architecture restoration, the following sections place the survey results under architecture recovery, discovery and reconciliation. In practice, however, these three techniques are used in close cooperation with one another.

2.8.1 Architecture Recovery

A large number of tools have been developed for recovering the implemented architecture using static and runtime analysis of software. Recovery is a necessary precursor to any systematic approach that restores an eroded architecture to its intended state. An early approach for architecture recovery can be found in reflexion models [58, 59]. As described in Section 2.3.3, this technique progressively refines a hypothetical architecture using information gathered by statically analysing the source code. Reflexion models can be further refined through a clustering technique [125] that groups related software entities such as classes and packages together to form a higher level abstraction, for instance a sub-system. This technique also has complementary tool support, such as the Bunch tool [126] that performs automatic clustering using dependency and call graphs generated from source code to produce graphical architecture views.

The Architecture Reconstruction Method (ARM) [127] is a semi-automatic method based on the Dali Workbench environment [128] that enables architecture recovery from source artefacts. ARM employs pattern-matching techniques to identify a given set of concrete patterns in the implementation. Detected pattern instances are used to reconstruct the implemented architecture.

Huang et al. [129] present a technique for architecture recovery by analysing the runtime state of a system. They propose using a reflective runtime infrastructure, in their case the PKUAS application server, to enable querying the runtime state of a system to determine its dynamic architecture. Although an objective of this strategy is to make it possible to modify the behaviour of an executing system by effecting architectural changes at runtime, the extent to which architectural changes can be made to an executing software system is not clear.
Abi-Antoun [130] proposes a method for recovering the runtime architecture of object-oriented systems using annotated type instances. The annotations are placed in the code by the developer to clarify design intent. These annotations provide the means for building a hierarchical object graph representing the runtime state of the system. The soundness of this technique is argued on the basis that it captures every runtime object instance and every relationship between these objects.

A process-oriented methodology for recovering architectural models and corresponding views is proposed by van Deursen et al. in their Symphony approach [131]. The recovery process focuses on building architecture views from source artefacts using any available technology. The views are constrained by viewpoints, which are defined for the problem to be addressed and reflect the source model of the system. In the second phase of this process, the source views are transformed into target architecture views using a mapping scheme. These architecture views then guide the erosion analysis and reconstruction tasks performed on the implementation.

A number of other techniques for recovering architectures are found in literature. Stringfellow et al. [28] introduce a method that uses change reports and history records obtained from revision control systems. This approach derives an architecture by “lifting” file level changes to component level and coupling this analysis with relationships among source files, in particular the \#include relationships in C/C++ source code. In addition, there have been efforts to recover useful architectural structure from program file names [132]. The researchers of this technique propose identifying the task of a source file by interpreting the meaning of its file name. They contend that this method could prove effective in recovering the decomposition views of large legacy software systems. In a method inspired by machine learning, Maqbool and Babri [35] present the use of a Naïve Bayes Classifier for architecture recovery. The classifier is trained with known elements in the target architecture and then directed to extract the complete architecture using its knowledge base.

Finally, there are a large number of reverse engineering tools that aid architecture recovery. Besides those highlighted in the process-oriented architecture compliance section, almost all modern software development tools have some form of reverse engineering capability to extract architectural structures from implementation.
2.8.2 Architecture Discovery

Architecture discovery becomes necessary when a documented architecture specification is unavailable. The discovery process builds the intended architecture from system requirements and possibly by studying system use cases. However, architecture discovery alone is not sufficient to repair erosion, and this activity often supplements architecture recovery and reconciliation. Most existing processes for discovering the intended architecture are adaptations of the reflexion model technique explained in section 2.3.3.

Medvidovic et al. [133] present a method that combines discovery, recovery and architectural styles to combat architecture erosion. In this three-step process, the intended architecture is first discovered through a process known as Component-Bus-System-Property [134] that provides a transitional model to build an architectural design from requirements. The technique uses four generic architectural primitives, namely components, connectors, configurations and properties, which can effectively capture early architectural design notions. In the second step, standard tools are used to generate class diagrams from the source code, which then go through an iterative clustering process until an architecturally significant model is developed. In the last step, the two different architectural models are integrated by identifying one or more architectural styles common to both models.

Besides the above, the survey does not identify architecture discovery methods that are particularly applicable for addressing architecture erosion. However, a large body of work exists on methods for transforming requirements into architectural design models. If the requirements are well-documented, these techniques could be used effectively to discover the intended architecture.

2.8.3 Architecture Reconciliation

Architecture reconciliation is the process by which the implementation is mended to conform to the intended architecture obtained from either a recovery or discovery activity. A common approach for reconciling an implementation with its architecture is code refactoring [135]. In this approach, the source code is systematically restructured to follow architectural design principles without altering the externally visible behaviour of the system. Similar to reverse engineering, code refactoring support is provided by a number of commercial software engineering tools [136]. However, the emphasis of these tools is generally on refactoring code to meet a specific design goal like a design pattern rather than to conform to an architecture property. Work in the area of architecture-driven refactoring is scarce and therefore could possibly become active research in future. Bourqun and Keller [137] present an approach based on an industrial case study called
“high-impact refactoring” that refactors a system at an architecture level. An architecture recovery tool and a set of informally specified architecture constraints are used to uncover architecture violations in the code. These violations are then addressed using best practices in architectural design.

Tran and Holt [23] discuss a more rigorous approach to reconcile intended and implemented architectures. They describe a two-pronged method, consisting of forward architecture repair to overhaul the implementation to match the intended architecture and reverse architecture repair to synchronise the intended architecture with the implementation. Forward repairing involves correcting invalid dependencies amongst modules while ensuring these corrections do not impact functionality. This process is partially automated using Grok [138], a language for dealing with binary relational algebra. Grok scripts describe the physical and logical dependency relations in the architecture. In reverse architecture repairing, architecture elements that do not contribute to the implementation (i.e. gratuity) are removed in a process that does not affect the source code or system functionality.

In an approach aimed at refactoring architectural models, Ivkovic and Kontiogiannis [139] use UML 2.0 profiles and semantic annotations to drive the refactoring process based on quality goals (or quality attributes). The quality goals are mapped to a graph in which quality metrics are associated to each node. This graph forms the basis for applying refactoring transformations to a UML model of the architecture. Refactoring is repeated for all the primary quality goals in the graph. The availability of tool support for this process is not clear in the literature.

2.8.4 Discussion

Table 2.3 summarises the contribution of each restoration technique towards controlling erosion.

Adoption: Despite the large number of tools and techniques currently available, architecture restoration does not appear to be broadly adopted in industry or academia. There are some documented case studies such as the architecture reconstruction of the VANISH tool using Dali Workbench [140] and the architecture recovery of the Apache web server [141]. However, these studies do not point to wider application of these methods. Although architecture recovery, in particular, is often necessary in the maintenance of mature software systems, techniques practised in industry are less rigorous than those discussed here. The usual method is to reverse engineer the source code to extract UML models that, together with available design documentation, form the basis for manually reconstructing the architecture. The derived architecture is then used as guidance for refactoring eroded parts of the implementation. Similarly, the survey does not find noticeable application of architecture discovery methods in industry.
CHAPTER 2. BACKGROUND

Strategy Contribution towards controlling erosion

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Recovery</td>
<td>▶ Establishes the necessary foundation for the detection, evaluation and repair of erosion by extracting the implemented architecture from source code and other artefacts.</td>
</tr>
<tr>
<td>Architecture Discovery</td>
<td>▶ Provides the means for building a model of the intended architecture in situations where a reliable specification is unavailable. The intended architecture is a necessary prerequisite for repairing architecture erosion.</td>
</tr>
<tr>
<td>Architecture Reconciliation</td>
<td>▶ Enables rectifying a system with an eroded architecture by aligning the implementation with the intended architecture. Reconciliation often requires the support of architecture recovery and discovery strategies.</td>
</tr>
</tbody>
</table>

**Table 2.3:** Controlling architecture erosion with architecture restoration strategies

**Efficacy:** Architecture restoration is an effective tool for controlling architecture erosion and extending the useful lifetime of software. Planned restoration work can be made part of an overall maintenance strategy. However, restoration may not always work well if the system is badly deteriorated. Therefore, for best results, restoration should be coupled with one of the preventive mechanisms presented earlier.

**Evaluation:** Although architecture restoration is an after-the-fact activity, the survey indicates that a significant number of approaches and tools developed for controlling erosion fall under this category. Therefore, it appears that architecture erosion is an inevitable consequence of the software engineering practices of today. However, repairing the implementation to comply with the intended architecture can be an expensive and tedious task in complex software systems. Furthermore, current architecture restoration techniques could recover only the structural and behavioural properties of an architecture and not architecture rationale. At best, the rationale can only be deduced from the recovered properties.
2.9 Conclusions

This chapter describes the current state-of-the-art for dealing with the problem of architecture erosion. A lightweight classification framework is presented to categorise available techniques for easier comparison and analysis. The presence of a vast number of diverse techniques, in both academic and industrial practice, indicates that software sustainability is an important problem in computer science. However, as discussed under each classification, none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

Enforcing architecture conformance through software engineering processes lacks scalability, does not resonate well with agile practices and is prone to human neglect. Managing the evolution of architectures through **SCM** tools could in theory prove very useful, but requires standardised architecture specification models and **SCM** systems that are capable of interpreting these models.

Design enforcement techniques, on the other hand, are effective in guiding the initial implementation phase to conform to the architecture. But they have the tendency to fall short during maintenance unless supported by compliance assurance checks of a rigorous software engineering process.

Approaches that link architectural models to implementation possibly make the strongest claim for constraining architecture erosion. The bi-directional associations between architecture and implementation enable these systems to continually enforce and synchronise architecture properties in an evolving system. But these techniques as well as self-adaptation technologies impose a heavy penalty on the system in the form of a runtime environment, and therefore have not been adopted for mainstream use.

Lastly, the recover-and-repair techniques are the most prevalent in practice because poorly maintained software systems eventually succumb to architecture erosion. Although restoration approaches help extend the lifetime of software, the cost of regaining the original state of the intended architecture can be economically and technically prohibitive.

Besides the existence of a gamut of technologies, fundamental causes of architecture erosion such as the inability to express architecture notions in programming languages, lack of standardised architecture description languages and insufficient software visualisation tools should be addressed in order to effectively control erosion. Finding a broad solution to architecture erosion may in fact require revisiting the basics of software engineering. Two possible strategies are identified to direct future research in this area.
• **A programming model capable of specifying and enforcing architectural principles.** Current programming languages are intended for building software design elements that are at a lower level of abstraction than software architectures. As such, these languages are not capable of expressing architecture properties within program code. A paradigm shift is required, similar to the shift that occurred from procedural programming to object-oriented programming, which places software architectures at the focal point of all programming activities.

• **Pluggable architecture environments.** A composite environment that is instantiated using an architecture specification of a system, thus encompassing all its properties, may act as an architectural “mould” within which the system evolves. It is envisaged that such an environment will monitor the structural and behavioural aspects of the implementation at development as well as execution time. However, a system being monitored in its architectural realm should also be executable without this environment. Such a strategy should encourage wider adoption as the software can be decoupled from this runtime framework if required. In this case, the task of validating runtime architecture conformance should become a regular activity in the maintenance and testing phases of the software engineering process.

Of the two, the thesis explores the second approach.
This chapter presents the conceptual foundations for building a holistic architecture conformance monitoring framework that complements existing software engineering practices while attempting to overcome the weaknesses of other available approaches. The motivations outlined in Chapter 1 are the drivers of this strategy.

3.1 Introduction

Software architecture is a key artefact that captures the most significant properties of a system and can therefore be used as the basis for stakeholder communication, reasoning about system qualities, guiding implementation, planning maintenance, etc. The essential structure, interactions and quality attributes captured at the architectural level guide the development of the system [6]. It is therefore imperative that a software system conforms to the prescribed architecture in order to deliver the required services and satisfy the required quality attributes.

As described in Chapter 2, conformance monitoring is one of the strategies used for controlling architecture erosion. While the conformance of most static properties can be checked during development, verifying the conformance of dynamic features such as interactions among components, however, is not always possible until the system has been deployed in its target operational environment. In addition, routine maintenance as well as changes to requirements and operating conditions can cause the behaviour of a deployed system to deviate from its intended architecture. Therefore, both static and dynamic conformance checking are required to ensure that a system continues to satisfy its intended architecture throughout its lifecycle. However, typical characteristics of current runtime conformance checking techniques such as modifying the application code, forming a tight coupling with the application, and causing significant performance implications have discouraged the adoption of these solutions [142].
The following sections of this chapter discuss various aspects related to architecture conformance checking and the overall strategy adopted by the architecture conformance checking framework covered in this thesis.

### 3.2 Architecture Conformance Checking

A software implementation that satisfies the properties specified in its intended architecture is considered architecturally conformant. Conformance can relate to a number of architecture properties relating to structure, interactions and quality of service (QoS) requirements. In this work we define architecture conformance and conformance checking as follows.

**Definition:** *Architecture conformance*, with respect to a software implementation, is the ability of source code and other implementation artefacts such as configuration files to satisfy both implicit and explicit properties asserted in the intended architecture of the software.

**Definition:** *Architecture conformance checking*, with respect to a software implementation, is the process of examining implementation artefacts and the runtime state of the software to derive sufficient information to determine whether the software conforms to its intended architecture.

#### 3.2.1 Static and Dynamic Conformance Checking

An architecture specification may contain multiple views of the system [20]. Some of these views are associated with static aspects of the system such as programming structures (e.g. class hierarchies), code organisation structures (e.g. packages) and deployment configurations [19]. Static conformance checks are done while a system is being built or when taken offline for maintenance. In either case, static checking involves examining the source code and other artefacts such as configuration files to extract architecturally meaningful information. This data is then used to determine any points of architecture non-conformance in the implementation. Static checking does not require the software to be executed, and therefore, the issue of performance impact on the target application does not arise. For the same reason, however, static checking is unable to verify whether an implementation conformance to the dynamic properties of its architecture.

Other architecture views express dynamic characteristics of a system [19]. Examples of these include instantiation of runtime structures and their interactions, dynamic loading and unloading of modules, live updates to the software and QoS measures. Dynamic conformance checking is performed while the system is executing and therefore requires access to runtime state and operations, from which data is continuously extracted for verifying architecture properties. As
method-level granularity is often required for this purpose, a key challenge is capturing relevant and useful data during execution while still keeping any performance impact to a minimum. However, in comparison to its static counterpart, dynamic checking provides wider conformance coverage due to its ability to analyse most static properties as well.

Most existing work in dynamic architecture conformance checking involves incorporating extra functionality into the target system, such as aspect weaving techniques employed by Merson [143] and Ganesan et al. [64], or decorating the source code with annotations representing architecture properties [144, 145]. In both cases an external monitoring system builds a runtime view of the architecture using data gathered from the added code or annotations. The reconstructed architecture is then used for checking conformance against a model of the intended architecture using other tools. Such conformance checking techniques are tightly coupled with the software and cannot be invoked only when required. This limitation can lead to problems such as permanent degradation of application performance, inflexibility in conformance checking and poor maintainability [142]. The dynamic conformance checker described in this thesis is designed to overcome these weaknesses and minimise performance impact on the software being tested.

3.2.2 Top-down vs. Bottom-up Checking

Checking for architecture conformance has two fundamental approaches. The first approach starts from the intended architecture, extracting and then breaking down properties in the architecture to fine-grained, implementation specific constraints that can be validated against information found in the implementation. This is considered a top-down process because properties at a higher abstraction are transformed to a lower abstraction for checking conformance. In the second approach, the process is reversed, whereby architecturally significant information is mined from the implementation to gradually reconstruct the architecture implemented in the code. Once a reasonably complete architecture has been extracted, this model is compared against the specification of the prescribed architecture to identify where deviations occur. We term this as bottom-up conformance checking. Both these approaches can be performed either statically or dynamically. However, the bottom-up approach can be considered an architecture recovery strategy that is extended to support conformance checking, typically with the help of manual comparison techniques [142].

3.2.3 Automated vs Manual Checking

Architecture conformance checking techniques can also be classified by the amount of human intervention required to accomplish the task. Some manual work is always required to specify the architecture used for conformance checking. Besides this groundwork, any technique that
can detect architecture non-conformance with no manual input can considered an automated conformance checking approach. However, techniques that can be classified as automated cannot be found in literature. Every documented static conformance checking method requires some manual analysis [146], while the few dynamic checking approaches follow a bottom-up strategy, necessitating manual verification of the extracted architecture [142].

3.2.4 Modelling Software Architectures

A necessary prerequisite for any architecture conformance checking approach is the availability of a specification describing the intended architecture of the software. Such a specification must be machine readable for automated checking, and also lend itself to be processed by a conformance checker. While software architectures can be described using numerous techniques (see Section 2.3.1), what is useful for conformance checking is specifications that contain sufficient architectural details for guiding the implementation. Most ADLs and other software modelling notations such as UML should be able to provide this required level of architectural knowledge for checking conformance. However, the type and breadth of conformance checking that can be performed, to a large extent, depends on the modelling capabilities of these languages [40]. For example, Acme is an ADL that follows the component-connector paradigm for describing software runtime structures with the help of other constructs such as interfaces and ports. An architecture specified using Acme should be able to feed useful information to a conformance checking tool of an objected-oriented implementation because Acme’s abstractions can be related to object-orientation. On the other hand, using Rapide, which has a strong emphasis on modelling behavioural aspects, could be challenging due the complex nature of its constructs.

Another concern related to specifying architectures is the simplicity or usability of the modelling notation. Chapter 2 argues that the specification of architectures using formal notations is not well adopted in industry. A survey by Malavolta et al. [147] reveals that most existing ADLs require a level of precision and completeness for specifying architectures that is often not possible in commercial software development. Hence, an important consideration in architecture conformance checking is ensuring that the specification medium is sufficiently flexible and easy to use. Furthermore, a conformance checking approach should be able to work with incomplete architecture specifications and cope with subsequent incremental refinements, as non-trivial software is often developed in this manner.
3.2.5 Mapping Architecture to Implementation

The software architecture of a system exists at a higher abstraction than the program code implementing the system. Although an architecture guides the implementation, traceability of architecture properties is often lost during the development process. Furthermore, an architecture may drive multiple system implementations. Therefore, to check architecture conformance an explicit implementation-specific mapping between architectural and implementation abstractions is required. Mechanisms for specifying such mappings are categorised as follows:

- **Sharing naming conventions between architectural and programming constructs.** For example, a component in the architecture is implemented by a class of the same name. Although such a mapping primarily provides structural information, it may facilitate checking conformance of dynamic and QoS properties as well. This technique can be further extended by supplementing architecture elements with annotations having information about corresponding implementation constructs. An advantage of this strategy is that it can be easily applied to mainstream software engineering tools due its simplicity. However, non-trivial software systems often have large numbers of implementation artefacts (e.g. Java classes), and maintaining name-based mappings between their architectures and implementations can amount to significant overhead for the developers.

  Alternatively, annotations in source code can be used for identifying relevant architectural entities. An annotation for a Java class, for example, can map to the architecture component implemented by that class. Source code annotations, however, are inherently unreliable because they could be easily lost as a result of programmer activity.

- **Combining architecture and implementation in a single artefact.** Architecture conformance checking in this case is minimised or not required since architecture and implementation are intrinsically linked to each other. However, existing work that follow this strategy either require a permanent runtime framework or that the application be implemented in a language not widely adopted in industry. ArchJava [34] and ArchWare [112] take this approach.

- **External (outwith both architecture and implementation) specification of the mapping.** While this mapping approach does not force architecture and implementation representations to follow naming conventions, it does require a separate artefact having mappings of all architecturally significant features. This artefact must be maintained in unison with the architecture specification and the implementation. The DiscoTect approach developed by Schmerl et al. [62] and a similar technique by Ganesan et al. [64] adopt this strategy to connect architecture with implementation.
CHAPTER 3. STRATEGY

This thesis adopts the first strategy. The rationale behind this decision is to avoid the need for a complex mapping scheme that could otherwise hinder adoption, while still allowing both static and dynamic checks to be performed. It also makes the framework more readily adaptable to existing architecture modelling schemes as well as programming languages.

3.3 Conceptual Foundation

This section discusses the conceptual underpinnings of the conformance monitoring approach covered in the thesis.

3.3.1 First Principles

The motivations presented in Chapter 1 and some of the drawbacks of existing approaches highlighted in Chapter 2 are expanded to elicit the following guiding principles.

- **Pluggable.** The target application can operate with or without the monitoring framework. When the framework is plugged in, the application does not change behaviour besides an expected slowdown in performance. When the framework is unplugged, the application executes ordinarily. This principle is relates directly to the core motivation of a pluggable conformance checker argued in Section 1.4.1.

- **Non-intrusive.** The source code of the target application is not modified. Furthermore, any instrumentation of the compiled code does not alter runtime state of the application and does not persist once execution has terminated. This principle derives from the motivation for a non-intrusive approach for checking architecture conformance discussed in Section 1.4.2.

- **Automated.** The process of deriving properties from an architecture specification and validating those properties against the implementation is automated. The motivation for this principle is the unavailability of automated conformance monitoring approaches in the literature, although automation is a strong influence for driving adoption. Existing conformance monitoring techniques, both static and dynamic, require significant amounts of manual labour. As argued in the evaluation of process-oriented architecture compliance techniques in Section 2.3.5 the poor adoption of conformance monitoring could be attributed to insufficient automation.

- **Specification-driven and notation-independent.** Architecture properties required for conformance checking are accepted only from an architecture specification of the software, and no other artefact is required. At the same time, the approach should not be bound to a
specific architecture modelling notation or medium. While a specification of an intended architecture is indeed mandatory for a top-down conformance checking approach, the use of other intermediary artefacts such as for describing implementation mappings could discourage adoption. An example is the DiscoTect approach that employs a custom notation called DiscoSTEP, which is not an ADL, for specifying mappings. Supporting notation independence could make an approach adaptable for different user requirements.

• **Incompleteness-tolerant.** The architecture specification does not have to be complete for conformance checking to be performed. Whatever properties available in the given specification are extracted and checked against the implementation. Industry surveys show that architecture definition is an on going activity along with the development process [147]. Architecture specifications are refined progressively as the system evolves. Therefore, a conformance monitoring approach should accommodate partially described architectures and still be able to function effectively.

• **Self-contained.** The framework has all the required capabilities and tools for checking architecture conformance. It does not rely upon or use an external entity. As highlighted in the background survey in Chapter 2, a number of proposed techniques for controlling architecture erosion are based on integrating software tools made for other purposes, such as SCM's and IDE's, to build a solution. Relying on external products may increase deployment and operation complexity, discouraging adoption. In this context, a self-reliant conformance monitoring approach that has minimal deployment prerequisites is considered beneficial.

• **Performance-centric.** Performance impact on the target application during runtime conformance checking is minimised as far as possible and limited to the period when the framework is plugged in. Ensuring that the subject application remains functional and testable during runtime conformance monitoring is particularly important if conformance checking is to be accepted as part of the software testing process.

• **Extensible.** The framework can accommodate input from heterogeneous sources providing static and dynamic implementation details useful for checking conformance. Existing approaches for monitoring architecture conformance focus on either static or dynamic properties in the architecture. However, a holistic approach that consider both these together could be more effective. Furthermore, software architectures that have other types of views, such as database related properties, should also be supported by a conformance checker. A framework with an extensible design, therefore, could potentially increase the effectiveness of conformance checking and its contribution towards controlling architecture erosion.
These principles form the basis for the design, development and evaluation of the architecture conformance checking framework described in subsequent chapters of this thesis. Hence as part of the overall evaluation, the framework is assessed against each of the above principles in Chapter 7.

3.3.2 The Architecture Environment

The conformance monitoring framework proposed in this thesis is inspired by the notion of a virtual architectural mould within which a software system is developed and executed. An architectural mould represents the constraints and properties in the software architecture of a system. The implementation of the system is logically constrained within the boundaries of this architectural mould and therefore made to comply with the rules of the architecture. The diagram in Figure 3.1 illustrates moulding of a hypothetical software implementation that has four components in the architecture.

![Diagram](image.png)

(a) An unconstrained software implementation.

(b) A software implementation constrained within an “architectural mould”.

Figure 3.1: The architectural moulding concept.

Figure 3.1(a) shows the unconstrained implementation in which architecture properties are not enforced. For example, if component $M_4$ is implemented as a child of component $S$ disregarding its intended position in the architecture as a child of component $M_2$, then this violation cannot be prevented. However, the same implementation encompassed within an architectural mould does not allow this architecture violation to take place. As shown in Figure 3.1(b) an implementation constrained within the mould allows minimum leeway for components to deviate from the arrangement specified in the architecture. Therefore, any attempt to relocate component $M_4$ in the implementation is not possible unless the architecture itself is changed to stipulate a different component composition.
The above notion is realised in the form of an architecture environment (AE) that forms the basis for checking architecture conformance of software systems. An AE is derived from the architecture of a system and is therefore specific to just that system. It encompasses the necessary constraints, properties and tools required for checking conformance of an implementation against its intended architecture. The AE has the ability to house the software both during development and at execution time. An important consideration, however, is that the software should also execute outwith this environment.

In this thesis, the AE is implemented as a customisable software framework capable of both statically and dynamically monitoring an application for architecture conformance. The customisation transforms the generic framework to an AE that is specific to a given software architecture. As shown in Figure 3.2, an architecture specification of the software drives the proposed AE strategy. The architecture, while guiding the software implementation, is also used for generating the customisable aspects of the AE. A generated AE, therefore, has the required architectural knowledge of the software that it will monitor. While the framework does not require a specific notation for describing architectures, an architecture description language (ADL) capable of describing useful structural and behavioural properties is particularly suitable for checking conformance.

![Architecture specification driving both the AE and system implementation.](image)

Once created, the AE is used in two possible modes for checking architecture conformance. In the first mode the AE is deployed to statically monitor architecture conformance of the software while it goes through phases of development. In this configuration, the source code is analysed either periodically or on-demand to uncover architecture violations. A typical use case is to make the AE carry out these checks as part of a build process or continuous integration activity. The diagram in Figure 3.3(a) illustrates this mode.
CHAPTER 3. STRATEGY

In the second scenario the AE performs dynamic conformance checking as the software executes. The software is launched from within the AE, thereby allowing the AE to continuously monitor the runtime status of the executing system for violations of architecture properties. The diagram in Figure 3.3(b) depicts the AE mode for runtime architecture conformance checking while Figure 3.3(c) shows the system executing unencumbered with no attached AE.

The AE strategy provides the conceptual foundation for the design and implementation of an architecture conformance checking framework that meets the core principles described earlier.

3.3.3 Architecture Conformance Rules

A cornerstone of a top-down conformance verification strategy is transforming the essential properties of an architecture to a medium that allows these properties to be effectively checked against programming constructs in the implementation. In this regard, the thesis introduces the notion of architecture conformance rules. An architecture conformance rule can be described as a simple implementation specific constraint derived from an architecture property and its corresponding implementation mapping, and one that should hold whenever the implementation conforms to the architecture. More formally, for a given architecture property $p$ with an associated implementation mapping $M_p$, if $CR_p$ is a conformance rule derived from $p$, then,

$$CR_p = c(p, M_p)$$

where $c$ is the constraint function that yields true whenever the implementation conforms to $p$.

As a non-trivial software architecture contains a number of properties of which implementation conformance must be verified, the conformance checker accepts a set of conformance rules derived from these properties. Thus, from the viewpoint of the conformance checker, an
3.3. CONCEPTUAL FOUNDATION

architecture takes the form of \( \{ CR_1, CR_2, CR_3, \ldots, CR_n \} \) for some \( n \in \mathbb{N} \). If the constraint functions of one or more conformance rules do not evaluate to \textit{true}, the implementation is deemed to be non-compliant with the prescribed architecture. However, the detection of architecture violations is dependent on the coverage of conformance rules provided and the level of detail of their implementation mappings. In the event some properties in the architecture are not represented by conformance rules or some conformance rules have mappings that are insufficient, architecture violations in these areas of the implementation may not be detected.

A conformance rule is not necessarily unique to the architecture property from which it is derived. If two architecture properties, \( p \) and \( p' \), produce conformance rules \( CR_p \) and \( CR_{p'} \) respectively, a possibility exists where \( CR_p = CR_{p'} \), provided their mappings refer to the same type of implementation. In a similar manner, two dissimilar architectures specifications have the possibility of generating identical sets of conformance rules, again, given both architectures are implemented using the same technology. Conversely, two different implementations of the same architecture are very likely to yield dissimilar sets of conformance rules.

Although a group of conformance rules together may represent, to a certain degree, the architecture from which those rules are derived, they may not serve useful for reverse engineering the architecture. Some architectural abstractions are lost during the generation of conformance rules as these rules are, by definition, implementation specific. Furthermore, conformance rules are specifically designed for comparing properties in the architectures with a corresponding implementation. Therefore, these are not suitable for comparing two architecture models, for instance to check the degree of conformance of an evolved architecture \( A' \) with its original architecture \( A \). Formal equivalence checking of two such models requires a more rigorous approach, possibly an adaptation of a method used in the design of digital circuits [148].

Conformance rules are fine-grained and specific to the technology or the language in which the target software is implemented. It is important that a conformance rule can be easily validated using information available in the implementation, without the need for additional mappings.

A single architecture property can be represented by a single conformance rule or broken down to multiple rules. Depending on the requirements of the property, multiple conformance rules derived from a single property are evaluated as group, or individually. As an example, the simple architecture declaration “component \( A \) provides interface \( IX \)” is translated to a conformance rule that verifies whether programming elements implementing component \( A \) has the interface \( IX \). In the case the implementation is in Java, the conformance rule checks whether the Java classes implementing component \( A \) or their superclasses has implemented a Java named interface \( IX \). If a single Java class has implemented the required interface in full, then the corresponding rule
deems the architecture property to have been satisfied. Alternatively, multiple classes (or their superclasses) together can implement interface IX, provided that the programming language supports this capability.

Another example is the architecture declaration “component A resides in layer L”. Verifying conformance of this property requires a conformance rule that checks whether a class mapped to component A is contained within the namespace associated with layer L. For the property to hold, every class that implements component A must belong to the stipulated namespace.

The conformance rules in the above examples can be formally specified as follows.

Let $C$ be a component in the architecture, $\mathcal{C}$ be a finite set of classes $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ that implements $C$ for some $n \in \mathbb{N}$, $P$ be a provide interface, $ip$ be an interface that implements $P$ and $ns$ be a namespace that implements layer $L$.

1. **Conformance rule for a ‘provides’ interface in the architecture:**
   For a given component $C$ that provides interface $P$, the following holds.
   
   $$\exists a \in \mathcal{C} \text{ implementsInterface}(a, ip)$$
   
   where predicate $\text{implementsInterface}(x, y)$ is true when class $x$ or one of its superclasses implements interface $y$.

2. **Conformance rule for a component residing in a layer in the architecture:**
   For a given component $C$ in residing in layer $L$, the following holds.
   
   $$\forall a \in \mathcal{C} \text{ inNamespace}(a, ns)$$
   
   where, predicate $\text{inNamespace}(x, y)$ is true when class $x$ is declared in namespace $y$.

An architecture may contain both implicit and explicit constraints. Implicit constraints are those that are described by the semantics of each individual architecture element in the ADL. An example is an architecture layer. If some component $A$ is placed inside layer $L$, then the implicit constraint derived from this organisation is “component $A$ resides within layer $L$ and it will not communicate with components residing in layers above $L$”.

Explicit constraints are useful for enforcing architecture rules that are specific to a given software system. They can be described as architecture properties or through some other means made available by the ADL. Similar to implicit constraints, conformance rules can be derived from explicit constraints as well, but their interpretation depends to a great degree on the constructs
provided by the ADL and the technology in which the system is implemented. For instance an explicit constraint enforcing a QoS measure can be translated to a conformance rule that checks, at runtime, the value of a global variable representative of that particular measure.

The concept of conformance rules forms the basis implementing the architecture conformance checking approach presented in this thesis.

### 3.4 Proposed Approach

In this thesis, the conceptual AE strategy is realised with the development of an architecture conformance checking framework called PANDArch \[149\]. Its design and implementation are guided by the core principles discussed earlier in this chapter. Therefore, PANDArch aspires to be pluggable, non-intrusive, automated, specification-driven and extensible. The following sections present the various aspects of the high-level approach taken by the framework for checking architecture conformance.

#### 3.4.1 Conformance Monitoring Process

The process for checking conformance using PANDArch is illustrated in Figure 3.4. This process is technology agnostic and therefore not tied to the specific implementation discussed later in this thesis.

![Figure 3.4: Process for checking architecture conformance in the proposed framework.](image)

The entry point to the process is an architecture specification that drives both system development and the conformance checking process. While this architecture guides the detailed design and subsequent programming activity that builds the software, it also provides the necessary input
for checking conformance. Architecture constraints and mapping information are extracted from the specification through a compilation process and used by the rules generator to create a set of conformance rules. In this approach the mapping scheme is part of the same architecture specification and customisable to suit the specific technology or language used for implementing the target software. For instance, if the system is implemented in Java, then the mapping scheme is specific to Java and object-orientation.

The conformance checker takes the set of conformance rules and validates them against runtime events while the program executes. It may produce a series of violation notifications where appropriate. This is a continuous process while the framework is plugged in and the application is executing. If required, execution data is cached by the framework until there is sufficient information to validate an architecture rule. The same process is applicable to events generated through static analysis. Although not covered in this thesis, the design sufficiently extensible to accommodate a static analyser using an adaptor to transform code inspection notifications to generic events understood by the framework.

### 3.4.2 Specifying Architectures for Conformance Checking

The PANDArch approach does not rely on a specific medium (i.e. an ADL, UML or some other notation) for acquiring architecture descriptions. It does, however, expect the chosen medium to be suitable for modelling the architecture of the subject software and express architectural concepts required for conformance checking. For instance, if the framework is required to perform runtime conformance checking, then the ADL should have the ability to model runtime architecture properties. Similarly, the ADL should support modelling logical and physical architecture views with static properties if static conformance checking is required.

An ADL or any architecture modelling notation to be used with PANDArch should also offer a facility for tagging elements in the architecture with additional information without altering the semantics of those elements. This capability is required to supplement elements in the architecture with their corresponding links to the implementation. Annotations or comment tags typically provide this feature in modelling languages, though other mechanisms may also be available. In case the chosen ADL does not support an annotation capability, a supplementary mapping scheme with its own notation is required to associate architecture elements to their corresponding constructs in the implementation.

Finally, the modelling notation of choice should have a mechanism or provide tool support for PANDArch to programatically extract architecture properties and mapping information required for conformance checking.
3.4.3 Linking Architecture to Implementation

An important aspect of the PANDArch approach is combining the architecture specification with information that links various architecture elements to their corresponding program constructs in the implementation. This mapping information is added only to the architecture specification, while implementation artefacts remain untouched.

As stated earlier, PANDArch uses naming conventions to establish associations the architecture and the implementation. In some cases names used in the architecture may refer directly to the corresponding construct in the implementation. In other cases, however, name of an architecture element may need to be expanded and/or transformed to a programming language specific name. Also, a single architecture element may correlate to a number of items in the implementation or even have to be mapped to a programming construct with a different name. PANDArch caters to all these scenarios by tagging elements in the architecture with explicit implementation specific mappings. These mappings take the form of key-value pairs. The keys belong to a vocabulary defined by the conformance checker with precise semantics. Additionally, the tags may also contain switches or directives requesting the conformance checker to alter its behaviour.

Figure 3.5 shows the tagging of architecture elements with implementation mappings.

![Diagram showing tagging of architecture elements with implementation mappings]

Figure 3.5: Tagging architecture elements with mappings to implementation.

A crucial consideration of the above mapping approach, as noted in section 3.4.2, is that the ADL used for modelling the architecture should provide the means for accommodating the tags as part of the specification. If no such facility is available, an external mapping scheme is needed.
3.4.4 Mining Implementation Data

PANDArch takes an event-oriented approach for collecting implementation details necessary for conformance checking. This means that the conformance checker waits until some event occurs in the implementation to collect some details for evaluating conformance rules. An event can be loosely described as some interesting occurrence taking place while the application is running or being analysed by an external tool. The reasons for taking this approach is to ensure that the framework remains non-intrusive and pluggable, two of its guiding principles, when checking conformance. The non-intrusiveness principle is applied in respect to the source code of the software being monitored, hence the framework refrains from modifying source files for the purpose of extracting implementation details. However, PANDArch is allowed to instrument the compiled code, if required, during or prior to execution in manner that is transparent to the application and does not leave behind any instrumentation residue after the application has terminated. Likewise, listening to events coming from the target application instead of embedding permanent monitoring code allows the framework to be easily unplugged from the application without having the need to recompile the application source code. Furthermore, events facilitate dynamic architecture conformance checking as certain runtime events, such as thread creation, are hard to detect even with intrusive source code modification techniques. Events could also be applied for checking conformance of distributed systems. An event listener capable of interpreting messages exchanged between the nodes of a distributed application could be deployed for generating application specific events, from which properties of the distributed architecture could checked for conformance.

Not all forms implementation metadata extraction techniques lend themselves well towards an event-driven model. In particular, static code analysis is a proactive task performed by a tool on dormant source or compiled code. Nevertheless, in such cases PANDArch may launch an external tool to perform the analysis work, and with the help of an adaptor, translate the results of the analysis into a series of events understood by the core framework. In either case, implementation or technology specific events are transformed into platform independent generic events to make processing easier for the conformance checking components of the framework.

The diagram in Figure 3.6 further illustrates the event-driven process used for gathering implementation data. In this example, the conformance checker is fed with events arriving from three distinct sources. These event streams can occur either simultaneously or otherwise, and the framework is capable of dealing with both situations. Once an event has been received through an event adaptor and transformed to a generic event, the framework uses the implementation metadata contained in the event to evaluate one or more conformance rules.
3.4. PROPOSED APPROACH

3.4.5 Using the Framework in the Software Development Life Cycle

The proposed approach for checking software architecture conformance can be readily adopted within most software development processes used in practice. The architecture of the software under consideration is specified during the design phase. This architecture specification, while guiding the implementation, also forms the basis for checking conformance. For the purpose of conformance checking, the architecture is supplemented with implementation mappings and these are refined and altered as the development of the software progresses. Furthermore, as the PANDArch framework has tolerance for incomplete architecture specifications, static conformance can be checked regularly during development. The specification and the implementation mappings are continually adapted, and further elaborated if required, allowing the architecture and the implementation to evolve in parallel.

When the development process reaches the verification phase, developers may choose to perform dynamic conformance in conjunction with the other software test procedures. In this case, test cases designed for validating functional aspects of the software are executed with the conformance checker plugged-in to the software. Any architecture violations detected during the execution of these test cases are reported by PANDArch next to the outcomes of the tests. The conformance checker could also be deployed in controlled production environments, such during beta testing, if the performance impact is not a concern. In either case, PANDArch can be easily unplugged whenever the native performance of the software needs to be evaluated.

Figure 3.6: The event-driven process used for gathering implementation data.
3.5 Conclusions

Software architecture is recognised as an important artefact in the software lifecycle. In order to reap the benefits of such a high level holistic model, it is essential that software systems conform to their intended architectures despite pressures of change. Architecture conformance checking is thus an essential process during development and maintenance of a system and it is important that this process is made as viable, practical and cost-effective as possible. To this end, the chapter presents a strategy that opens up the possibility of developing an automated, non-intrusive, pluggable, self-contained and extensible conformance checking framework. A notion of an architecture environment capable of guiding the application to conform to its architecture both during development and execution serves as the conceptual foundation for this strategy. Furthermore, a number of core principles are introduced to avoid pitfalls of existing approaches. These principles and the conceptual foundation is adopted for building a concrete conformance checking framework called PANDArch. The design and implementation of this framework are described in the following two chapters.
The general architecture and core design aspects of the PANDArch framework are presented in this chapter. The design process starts with the formulation of design decisions that support the primary requirements of the framework. The guiding principles presented in the previous chapter act as the de-facto requirements specification, from which a number of design decisions are derived. These design decisions are used to devise a technology independent, generic architecture in the first instance, which is thereafter refined into a Java specific architecture and a detailed design that drives the implementation. The Grasp ADL is adopted for specifying architectures and implementation mappings of applications that are to be checked for conformance using PANDArch. The chapter concludes with a discussion that evaluates the framework design against the guiding principles.

4.1 Introduction

The architecture and the detailed design of the PANDArch framework are crucial for ensuring that the framework satisfies the core principles presented in the previous chapter. While the event-driven approach lays the foundation for making PANDArch non-intrusive in terms of modifying source artefacts of the target application, a component-oriented architecture is imperative for developing an extensible framework. This extensibility enables the framework to accommodate different event sources and specification parsers through well-defined interfaces. It also allows the framework to be incrementally built and evaluated, adding new modules only when those that already exist have been well tested.
The architecture also plays an important role in meeting the performance goals of the framework. In particular, a modular architecture often has performance implications due to the extra levels of abstraction and layers of delegation introduced to achieve modularity. Hence the architecture of PANDArch pays close attention to minimising the impact on the execution performance of an application while retaining the modularity and extensibility of the framework. In this regard, a number of design rules are prescribed in the general architecture of PANDArch irrespective of the technology used for implementing the framework. Generic event containers are designed to ensure that they convey only the necessary information needed for evaluating conformance rules. Additionally, memory caching is used for avoid recopying information already received from an event source while thread pools are utilised for processing events in parallel.

4.2 Architectural Design Decisions

The architecture of the conformance checking framework is primarily driven by the guiding principles and the high-level approach presented in Chapter 3. These motivate the following key architectural decisions taken at the onset of the design process.

1. **Use an event-driven strategy for collecting implementation metadata.** In accordance with the conceptual strategy discussed in Section 3.4.4, the framework utilises an event-driven mechanism for extracting implementation details required for architecture conformance checking. PANDArch defines a set of well-known events and their properties that carry information relevant to each event.

An event is triggered in response to an occurrence of interest in the software, and these events can be either dynamic or static. Dynamic events are raised when the application is executing. Static events, however, originate from a process scanning the application source code. The framework defines a number of different types of events. Some of these are purely dynamic events while certain others are shared by both dynamic and static processes. For example, an exception-raised event is generated whenever an software exception is thrown in the executing code. This is a dynamic event that does not occur when statically analysing the code. On the contrary, the method-entry event can originate from both a running program and a tool analysing its code.

An event from a native source consists of platform specific properties such as the event type, details about the source and other implementation metadata relevant to that event. The method-entry event, for instance, has the names of both the calling and called methods along with their respective class names. Platform specific events are referred to as native events in this thesis.
Native events, both dynamic and static, originate from platform specific event sources. Events for an executing program, for example, can come from the VM hosting the program or instrumentation placed in the code. The architecture of PANDArch isolates the core components of the framework responsible for conformance checking from these platform dependencies. As part of this strategy, native events are transformed into generic events understood by these core components. Similarly, implementation metadata contained in native events are converted into a generic form called implementation descriptors. The architecture includes a specification of the implementation descriptors it supports.

This design decision supports the **pluggability** and **non-intrusiveness** principles and also contributes towards satisfying the **automation** principle.

2. **Make events and implementation metadata platform independent.** The conformance checking elements of the framework make use of platform independent event objects to carry out their work. Additionally, the various items of metadata in these event objects are also specified in a technology agnostic manner. For example, the fully-qualified class name is denoted using a generic notation irrespective of the underlying technology or the programming language. Event adaptors make this transformation possible (see next item).

This design decision supports the **extensibility** principle.

3. **Adopt a two-stage processing pipeline.** The PANDArch architecture consists of two types fundamental components, each performing a distinct task in the conformance checking pipeline. The first type is an event monitor component, while the other is an evaluator component. An event monitor is a pluggable adaptor that is responsible for receiving and interpreting events arriving from the target application. Different classes of events are processed by different monitor components. For instance, a runtime monitor listens to an application’s runtime events while a static monitor receives events emitted by a static code analyser. Alternatively, multiple runtime monitors may simultaneously handle multiple execution processes of the same application. The framework may also include monitors capable of reading events of software developed in a technology different from that of the framework. An example is a monitor component implemented in Java handling the runtime events emitted from .NET applications. In all cases a monitor component has the necessary knowledge to connect to a compatible event source and interpret its generated events.

An important function of monitor components is translating platform-specific native events into generic events understood by the rest of the framework. A native event typically maps directly to a generic event, though a monitor can generate multiple generic events from a single native event or collate multiple native events into a single generic event.
Every event monitor ensures that the generic events they emit are correctly ordered, if required, to avoid encountering unresolved references. For example, a method-exit event must occur only after its corresponding method-entry event. Similarly, a class-load event for a class occurs only after the load events for all its superclasses and implemented interfaces have been generated.

The second type of component, an evaluator, listens to generic events produced by a monitor component and performs the necessary work to evaluate architecture conformance rules after extracting implementation data from these events. An evaluator component attaches to one or more matching monitor components as they are expected to work in tandem. For example, a runtime evaluator may bind to a runtime monitor for carrying out architecture conformance checks on an executing software system. An evaluator also has the option of connecting with multiple monitors if required, while the framework has the facility to invoke multiple evaluators simultaneously. Prior to starting a conformance checking session, all participating evaluator components are configured with appropriate conformance rules and bound to their respective event monitors.

An illustrated view of the two-stage event processing pipeline shown in Figure 4.1 represents the general architecture of the PANDArch framework. The arrows in this diagram indicate the flow of information. The general architecture is platform independent and therefore serves as the foundation for a technology specific architecture suitable for guiding an implementation. Furthermore, the view in Figure 4.1 depicts one possible instance of the general architecture. The architecture, in this example, consists of three runtime monitors for listening to events taking place during program execution, one static monitor for receiving events generated by a static analysis tool and one distributed monitor for gathering application specific events arriving from distributed elements of the application. The outputs of these monitor components are pipelined to matching evaluator components where the information contained in generic events is used for evaluating conformance.

While evaluators are the primary consumers of the data made available by event monitors, the open architecture of the framework enables additional components such as visualisation tools to be built to consume these event streams.

This design decision supports the extensibility principle.
Figure 4.1: The general architecture of PANDArch showing the two-stage conformance evaluation pipeline. Three evaluators are used for checking conformance in the target application with the runtime evaluator, in this case, configured with three runtime monitors to receive events arriving from three separate processes of the executing application. In addition to the monitors and evaluators, the architecture also includes two distiller components for extracting properties from the architecture specification of the target application. The extracted conformance rules are fed to all three evaluators, although for improved clarity, the diagram depicts only the runtime evaluator accepting information from the distillers.
4. **Make the framework independent of any specific architecture notation.** The PANDArch framework is not tied to a specific ADL. Instead its architecture allows the framework to be extended with custom modules capable of generating standard conformance rules from suitable architecture specifications. Known as *distiller* components, the framework may support any number of such modules, each capable of processing a different ADL. These modules are not only specification parsers, but also providers of conformance rules. The framework architecture describes the interfaces that all conformance rule providers must implement. It also facilitates the automatic generation of conformance rules according to a published specification, allowing providers to easily translate properties and mappings in architectures to a medium useful for checking conformance. When PANDArch is invoked with an architecture specification as input, it examines the properties of the file containing the specification to determine the correct module to use for parsing and extracting conformance rules from the given architecture.

The framework architecture in Figure 4.1 includes two distiller components, one each for extracting architecture properties from Grasp specifications and UML models of the application architecture.

This design decision supports the principle of *notation-independence*.

5. **Implement conformance rules as data structures.** In PANDArch, conformance rules generated from an architecture specification are represented as data structures (i.e. objects) in the technology used for implementing the framework. These objects are defined by the PANDArch architecture and bound to evaluator components. While some objects are shared between different evaluators, others could be more specific to one evaluator. For example, the object representing the conformance rule EvalProvidesInterface can be used by a static evaluator as well as a runtime evaluator because both are capable of evaluating whether some component implementation has the *provides* interface specified in its architecture. On the other hand, the EvalRaisesException rule can be evaluated by only a runtime evaluator, because exceptions are runtime only events. In either case, every conformance rule object is bound to a specific type of event. Multiple rules can be associated with a single event type, and each one of those rules is evaluated whenever an event of that type is received.

Conformance rules are customised by providing them with the required implementation mappings as construction arguments. Every rule object implements an interface that has two methods named `evaluate()` and `conclude()`. The `evaluate()` method is invoked whenever an event to which the rule has been bound is received. This method can either evaluate the rule immediately, in which case conformance checking is done in “realtime”,
or save the event data for later until sufficient information is available for evaluation. On the other hand, the evaluator invokes the `conclude()` method when the event stream has stopped, for example when the target application has finished executing. This allows every rule object to carry out post-processing, required in certain instances where a conformance rule can be evaluated only at the end of the process. For example, checking whether some interaction between two components has occurred at runtime requires the evaluator to wait until the program has terminated to provide a definitive answer.

A bespoke notation for specifying conformance rules is decided against, primarily because these rules are not hand-crafted but instead generated using information extracted from an architecture specification.

This design decision contributes towards satisfying the **automation** principle.

6. **Component-oriented architecture.** The architecture of the PANDArch framework follows a component-oriented approach. All top-level functional elements are structured as components. Services provided by these components are accessible only through one or more published interfaces.

This design decision supports the **extensibility** principle.

The above architectural decisions are reified in the Java specific PANDArch architecture and related design aspects discussed next, and in the Java implementation of the framework presented in Chapter 5.

### 4.3 The Java Specific Architecture of the Framework

The PANDArch framework is currently implemented in Java and supports monitoring conformance of Java applications executing in a single instance of the Java VM. Therefore, certain aspects of the design are dependent on the Java infrastructure, and may require redesigning if the need arises to implemented the framework in another technology. However, portability of the framework is not considered to be as important as the ability to check conformance of software developed on non-Java platforms. The latter capability has been accommodated with the design decision to use event monitors to isolate native event sources from the rest of the framework. Accordingly, the Java specific design and implementation is contained within a monitor component that monitors Java runtime events. Besides this component, other elements in the design are portable to another object-oriented platform such as the .NET Framework. Where applicable, Java specific dependencies are noted in the discussions.
The following sections of this chapter focus on the significant technical areas of the PANDArch design. UML diagrams are used wherever applicable to illustrate important parts of the design.

### 4.3.1 Logical Layers

The layered view of the PANDArch architecture used for its Java implementation is shown in Figure 4.2. This architecture is a Java specific adaptation of the general PANDArch architecture presented earlier in Figure 4.1. The two bottom layers in the diagram are related to the Java technology stack, though the target application and framework execute in separate JVM instances. The following subsections explain the other layers.

![Figure 4.2: The layered architecture used in the Java implementation of the PANDArch framework.](image-url)
4.3.1.1 Thread Pooling, Caching and Event Queuing Layer

This layer which sits immediately above the two Java infrastructure layers, contains a collection of components that offer utility services to components in the higher layers. Most of these services are related to event queueing, caching of implementation metadata objects, and event dispatching through thread pools. They are fairly unspecific in nature, allowing components dealing with both native and generic events to make use of these services. This layer also contains components that provide utility services such as logging.

4.3.1.2 Event Monitoring and Genericising Layer

This layer contains components that monitor native events. Besides the different event monitor implementations, the layer also includes definitions of the public interfaces exposed by these monitors, some reusable base implementations, and components for translating native events to generic events. The monitor components provide components in higher layers the means to subscribe as event listeners. These listeners are fed with generic events types recognised by the framework. Generic events are created from their corresponding native events, though a one-to-one mapping is not always required and each monitor implementation is free to perform the most appropriate translation. Monitor components may also filter out events that are not useful in checking conformance. For instance, events related to private, protected or synthetic method invocations are ignored because they have not architectural significance. Both private and protected methods invocations occur only within a class, while synthetic methods are compiler generated and therefore do not appear in the source code. As an additional performance improvement, this layer caches properties of native events. The property that describes the class in a class-load event, for example, is cached to allow subsequent references to that class to be retrieved from the cache instead from the native event.
The genericised events are then sent to components in higher layers that have registered as event listeners. As another performance improvement strategy, events are placed in a queue and then dispatched using different threads than those on which events are received. This ensures that an event receiver thread is not blocked while an event is being handled by a consumer component, in particular, an evaluator. Blocking receiver threads for long periods could in some platforms impede the execution performance of an application.

The framework currently implements a Java runtime monitor component within this layer. This component employs the Java Debug Interface (JDI) [150] for receiving events from executing Java code. The JDI is primarily intended for developers building debugging tools for Java software. However, because it has an event model that aligns well with PANDArch and provides access to a large number Java code properties, the JDI is a useful platform for checking conformance of running Java applications. The JDI is also a high-level API that simplifies tool development by abstracting JVM specific details from the developer. The Java technology stack offers other low-level programming interfaces to extract execution events and properties from a JVM instance. However, the JDI is chosen due to its simplicity and pure Java implementation.

A weakness of the JDI, however, is the large performance impact on the application being monitored. Measurements taken from early prototypes revealed that the JDI causes approximately a 4 times slowdown in the trivial Java application used for testing. In order to reduce performance degradation, the JDI is complemented by runtime instrumentation probes. These probes are injected into the byte code streams of the Java application classes when they are loaded by the JVM. They are optimised for performance and designed as a replacement for the JDI for generating three runtime events, namely interface-load, class-load and method-entry, but with a reduced impact on the application. As these probes are both additive and stateless, they do not alter application behaviour. Furthermore, the byte code of Java classes are modified in-memory, therefore changes are not persistent.

A detailed description of the use of the JDI and instrumentation probes to monitor an executing Java application is given in Chapter 5.

4.3.1.3 Conformance Evaluation Layer

Evaluator components that perform conformance checking reside in this layer. Similar to the layer containing monitors, this layer has the specifications of public interfaces implemented by every evaluator component, some base implementation shared by all evaluators and the fully implemented evaluators. In addition, a factory for creating conformance rules is housed here. Every evaluator in this layer has one or more compatible monitors in the layer beneath.
4.3. THE JAVA SPECIFIC ARCHITECTURE OF THE FRAMEWORK

During normal operation, an evaluator component is first given a set of conformance rules that it should evaluate using the implementation metadata it receives through events. The evaluator thereafter invokes one or more matching monitors, providing any required arguments to connect with the event source. An event source is a tool or an interface that generates native events on behalf of the application. The event source for a runtime monitor is often a VM, and for a static monitor the most likely source a tool scanning the code. The evaluator also registers with the monitors to listen to events it needs to process conformance rules. From this point onwards the monitors continuously feed the evaluator with generic events corresponding to the native events arriving from the event sources. This process continues until the source stops sending events.

To complement the Java runtime monitor noted earlier, the current PANDArch implementation has a runtime evaluator component. This runtime evaluator is meant to be used with the Java runtime monitor and this is the case when PANDArch is evaluated as part of this work. However, because the evaluator design relies purely on generic events and implementation metadata descriptors, the implemented runtime evaluator can support applications executing on other platforms with the help of an appropriate monitor.

4.3.1.4 Conformance Rule Generation Layer

This layer includes components that parse architecture specifications and generate conformance rules. This layer also contains the public interfaces of those components and some shared implementation. The function of a rules generator is to parse an architecture specification and translate the properties and implementation mappings in the architecture into a collection of conformance rules. These rules are thereafter given to an appropriate evaluator component to begin the conformance checking process.

In the current framework implementation the layer consists of one component for generating conformance rules. This component parses textual architectures described in the Grasp notation. A detailed discussion about employing Grasp for checking conformance with PANDArch is given in Chapter 5 under Section 5.6.1.

4.3.1.5 Application and User Interface Layer

As the topmost layer in the architecture, this contains components that provide the front-end services to PANDArch users. These include the command interface to launch the conformance checker and associated error handlers. The launcher component is responsible for invoking the correct rules generator for the given architecture specification and also configuring the necessary evaluators.
4.3.2 Component Decomposition

As discussed in the previous section, each logical layer in the architecture houses a number of components that carry out the functional aspects related to that layer. A decomposed view of the layers (except for the utility layer and the two Java specific layers) showing important components and their interactions is given in Figure 4.3.

As shown, the Event Monitoring layer has a factory for creating instances of monitor components. This factory enables consumers in higher layers to easily select the type of monitor required, although only a Java runtime monitor is currently implemented. This layer also has components that represent generic events and implementation descriptors. Note that these two components in the diagram are place holders for different types of event and descriptor components available in this layer. Furthermore, the Java runtime monitor contains a number of other components, in particular those that process JDI events and messages sent from Java probes. These are discussed in detail along with other aspects of the framework implementation in Chapter 5.

The components in the Rules Evaluation layer are somewhat similar to those in the monitoring layer. The evaluator factory is responsible for creating instances of various evaluators though only a runtime evaluator is implemented as part for this thesis. A second factory component in this layer provides for creating conformance rule objects. Again, there are a number of different conformance rule types represented by different components, and these are all denoted by a single place holder in the diagram. Both factory components and the components they produce are available for consumption by the higher layers.

The factory in the Rules Generation layer makes instances of components that generate conformance rules from architecture specifications. The current implementation has a rules generator for processing Grasp specifications. Both are used by the components in the topmost layer, which are responsible for accepting user input, launching the framework as well as the subject application, and presenting conformance violations and errors reported by other components in the framework.

As noted in Figure 4.3 and the accompanying discussion above, the PANDArch framework implemented for this thesis consists of a limited number of components. However, the extensible nature of the design allows different types of monitors, evaluators, rules generators and other supporting components to coexist. A summary of currently implemented components in each layer of the PANDArch architecture and components that could be implemented in possible future expansion of the framework is given in Table 4.1.
4.3. THE JAVA SPECIFIC ARCHITECTURE OF THE FRAMEWORK

Figure 4.3: The component decomposition view of the PANDArch architecture. This diagrams shows the main components available in the current implementation of the framework and their interactions.
### 4.3.3 Physical Deployment

The PANDArch framework is designed to be a stand-alone Java application. In order to simplify use, the design also strives to make the framework deployable as a single physical unit as much as possible. In Java this translates to a single executable Java archive (JAR) file. Although the extensible, component-oriented design allows new components (such as new a monitor or evaluator) to be easily introduced, they are not hot-pluggable. When a new component is introduced to the framework, it needs to be linked together with the rest of the framework and a new Java deployment package created. This decision ensures simplicity while a capability to introduce components without rebuilding the framework does not improve its effectiveness or usability.

The diagram in Figure 4.4 illustrates the deployment architecture of the framework.

The only module that is part of PANDArch but physically separate from the rest of the framework is the JAR package containing the Java instrumentation probes. The design uses the Java agent technology [151] to inject instrumentation code to the Java application being monitored. The Java specification requires an agent archive file to conform to a certain structure. This archive file is specified as a command line argument to the JVM when the target application is invoked. The probes communicate with the main framework through an asynchronous socket channel.

A JAR package containing the Grasp compiler and its API is used by the framework for reading architectures specified in Grasp. This is the only external dependency the framework requires.

---

<table>
<thead>
<tr>
<th>Architectural Layer</th>
<th>Implemented Components</th>
<th>Future Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformance Rule Generation</td>
<td>Grasp rules generator, Rules generator factory</td>
<td>UML rules generator</td>
</tr>
<tr>
<td>Conformance Evaluation</td>
<td>Runtime evaluator, Conformance rules, Evaluator factory</td>
<td>Static evaluator, Other conformance rules</td>
</tr>
<tr>
<td>Event Monitoring and Genericising</td>
<td>Java runtime monitor, Runtime events, Implementation descriptors, Monitor factory</td>
<td>Java static monitor, Static events</td>
</tr>
</tbody>
</table>

**Table 4.1:** Currently implemented components in the framework and possible future components.
4.4 Choice of Modelling Notation

The current implementation of the PANDArch framework makes use of the Grasp ADL to describe architectures of software to be checked for conformance. While the conformance checking capabilities of the framework are designed to be independent of any specific modelling notation, Grasp is chosen for this purpose mostly due its adaptability and familiarity, and also its annotation capability that can be readily used for mapping architecture properties to implementation. Hence a rules generator component that processes specifications written in Grasp is built as part of the framework implementation. A detailed explanation of Grasp related adaptations used for conformance checking is given in Section 5.6.1 in the following chapter.

![Deployment architecture of the PANDArch framework.](image)
4.5 Evaluation of the Design

An important consideration regarding the design of PANDArch is evaluating whether it has established the necessary foundation for satisfying the core principles of the framework. The following table summarises the contribution of the adopted design strategies towards this end.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Realisation in the Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pluggable</td>
<td>The design achieves pluggability by adopting an event-driven approach for gathering implementation metadata. By listening to events emitting from the software, either through a VM hosting the executing application or an external tool scanning its static code, the framework can operate detached from the application. This physical separation is possible because the event listener is not part of the software producing the events, and events are generally transmitted through a network channel or interprocess communication.</td>
</tr>
<tr>
<td>Non-intrusive</td>
<td>The event-driven design also provides the foundation for extracting implementation metadata without altering the application source code. Events produced by an executing application occur in response to changing state, while events from static code analysis are generated when certain programming constructs are encountered during the scanning process. Neither of these require any code modifications. The design proposes instrumentation probes as an optimisation over the JDI for collecting runtime events of Java applications. While these probes are injected into the memory images of compiled Java classes, they do not alter behaviour, nor do they persist after the application terminates.</td>
</tr>
<tr>
<td>Automated</td>
<td>The automation principle is primarily satisfied through the use of components that parse architecture specifications and translate their essential properties into conformance rules. The framework requires no further manual intervention for checking conformance once an architecture description along with its implementation mappings are made available. Furthermore, the event-driven design also contributes towards automation, and conformance rules are evaluated as and when relevant events are received. Conformance violations are often reported immediately after a rule has been evaluated against an event.</td>
</tr>
</tbody>
</table>

*Table 4.2: The evaluation of the design of PANDArch against guiding principles (part 1).*
### Principle and Realisation in the Design

<table>
<thead>
<tr>
<th>Principle</th>
<th>Realisation in the Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification-driven and notation-independent</td>
<td>The design satisfies the first portion of the principle by allowing architecture conformance rules to be generated from architecture specifications. As conformance rules are the primary points at which conformance checking takes place, the entire process of checking conformance is driven by an architecture specification of the subject application. The second part of the principle is satisfied by adopting an extensible component-driven approach whereby components for processing different architecture notations plugged into the framework when required. The core framework is independent of a specific notation, and so are the conformance rules.</td>
</tr>
<tr>
<td>Incompleteness-tolerant</td>
<td>The design of PANDArch does not specifically cater for this principle. It is, however, accommodated in the implementation of the framework and therefore discussed in Chapter 5.</td>
</tr>
<tr>
<td>Self-contained</td>
<td>The framework design does not employ any external tools for checking conformance or satisfying other core requirements. Techniques proposed for optimising event processing and reducing performance implications are implementable in full with the technology available in the standard Java platform. If external libraries are used, these are packaged with the rest of the framework in manner that is not visible to the end user.</td>
</tr>
<tr>
<td>Performance-centric</td>
<td>Based on observations made during initial prototyping, the design suggests the use of instrumentation probes as an alternative to the JDI for improving the performance of Java runtime monitoring. However, more thorough performance-centric measures are taken in the implementation of the framework as discussed in Chapter 5.</td>
</tr>
<tr>
<td>Extensible</td>
<td>Extensibility is a cornerstone of the PANDArch design, which is enabled by employing an interface-oriented, component-based approach. All core elements of the framework such as monitors, evaluators, conformance rule generators and conformance rules are inherently loosely coupled with the body of the framework and with each other. New components can be built without affecting changes in the existing framework, while conforming to published interfaces allow these components to be easily integrated.</td>
</tr>
</tbody>
</table>

**Table 4.3:** The evaluation of the design of PANDArch against guiding principles (part 2).
4.6 Conclusions

This chapter presents the approach and the outcome of the PANDArch design process. A few important design decisions that reflect the guiding principles establish the ground rules at the beginning of the process. These decisions drive the architecture and the high-level design of the framework which, in turn, provide the foundation for implementing the framework using Java as the technology platform.

The adoption of an event-driven strategy in the design allows the framework to be pluggable. An application being monitored is physically and logically detached from the framework and is unaware of its presence. At most the application will only see a slowdown in performance, and monitoring can be completely stopped by restarting the application outside the framework. The use of events to interrogate the implementation also helps conform to the non-intrusiveness principle. The source code is not modified, and even though the compiled code is dynamically instrumented, application behaviour or state is not altered and the changes are not persistent.

The extensibility principle is met by following a component-oriented approach in the architecture and by the use of generic event objects and implementation descriptors. Allowing component interactions only through published interfaces enables new monitors and evaluators to be easily introduced. Generic metadata containers take this extensibility a step further by equipping the framework work with diverse event sources or technologies.

The design also caters for an automated conformance checking process, requiring the user the specify only the architecture and the implementations of the target application.

Finally, the design provides for a self-contained implementation of the framework that does not rely on other tools or frameworks to carry out its core objectives. The design, however, works through an adaptor mechanism to invoke static code analysers and other implementation data collectors. The same applies for generating conformance rules from architecture descriptions that are not specified in Grasp.

The next chapter presents the current Java implementation of the PANDArch framework.
The methodology adopted for implementing PANDArch plays a crucial role in achieving its objectives. While the implementation follows the architecture described in the previous chapter, the guiding principles also feed directly into some of the programming decisions. In particular, performance aspects of the framework have been given considerable thought at the implementation level. The implementation follows an interface-oriented strategy in order to conform to the component design in the architecture, and makes extensive use of Java generics. The chapter presents the implementation considerations of the core components in the framework. Both UML notations and informal diagrams are used for describing interesting programming aspects of these components. The chapter concludes with an evaluation of the employed implementation strategies against the guiding principles.

5.1 Introduction

Implementing the PANDArch framework involves expanding the architecture and high-level design to a more detailed level of abstraction and using these detailed abstractions as the basis for programming. The framework is implemented in Java. The rationale behind this decision is twofold. Firstly, it is a language and a technology with which the developer is very familiar. Secondly, this thesis focuses primarily on runtime conformance checking which involves harnessing events occurring in executing software to extract implementation details. Trapping execution events typically requires the help of the host infrastructure. In this regard, the JVM and the surrounding technology provide extensive, well-documented APIs to interrogate running Java applications. Therefore, Java provides a good basis for a first implementation of PANDArch. Use of Java also helps in the framework evaluation due to the availability of a large selection of Java open source software. However, the technology employed for implementing the framework should not hamper its ability to check conformance of software implemented in a
different technology. For instance, the Java implementation of PANDArch should be capable of monitoring not only Java software, but also software developed in C#, provided that the framework has implemented the necessary components for reading events originating from C# code or the .NET infrastructure.

In accordance with the PANDArch architecture, the implementation adopts the following Java namespaces (i.e. packages) to represent the different layers. These namespaces have additional namespaces within them. These are introduced when key components in each individual layer are discussed. The following table shows the top-level Java namespaces assigned to the different layers in the architecture.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Architectural Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ness.app</td>
<td>Application Services and Error Reporting</td>
</tr>
<tr>
<td>ness.generator</td>
<td>Conformance Rule Generation</td>
</tr>
<tr>
<td>ness.evaluator</td>
<td>Conformance Evaluation</td>
</tr>
<tr>
<td>ness.monitor</td>
<td>Event Monitoring and Genericising</td>
</tr>
<tr>
<td>ness.utility</td>
<td>Thread Pooling, Caching and Queueing (also includes other sundry services)</td>
</tr>
</tbody>
</table>

Table 5.1: Java namespace assignments for representing layers in the PANDArch architecture.

The remainder of this chapter, until the concluding section, discusses implementation strategies employed by the main components in the framework.

5.2 Shared Components

The PANDArch implementation has two groups of objects that are shared among all other core components of the framework. These are the implementation descriptors and generic events, which are explained in the next two subsections.

5.2.1 Implementation Descriptors

Implementation descriptors are used for describing various programming elements suitable for checking architecture conformance, such as interfaces and classes, in a platform neutral manner. These descriptors are primarily used for describing properties of generic events (see Section 5.2.2), but they can also be utilised for other purposes. The framework defines a number of implementation descriptors which are shown in Table 5.2. These descriptors are loosely modelled after the mirrored proxy objects in the JDI [152]. However, implementation descriptors
### 5.2. SHARED COMPONENTS

<table>
<thead>
<tr>
<th>Implementation Descriptor</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IInterfaceType</td>
<td>Describes an interface type in the subject implementation.</td>
</tr>
<tr>
<td>IClassType</td>
<td>Describes a class type in the subject implementation.</td>
</tr>
<tr>
<td>IArrayType</td>
<td>Describes an array type in the subject implementation.</td>
</tr>
<tr>
<td>INativeType</td>
<td>Identifies a native type in the subject implementation. Native types are typically types such as integer, float, boolean, etc.</td>
</tr>
<tr>
<td>IVoidType</td>
<td>Identifies a void type in the subject implementation.</td>
</tr>
<tr>
<td>IObjectReference</td>
<td>Describes an object (i.e. an instantiation of a class type) in the subject implementation.</td>
</tr>
<tr>
<td>IThreadReference</td>
<td>Describes a thread object in the subject implementation.</td>
</tr>
<tr>
<td>IMethod</td>
<td>Describes a method in the subject implementation.</td>
</tr>
<tr>
<td>ILocation</td>
<td>Describes a code location in the subject implementation. A code location is the line number and source file name of locatable implementation descriptor such as an IMethod.</td>
</tr>
</tbody>
</table>

#### Table 5.2: Implementation descriptors supported by the framework.

In PANDArch are light-weight in comparison to their JDI equivalents and do not mirror objects in the subject implementation.

## 5.2.2 Generic Events

Similar to native events, generic events convey implementation details of a software and the context in which the details were discovered. However, as the name suggests, generic events are free of any platform-specific properties, even though a generic event is often created from a matching platform-specific native event. The evaluator components in PANDArch rely on generic events to supply implementation metadata required for evaluating conformance rules. This metadata is contained in the properties of generic events as implementation descriptors.

Generic events are produced by monitor components, and therefore each class of monitors declares the types of events it supports. Nonetheless, the IEvent interface serves as the superinterface for all generic events. Typically, a monitor component declares a base interface that extends IEvent and binds to that monitor, from which all supported event type interfaces are derived. An example is the IRuntimeMonitor interface declared by runtime monitor components. This interface serves as the base for the different event types generated by every runtime monitor component in the PANDArch implementation.
Figure 5.1: The interface-oriented design used for implementing generic events. Note that the notation used for specifying the template parameter is Java specific.

The interface-oriented design of generic events is shown in Figure 5.1. The diagram has examples of two concrete event types, i.e., IMethodEntryEvent and IMethodExitEvent, extending IRuntimeMonitor through an intermediary interface. In addition, two interfaces in the example have methods (or properties) that return implementation descriptors discussed earlier. These properties play a crucial role as their values are used for evaluating conformance rules.

Generic events are implemented within the ness.monitor namespace. Section 5.3.1 provides more details of the specific event types currently supported by runtime monitors.

5.3 Monitor Components

In the PANDArch approach, monitor components are processors of native events. Native events originate from runtime sources such as a VM hosting an executing application or static sources such as a static code analyser. Monitor components connect with these native event sources, listen to events that these sources generate, and translate their native events to generic events understood by the rest of the framework. Monitors are also responsible for converting properties in native events that are platform specific into generic implementation descriptors.

Monitor components are implementation within the ness.monitor namespace which corresponds to the Event Monitoring layer discussed in Section 4.3.1. Table 5.3 describes the top level namespaces assigned to current and future implementations of different monitor components.
### Table 5.3: Java namespace assignments for implementing monitor components.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ness.monitor</td>
<td>Root namespace for monitor implementations and includes definitions of interfaces that all monitor components must implement. Also, interface definitions for implementation descriptors and generic event objects are contained here. In all cases, the interfaces here are generic in nature, and specialised by extending interfaces for specific implementations.</td>
</tr>
<tr>
<td>ness.monitor.impl</td>
<td>Includes abstract classes that implement base functionality for interfaces in the above namespace. These abstract classes serve as superclasses for classes that implement these interfaces in full.</td>
</tr>
<tr>
<td>ness.monitor.runtime</td>
<td>This namespace contains interfaces that are specific to components that monitor runtime events. These extend the generic interfaces in ness.monitor to make them specific to runtime event collection. As such, every runtime event object and runtime component is required to implement these interfaces.</td>
</tr>
<tr>
<td>ness.monitor.runtime.impl</td>
<td>Comprises of reusable abstract implementations for interfaces in the above namespace.</td>
</tr>
<tr>
<td>ness.monitor.runtime.impl.java</td>
<td>This namespace classes and other artefacts that implement a runtime event monitoring component for Java applications. Java specific implementation strategies such interacting with the JDI are found here.</td>
</tr>
<tr>
<td>ness.monitor.runtime.impl.net</td>
<td>Similar to the above, but this namespace includes implementation for monitoring applications hosted by the .NET framework (not yet implemented).</td>
</tr>
<tr>
<td>ness.monitor.statiic</td>
<td>Contains base interfaces for components and events related to static monitoring. This name space is the static counterpart of the ness.monitor.runtime namespace (not yet implemented).</td>
</tr>
<tr>
<td>ness.monitor.statiic.impl</td>
<td>Has abstract implementations for above interfaces (not yet implemented).</td>
</tr>
<tr>
<td>ness.monitor.statiic.impl.java</td>
<td>Holds code that implements a static monitor component for Java applications (not yet implemented).</td>
</tr>
<tr>
<td>ness.monitor.statiic.impl.net</td>
<td>Holds code that implements a static monitor component for .NET applications (not yet implemented).</td>
</tr>
</tbody>
</table>


Every monitor component implements an interface named IMonitor. However, this interface usually serves as the super-interface for more specific interfaces that are exposed by different classes of monitor components. As an example, the diagram in Figure 5.2 depicts the interface design of runtime monitors. The IRuntimeMonitor interface, which is the interface used by all runtime monitor components, extends IMonitor by providing IRuntimeEvent as the template (or binding) parameter. Binding IRuntimeMonitor to IRuntimeEvent ensures that runtime monitor implementations are tied to runtime events, allowing the Java compiler to perform stronger type checking. The same design pattern is followed for defining interfaces of other classes of monitors as well.

Figure 5.2: The interface-oriented design used for implementing monitor components. Note that the notation used for specifying the template parameter is Java specific.

The above design diagram also illustrates the approach taken to disseminate events from a monitor component to components in higher layers, such as evaluators. A client that requires event notifications from a monitor registers itself as an event listener with the monitor. A monitor allows registering separate listeners for the different types of events it supports. To register as a listener for a specific event type, the client invokes the corresponding addListener() method in the monitor. As the super-interface for all monitors, the IMonitor interface defines methods for adding listeners for two basic types of events supported by every monitor. Interfaces extending IMonitor may introduce methods for adding listeners for event types supported by different classes of monitors.
The IMonitor interface provides methods for specifying namespaces in the target application that should not be monitored. By informing a monitor to exclude (or only include) specific areas of the implementation from event notifications, an evaluator is able to target portions of the application for checking conformance. This facility is particularly useful when the architecture specification or the implementation is incomplete, or the implementation has external libraries that are not of interest to the evaluator. Furthermore, a client using a monitor component can limit the number of events it receives by using the sinkEvent() method to request the monitor to discard certain event types it does not require. The other methods in the IMonitor interface permit a client to control the flow of events from the monitor and query its status.

Although IMonitor forms the basis for implementing every monitor component, in its present state the framework defines only the IRuntimeMonitor sub-interface for implementing runtime monitors. Possible future expansion of the framework involves developing static monitors that expose an IStaticMonitor interface as well as dynamic distributed monitors implementing an IDistributedMonitor interface.

5.3.1 Runtime Monitors

Runtime monitor components are those that receive platform-specific native events from executing applications and distribute these events in a platform-independent and uniform manner to other components that consume these events. From an implementation point of view, a runtime component is one that implements the IRuntimeMonitor interface. As shown earlier, IRuntimeMonitor extends IMonitor by introducing methods that are related to runtime application monitoring and runtime event types. The bulk these additions are addListener() methods, through which clients can register listeners to events that occur specifically at program execution. Table 5.4 on the following page gives a list of runtime event types supported by IRuntimeMonitor. All these event types extend IRuntimeEvent either directly or indirectly.

As can be seen in Figure 5.2, IRuntimeMonitor interface provides two different methods for initiating an event listening session. The appLaunch() method allows the target application to be launched, after which the monitor starts receiving events. Alternatively, the monitor can be made to connect to an application that is already running through the appAttach() method. The event gathering process, in this case, begins immediately after the monitor establishes a successful connection with the application. In both cases, a monitoring session ends when the application terminates or the monitor disconnects from the application.

The framework currently has a single IRuntimeMonitor based monitor component for listening to events in running Java software. Details of this component are described next.
CHAPTER 5. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Runtime Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVmConnectEvent</td>
<td>Notifies when a monitor connects to the VM hosting the target application.</td>
</tr>
<tr>
<td>IVmDisconnectEvent</td>
<td>Notifies when a monitor disconnects from the target VM. This event and the above IVmConnectEvent are not used in evaluating conformance rules. However, they are useful for generating statistics and controlling the process flow of an evaluator.</td>
</tr>
<tr>
<td>IAppStartEvent</td>
<td>Notifies when the target application begins executions. This event is not raised if the monitor connects to an application that is already running.</td>
</tr>
<tr>
<td>IAppStopEvent</td>
<td>Notifies when the target application terminates.</td>
</tr>
<tr>
<td>IThreadStartEvent</td>
<td>Notifies the start of an execution thread in the target application.</td>
</tr>
<tr>
<td>IThreadStopEvent</td>
<td>Notifies the death of an execution thread in the target application.</td>
</tr>
<tr>
<td>IInterfaceLoadEvent</td>
<td>Notifies when the VM loads an interface type of the target application.</td>
</tr>
<tr>
<td>IClassLoadEvent</td>
<td>Notifies when the VM loads a class type of the target application.</td>
</tr>
<tr>
<td>IClassInitEvent</td>
<td>Notifies when a class type of the target application has been initialised. This initialisation refers to invoking the static constructor of a class, if any. This event is not raised if a class has no static constructor.</td>
</tr>
<tr>
<td>IObjectCreateEvent</td>
<td>Notifies when a class type has been instantiated.</td>
</tr>
<tr>
<td>IMethodEntryEvent</td>
<td>Notifies when execution enters a method. Private, protected and compiler generated (i.e. synthetic) method entries are not reported by this event.</td>
</tr>
<tr>
<td>IMethodExitEvent</td>
<td>Notifies method exits. Similar to IMethodEntryEvent, private, protected and synthetic method exits are not reported.</td>
</tr>
<tr>
<td>IExceptionEvent</td>
<td>Notifies when an exception is thrown in the target application.</td>
</tr>
</tbody>
</table>

Table 5.4: The list of event types supported by runtime monitors.

### 5.3.2 The Java Runtime Monitor

The Java runtime monitor component listens to events taking place in an executing Java program, transforms these events into generic event objects, transforms Java specific implementation metadata to generic implementation descriptors and makes these events available to other components in the PANDArch framework. The monitor works closely with the JVM hosting the subject Java application and performs numerous optimisations to limit the negative impact on the performance of the application. However, following the design principles of PANDArch, the events produced by this monitor are generic in nature and therefore do not carry Java specific implementation metadata.

The next two subsections discuss important implementation aspects of the Java runtime monitor.
5.3. MONITOR COMPONENTS

5.3.2.1 Event Collection Through the JDI

The primary source of events for the Java runtime monitor is the JDI. The JDI integrates tightly with the JVM to capture events in executing Java code and report these to a client.

To monitor a Java application, PANDArch may request the JDI to either launch the application or attach to a running instance of the application. The JDI is provided with the appropriate command line arguments if it is required to launch the application, or otherwise the network address of a running JVM instance to which it should attach. In either case, the JDI starts reporting events the moment it connects successfully to the JVM instance hosting the application.

The Java runtime monitor deals with the native Java events it receives through an internal sub-component called the Event Handler. This component in turn has two event listener modules, one listening to a JDI event stream and the other listening to events emitted by instrumentation probes embedded in the target application (see 5.3.2.2). These listeners execute on two separate threads, allowing them work independent of each other. This process is illustrated in Figure 5.3.

The Event Handler translates events received through either listener to their generic equivalents. In addition, Java specific metadata in native events are used for creating implementation descriptors before associating them with generic events. As shown, the Event Handler places
these descriptors in a cache as an optimisation strategy. A unique signature available in the corresponding Java type is used as the key for caching descriptors, allowing subsequent references to an implementation descriptor to be retrieved from the cache instead of creating anew. Caching is particularly effective when dealing with high-volume events like *method-entry* events, as it alleviates the need to repeatedly copy metadata from native events. Once a generic event has been created, the *Event Handler* places the new event in a queue and triggers the thread pool to dispatch the event using one of its threads. A dispatcher thread then removes the event from the queue and posts its to listeners registered for receiving that type of event. For this purpose, the monitor maintains a list each for listeners of each event type. An event is considered serviced once it has been posted to every listener in a list.

The use of a thread pool improves the throughput of event dispatching, but does not guarantee the natural order of events. The design assumes that event order does not matter for evaluating conformance rules. If event order must be preserved, which maybe useful for checking temporal architecture properties, a monitor can be configured with a single thread for dispatching events.

The Java monitoring component implements the following additional optimisations to improve performance of the event gathering and redistribution process.

1. **Unsubscribed events are disabled at the source.** Prior to connecting to the application, the Java monitor looks for event types for which there are no subscribed listeners and requests the JDI to disable reporting these events. For example, if there are no listeners for the *class-load* event type, the JDI does not trigger an event when an application class loads. The deactivation stays effective for the whole monitoring session. Disabling events at the point of origin contributes towards significant performance gains, in particular when used with high-volume events types such as *class-load* or *method-entry*.

2. **Thread suspension is kept to a minimum.** The JDI allows suspending the application thread in which an event occurs prior reading the event details. While required in some cases to ensure that event data stays relevant, suspending threads with every event has a substantial downside in terms of application performance. Therefore, in order to minimise performance implications, thread suspension in the current Java runtime monitor implementation is done only for *method-entry* and *method-exit* events.
3. **Generic runtime events are designed to be lightweight.** The generic runtime events contain only the essential information required for conformance checking. Although these events are designed to compatible with every runtime monitor implementation, having them lightweight is particularly important for JDI-based monitors. Reading metadata information from native events using the JDI is an expensive operation as these values are requested from the application on-demand. As only a selected few metadata fields are copied from native events to generic events, the performance impact is kept as low as possible.

### 5.3.2.2 Event Collection Through Instrumentation Probes

While the JDI makes available a comprehensive collection of events for mining implementation metadata from an executing Java application, using this API has a significant performance impact on the application being monitored. Notwithstanding the strategies employed to minimise the impact on performance, early tests conducted on the Java monitor component indicated that the JDI causes the target application to run approximately 360% slower with the monitor plugged-in than otherwise. A performance decrease of this magnitude can make certain classes of applications, such as server applications, hard to use and test. Although PANDArch can be easily decoupled from an application to allow normal operation, ensuring efficient testability during conformance monitoring is particularly important in real-world software projects. The approach taken by the JDI infrastructure to collect Java runtime events and runtime metadata makes further performance improvements of significant scale difficult to achieve. As per its documented specification, the JDI follows a design strategy that uses proxy objects to mirror objects created by the application [152]. When a runtime event occurs, properties in the relevant application objects are copied to their corresponding proxy objects. These proxy objects, which are maintained by the JDI, are exposed its clients as part of an event notification. Also, for some events, the application thread in which these events take place must be suspended prior to inspecting the corresponding proxy objects. Failure to do so could result in the data getting overwritten by subsequent events or becoming irrelevant as the thread continues execution.

In order to overcome the performance issues of the JDI, an alternative mechanism to capture certain runtime events using instrumentation probes is provided with the Java monitor. These probes do not alter the state or the behaviour of the application, and they do not persist in its source or compiled code. The instrumentation code is injected into the bytecode of the application classes after the JVM loads these classes into memory. The framework uses the Java instrumentation API and the agent technology [151] for this purpose. The Java agent containing the injection code (which is also written in Java) is deployed as a JAR file separate from the main PANDArch framework. The agent JAR is specified to the JVM along with other application launch arguments. The agent executes within the same JVM process as the application.
Once loaded, the monitoring agent registers an instance of a class that implements the `ClassFileTransformer` interface published as part of the Java instrumentation API. The JVM thereafter invokes the `transform` method of this interface for each application class it loads, providing access to the memory image of those classes. This mechanism allows the agent to modify the memory images of the required application classes prior to their initialisation.

In the current PANDArch implementation, the Java agent module captures three types of events. These are the `interface-load`, `class-load` and the `method-entry` events. These three are among the most prevalent events generated by a running Java program and optimising them can provide a significant performance improvement. Although the `method-exit`, `field-change`, and `object-create` events are also generated in high volume, these events are currently not used for evaluating conformance rules.

The agent module is able to capture the `interface-load` and `class-load` events without altering the Java bytecode as the JVM informs the agent when Java types are loaded into memory. In these two cases, the agent only has to inspect the type metadata structures to extract the required information. However, in order to trap `method-entry` events, the agent inserts instrumentation code at appropriate points in the application classes that are to be monitored. The ASM Java bytecode manipulation library [153] is used for modifying the bytecode of the in-memory images of the application classes. The instrumentation code takes the form of “probes” that call back static Java methods in the agent implementation. The agent needs to plant a number of these probes to be able to monitor `method-entry` events related to a single method. For instance, to monitor calls to method $M$, the agent places a pre-entry probe before every invocation of $M$ and a post-entry probe at the very beginning of $M$. The twin-probe approach enables the collection of caller and callee details without resorting to stack exploration techniques, which can be expensive in the case of the JVM. The probes also record caller and callee contexts of a method invocation. These contexts refer to the class instantiations (i.e. objects) upon which the method invocation takes places, and they are required for runtime conformance checking. The data returned from a pair of pre-entry / post-entry probes are used for constructing a single `method-entry` event. The diagram in Figure 5.4 illustrates this technique.

As the instrumented code in the application classes executes, a pre-entry probe first saves the caller data just before the method is invoked. The post-entry probe within the called method retrieves the caller data and combines this with the callee data to generate a complete `method-entry` event. An important consideration here is ensuring thread safety when saving caller details. To reliably process simultaneous method invocations on multiple threads, pre-entry probes save the caller data in a thread locale storage (TLS) variable.
Figure 5.4: Use of instrumentation probes to capture method-entry events in the Java agent implementation. The shaded pseudo-bytecode instructions in the two application classes represent the injected probes. In the current implementation of the Java runtime monitor, only method-entry events captured using probes. The technique, however, can be used for certain types other event as well.

The agent uses an asynchronous local socket channel to send events to the Java runtime monitor component in the main framework. It uses a simple textual protocol this purpose. An event is represented by a string message consisting of a number of fields. Fields are separated by ‘|’ characters, while commas serve as delimiters for lists within a field. The end of a message is marked with a newline character.

The grammar specification of the agent protocol using the Extended Backus-Naur Form (EBNF) notation is given in Listing 5.1.
Listing 5.1: The grammar specification of the agent protocol.

The Java monitor component de-serialises the textual event messages it receives from the agent into generic event objects. These events are then dispatched using the same mechanism used for distributing events originating from the JDI. The routine, therefore, is made completely transparent to components that consume generic events. Therefore, these components do not have to distinguish between JDI and agent events.

The agent implementation includes a number of optimisations to further reduce the performance impact on target applications. They are also designed to ensure that the memory footprint of the agent is kept as low as possible. These optimisations are discussed next.
1. **Uses a thread pool.** When sufficient information has been gathered to generate an event, the agent uses a worker thread from its own fixed-size thread pool to continue the generation process. This minimises the workload on the thread that collects the event data, which is particularly critical in the case of *method-entry* events. The probes used for capturing these events are executed by threads in the target application, and therefore have the biggest impact on its performance. The probes are, however, programmed to perform the bare minimum needed to collect the required event data, after which they invoke a worker thread. Similarly, the bulk of the workload involving the generation *interface-load* and *class-load* events is processed in worker threads to minimise the slowdown at application load time. The task of a worker thread is to package the event data into a message in accordance with the agent protocol and thereafter dispatch the message. The thread pool implementation includes a queue where tasks are placed until a thread becomes available to process a task.

2. **Uses asynchronous sockets.** The agents dispatches event messages through an asynchronous socket channel. In addition to providing non-blocking write operations, the asynchronous sockets makes a best effort to accomplish these operations natively through the underlying host environment \[154\]. This arrangement allows queued tasks in the thread pool to be serviced sooner, and prevents the queue from growing unreasonably large.

3. **Indexes type names.** Names of classes or interfaces are indexed using a running counter. A unique number is assigned to a type with both *interface-load* and *class-load* events. These numeric identifiers are used in *method-entry* events to refer to caller and callee types. The \( \langle \text{id} \rangle \) field in the above agent protocol denotes this identifier. The Java monitoring component, which receives event messages from the agent, uses a simple lookup table to dereference type identifiers before constructing generic event objects. The use of type identifiers reduces the memory consumed by the message queue and increases transmission throughput.

4. **Event filters.** In order to prevent events that are not useful for checking conformance from reaching the main framework, the agent filters outs events related to specific types or method invocations. Events related to Java enum types, annotations and synthetic classes are not reported, while instrumentation is not added to monitor private, protected and synthetic method invocations. In addition, instrumentation is not performed on types in the standard Java libraries and in namespaces that are not part of the application. Although performing selective instrumentation at load time increases the time taken for an application to initialise, it makes a significant contribution towards minimising the performance impact when the application executes with the embedded probes.
CHAPTER 5. IMPLEMENTATION

5. **Keeps execution-time data mining to a minimum.** Probes inserted into the bytecode of the application classes for capturing *method-entry* events are coded with static details that can be obtained when these classes are first loaded. For instance, the name of the type that declares a method along with the method name are known when that method is instrumented. The agent can, therefore, minimise the runtime workload by looking for data that can be discovered only when the application code executes. In the current agent implementation, the only information probed after the application has begun executing are the caller and callee contexts.

These optimisations together with the low cost of executing the embedded bytecode probes give the agent a considerable performance improvement over the JDI. Early tests indicated that the performance impact on a target application made to execute with the agent amounts to approximately 40%. The agent, however, may not be a permanent replacement for the JDI. Firstly, the agent, in its current form, is capable of capturing only three types of events. The JDI, on the other hand, is able to support all the runtime events published by PANDArch. Secondly, in comparison to the JDI, the agent supplies only a limited amount of information in the events it generates. For example, a *method-entry* event of the JDI gives details about the thread that invokes the method, but this information is not available in events produced by the agent. Similarly, the JDI provides location information (i.e. line number and source file name) for these events which is not the case with the agent. Therefore, in order to cater to different requirements of higher-level components, the Java runtime monitor can be configured to use the JDI alone or together with the agent module. The mode of operation is decided, to a great extent, by a runtime evaluator component. If the evaluator needs details of execution threads, for example, then it needs to configure the Java monitor to use only the JDI.

### 5.4 Evaluator Components

In PANDArch, the task of an evaluator component is to listen to generic events produced by monitors and use implementation metadata contained in those events to evaluate a collection of conformance rules assigned to that evaluator. For this purpose, an evaluator subscribes to one or more monitors to receive the required events, and thereafter invokes those monitors with the necessary information to connect to the subject application.

Java classes that implement evaluator components are located in the `ness.evaluator` namespace. This namespace corresponds to the *Conformance Rule Evaluation* layer in the PANDArch architecture as discussed in Section 4.3.1. Table 5.5 on the following page describes the top level namespaces assigned to current and future implementations of different evaluator components.
5.4. EVALUATOR COMPONENTS

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ness.evaluator</td>
<td>Root namespace for evaluator implementations and includes definitions of interfaces that all evaluator components must implement. This namespace also include interface definitions for conformance rules. The interfaces here are generic in nature and specialised by extending interfaces for specific evaluator implementations.</td>
</tr>
<tr>
<td>ness.evaluator.impl</td>
<td>Includes abstract classes that implement base functionality for interfaces in the above namespace. These abstract classes serve as superclasses for classes that implement these interfaces in full.</td>
</tr>
<tr>
<td>ness.evaluator.runtime</td>
<td>This namespace contains interfaces that are specific to runtime evaluators. These extend the generic interfaces in ness.evaluator to make them specific for evaluating conformance rules against runtime events. The namespace also defines concrete rule types (i.e. non-parameterised interfaces) for implementing runtime conformance rules.</td>
</tr>
<tr>
<td>ness.evaluator.runtime.impl</td>
<td>This namespace includes a complete implementation of a runtime evaluator. In addition, implementations of a number of runtime conformance rules are contained here.</td>
</tr>
<tr>
<td>ness.evaluator.static</td>
<td>This contains base interfaces for components that evaluate conformance rules against static events. It additionally contains concrete type definitions for static conformance rules. This namespace is the static counterpart of the ness.evaluator.runtime namespace (not yet implemented).</td>
</tr>
<tr>
<td>ness.monitor.static.impl</td>
<td>Has abstract implementations for above interfaces (not yet implemented).</td>
</tr>
</tbody>
</table>

Table 5.5: Java namespace assignments for implementing evaluator components.

Every evaluator component exposes an interface that extends IEvaluator, the topmost supertype of evaluators. An evaluator interface is also bound to a monitor interface through Java generics. The design in Figure 5.5 shows that the IRuntimeEvaluator interface is bound to the IRuntimeMonitor interface, the IRuntimeEvent interface and the IRuntimeTarget interface. This means that an evaluator component implementing IRuntimeEvaluator can subscribe only to monitors that implement IRuntimeMonitor and listen to events only of type IRuntimeEvents. This ensures that a runtime evaluator processes only runtime events, and a runtime monitor is what produces these events. Also by binding to IRuntimeTarget, a runtime evaluator is allowed specify to targets that only a runtime monitor may launch.
An important function of an evaluator is accepting conformance rules for evaluation. A given conformance rule is bound to one specific event type that provides the most suitable information for its evaluation. Because an evaluator is always associated with a particular class of events, such as runtime events, conformance rules accepted by the the evaluator must also be bound to events belonging to the same class. The IRuntimeEvaluator interface shown in Figure 5.5 contains a number of addRule() methods for adding conformance rules tied to each supported runtime event type. For example, one of the addRule() methods accepts an IRuleClassLoad type conformance rule. As the name suggests, this conformance rule type is bound to the IClassLoadEvent event. As such, a conformance rule that needs metadata information of a class-load event for evaluation should implement the IRuleClassLoad interface.

When an evaluator receives an event from a monitor, it attempts to propagate this event to every conformance rule bound to that type of event. The evaluator maintains a list of conformance rules for each supported event type, whilst each list preserves the order in which the rules were added. This allows rules with a higher priority to be placed at the top of the list. The evaluation priority itself is determined by the component adding the conformance rules to the evaluator. For each received event, conformance checking is triggered by invoking the evaluate() method exposed by every conformance rule, giving the event as an argument. A conformance rule can either immediately use the event to evaluate its constraints, or cache its properties for later use. If the implementation metadata of an event fails to satisfy the constraints of a conformance rule, the evaluate() method returns false to notify the evaluator of the outcome. At this point the evaluator aborts invoking the remaining conformance rules for that event.
An evaluator also invokes the `conclude()` method in each conformance rule at the end of a monitoring session to give them an opportunity to evaluate constraints using previously cached implementation metadata, if any.

The framework currently provides a complete implementation for a single runtime evaluator. As with all evaluators, the runtime evaluator is platform independent and able to function with any runtime monitor implementation. However, the framework has so far been tested by coupling the runtime evaluator to the Java runtime monitor component.

### 5.5 Conformance Rules

Together with evaluators, conformance rules perform the fundamental function of checking architecture conformance in PANDArch. A conformance rule is a combination of a constraint derived from an architecture property, and the mappings that link to the programming constructs implementing the architecture property. Following the familiar design pattern of other components, every conformance rule object exposes a derived interface of the `IConformanceRule` interface. Figure 5.6 below illustrates the design adopted for conformance rules.

![Figure 5.6](image_url)

**Figure 5.6:** The interface-oriented design used for implementing conformance rules. Note that the notation used for specifying template parameters is Java specific.

The diagram shows two examples of concrete conformance rules, namely `IRuleClassLoad` and `IRuleMethodEntry`, extending from an intermediary interface, i.e. `IRuntimeConformanceRule`, that acts as the base for all conformance rules associated with runtime events. The two concrete conformance rule types are bound to the `IRuntimeConformanceRule` interface using their respect runtime event types. In turn, the `IRuntimeConformanceRule`, bound to its super-interface using `IRuntimeEvaluator` and the binding parameter of its deriving interface.
The framework currently implements five conformance rules which are described below.

<table>
<thead>
<tr>
<th>Conformance Rule</th>
<th>Event Bound To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvalNamespace</td>
<td>IClassLoadEvent</td>
<td>Checks if the namespace associated with the event is a valid namespace in the architecture (i.e. one that is mapped to an element or otherwise has been specifically excluded).</td>
</tr>
<tr>
<td>EvalType</td>
<td>IClassLoadEvent</td>
<td>Checks if the type associated with the event is a valid type in the architecture (i.e. one that is mapped to an element or otherwise has been specifically excluded).</td>
</tr>
<tr>
<td>EvalProvidesInterface</td>
<td>IClassLoadEvent</td>
<td>Checks if the type associated with the event matches one of the types configured with this rule, and if so, whether the type satisfies a provides interface constraint in the architecture. The constraint is derived from the component or connector to which the type maps. If the element maps to more than one type, then at least one of those types should implement the provides interface configured with this rule.</td>
</tr>
<tr>
<td>EvalRequiresInterface</td>
<td>IMethodEntryEvent</td>
<td>Checks if the calling type associated with the event matches one of types configured with this rule, and if so, whether the type satisfies a requires interface constraint in the architecture. The constraint is derived from the component or connector to which the type maps. The called method must be an interface method, and the declaring interface is compared against the requires interface configured with this rule.</td>
</tr>
<tr>
<td>EvalLinkInteraction</td>
<td>IMethodEntryEvent</td>
<td>Checks if both the calling and called types associated with the event satisfy a link interaction constraint in the architecture. A single conformance rule of this type may hold a number of interaction constraints derived from link elements or implicit constraints related to connectors and components. Types implementing a component or connector are allowed to interact among themselves.</td>
</tr>
</tbody>
</table>

Table 5.6: List of conformance rules currently implemented in PANDArch.
While the logic for evaluating constraints is implemented within each conformance rule, they must be provided the required mapping information through constructor arguments. Furthermore, new conformance rules can be implemented as and when required to check conformance of architecture properties previously not available. However, the evaluation of the framework as described in this thesis is carried out using the conformance rules listed in Table 5.6.

Classes and interfaces implementing conformance rules are located in the ness.evaluator namespace.

5.6 Rules Generator Components

While evaluators and conformance rules together provide the means for checking architecture conformance, an important aspect of the PANDArch framework is the automated extraction of architecture properties and their corresponding implementation mappings from an architecture specification. Although encoding conformance rules manually is possible with the framework, doing so violates the principles of automation and a specification-driven approach. As detailed in Chapter 3, the architecture serves as the basis for checking architecture conformance of a software application. The conformance checker takes the architecture specification as one of its two mandatory inputs. The other is a reference to the implementation, which is the executable application for runtime checking or the program code for static checking. The generation of conformance rules is largely guided by predefined mapping specifiers and directives embedded in the architecture specification. Once a collection of conformance rules have been derived from the architecture, they are fed into an appropriate evaluator after which the process of conformance checking begins.

A component that parses an architecture specification to extract conformance rules is called a rules generator or a distiller. A given rules generator is able to processes a specific architecture modelling notation, such as an ADL, and the framework may implement a number of rules generators to handle different notations. Java classes that implement rules generator components are located in the ness.distiller namespace. This namespace corresponds to the Conformance Rules Generation layer of the PANDArch architecture. Table 5.7 on the following page describes the top level namespace assignments for classes implementing rules generators.
CHAPTER 5. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ness.distiller</td>
<td>Root namespace for rules generator implementations and includes definitions of interfaces that all rules generators must implement. The interfaces here are generic in nature and specialised by extending interfaces for specific implementations.</td>
</tr>
<tr>
<td>ness.distiller.impl</td>
<td>Includes abstract classes that implement base functionality for interfaces in the above namespace. These abstract classes serve as superclasses for classes that implement these interfaces in full.</td>
</tr>
<tr>
<td>ness.distiller.impl.grasp</td>
<td>Contains the complete implementation of a rules generator that parses Grasp architecture specifications.</td>
</tr>
</tbody>
</table>

Table 5.7: Java namespace assignments for implementing rules generator (i.e. distiller) components.

The general design of rules generators is illustrated in Figure 5.7. As shown, a rules generator component implements an interface derived from IDistiller. The current implementation of PANDArch contains a rules generator for parsing architectures specified in the Grasp ADL, and this component implements the IGraspDistiller interface. A rules generator implementation may, depending on its requirements, bind itself one or more available evaluator components. As the current implementation of the Grasp rules generator supports only runtime conformance checking, it binds to the runtime evaluator component discussed earlier.

Figure 5.7: The interface oriented design used for implementing rules generator (i.e. distiller) components. Note that the notation used for specifying template parameters is Java specific.
5.6. RULES GENERATOR COMPONENTS

5.6.1 The Grasp Rules Generator

The Grasp rules generator implemented as part of the PANDArch framework accepts a Grasp architecture specification as input and produces a collection of conformance rules from the architecture properties and their mapping details contained in the specification. The generated conformance rules are those that are currently supported by the framework. If the architecture specification contains properties that cannot be translated to any of the available conformance rules, those properties are ignored by the rules generator. The following sections discuss the adaptation of Grasp for use with PANDArch.

5.6.1.1 Specifying Architectures

Grasp is a textual ADL capable of modelling the static and runtime structures along with design rationale. Besides common architectural concepts such as components, connectors and layers, Grasp also supports annotations. These are name-value pairs that tag additional details to architecture elements without altering their semantics. In the case of PANDArch, annotations carry crucial mapping information linking architecture elements to their implementation.

The Grasp example in Listing 5.2 demonstrates the use of annotations. An annotation statement begins with the ‘@’ character followed by an identifier recognised by an external tool to which the annotation applies. As shown, all PANDArch specific annotations are labelled @confmon.

---

@confmon(ignoreCase=false)
@confmon(include=['"core"'])
@confmon(exclude=['"core.query.lucene"'])

architecture Jackrabbit {
  // Templates
  template NodeTypeComponent() { requires ISession; }
  template SessionComponent() { provides ISession; }

  system Core {
    // Components
    @confmon(ns=['"core.nodetype"'], classes=['"*"'])
    component NodeType = NodeTypeComponent();

    @confmon(ns=['"core.session"'], classes=['"*"'])
    component Session = SessionComponent();

    // Interactions
    link NodeType.ISession to Session.ISession;
  }
}

---

Listing 5.2: Using Grasp to specify architecture and implementation mappings
Annotations can be attached to first-class elements in a Grasp model. In the given example the architecture statement is decorated with four pairs of name-values, while the two component elements are attached two name-values each.

The Grasp compiler generates an in-memory graph representing the textual architecture model. Each node in the graph denotes an element in the architecture while an edge describes some relationship between two elements. Annotations are made properties in nodes (i.e. elements) to which they are attached. The rules generator component for Grasp uses this graph to abstract out properties in the architecture, from which conformance rules are generated after applying mapping information contained in the annotations.

A comprehensive reference of Grasp is given in Appendix A.

5.6.2 Mapping Architecture to Implementation

As stated earlier, PANDArch uses naming conventions to establish associations between elements in the architecture and programming constructs in the implementation. In some cases names used in the architecture may map directly to the corresponding construct in the implementation. In other cases a name reference in the architecture needs to be expanded and/or transformed to a programming language specific name. Often an architecture element may correlate to a number of items in the implementation or even have to be mapped to a programming concept with a different name. PANDArch caters to all these scenarios. While the annotation capability in Grasp is the medium through which these mappings are delivered, PANDArch defines precise mapping configurations an architecture specification should supply. The Grasp code snippet in Listing 5.2 shows a few of these mapping configurations. For instance, the baseNs configuration in the topmost annotation instructs the conformance checker to qualify all name references using the org.apache.jackrabbit namespace. Similarly, the ns and the classes configurations specified in the annotation attached to the NodeType component inform PANDArch that this component is implemented by all the classes inside the relative namespace of core.nodetype. The annotation for the Session component conveys similar information to the conformance checker about that component.

Not all architecture elements have to be annotated with implementation mappings. In the given example, the NodeType component has a “requires” interface named ISession that is connected to a “provides” interface of the same type in the Session component. Neither of these interfaces have an explicit mapping to the implementation. In the absence of an explicit mapping, the conformance checker uses the name of the element, i.e. ISession in this case, when generating the appropriate conformance rule.
The Grasp rules generator recognises the following directives and mapping configurations. The ‘Type’ column shows the Grasp data type that can be assigned to each mapping configuration.

<table>
<thead>
<tr>
<th>Mapping Configuration or Directive</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>List of strings</td>
<td>Specifies the namespaces in the target implementation that must be included for checking conformance.</td>
</tr>
<tr>
<td>exclude</td>
<td>List of strings</td>
<td>Specifies the namespaces in the target implementation that must be excluded from conformance checking.</td>
</tr>
<tr>
<td>prefixNs</td>
<td>String</td>
<td>Specifies the namespace used for prefixing (or qualifying) partially qualified namespaces in the model.</td>
</tr>
<tr>
<td>ns / namespace</td>
<td>List of strings</td>
<td>Specifies one or more namespaces that implements an element in the architecture. Partially qualified (i.e. relative) namespaces are resolved using the base namespace given in the prefixNs directive.</td>
</tr>
<tr>
<td>classes</td>
<td>List of strings</td>
<td>Specifies one or more classes that implements an element in the architecture. Accepts wildcards.</td>
</tr>
<tr>
<td>linkFrom</td>
<td>List of strings</td>
<td>Specifies one or more classes that implements the requires (or ‘from’) interface endpoint of a Grasp link element. By default the interface endpoints of link elements are mapped to the classes implementing the components exposing those interfaces. Accepts wildcards.</td>
</tr>
<tr>
<td>linkTo</td>
<td>List of strings</td>
<td>Specifies one or more classes that implements the provides (or ‘to’) interface endpoint of a Grasp link element. By default the interface endpoints of link elements are mapped to the classes implementing the components exposing those interfaces. Accepts wildcards.</td>
</tr>
<tr>
<td>ignoreCase</td>
<td>Boolean</td>
<td>Directs the conformance checker to ignore character casing when matching names in the architecture with those in the implementation. Default value is false.</td>
</tr>
</tbody>
</table>

Table 5.8: Mapping configurations supported by the Grasp rules generator.

Some mapping configurations accept wildcard characters for specifying classes or namespaces. The following two wildcards currently supported.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>All relevant constructs in a given namespace excluding those inside sub-namespaces.</td>
</tr>
<tr>
<td>*</td>
<td>All relevant constructs in a given namespace including those inside sub-namespaces.</td>
</tr>
</tbody>
</table>

Table 5.9: Mapping configurations supported by the Grasp rules generator.
5.7 Launching the Framework

The PANDArch framework is implemented as a standalone Java application. It is developed entirely in Java and therefore requires only the Java runtime environment (JRE) as a prerequisite for deployment. The framework is compatible with JRE version 1.6 or later.

PANDArch can be launched as a standard Java application from the command line of the host operating system. The main framework is packaged as an executable JAR file for convenience, while the deployment also has JARs containing the instrumentation probes and the Grasp compiler. The command for launching the framework is as follows.

```
java -jar pandarch.jar [-probes] [-cp <class-paths>] <architecture-spec> <application> [application-options]
```

The following table details the mandatory and optional arguments of the command.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-probes</td>
<td>Enables instrumentation probes for monitoring certain Java runtime events, while the JDI is used for others. If this option is not specified, the JDI provides the means for capturing all Java runtime events.</td>
</tr>
<tr>
<td>-cp &lt;class-paths&gt;</td>
<td>Java class path(s) used by the subject application. Multiple class paths should be specified in the exact same manner as they are used with the <code>-cp</code> option available in the <code>java</code> command.</td>
</tr>
<tr>
<td>&lt;architecture-spec&gt;</td>
<td>File or directory containing the architecture specification of the subject application. The current framework implementation supports only Grasp specifications.</td>
</tr>
<tr>
<td>&lt;application&gt;</td>
<td>Artefact required to launch the subject application. This should be a Java class file or an executable JAR file. No preprocessing of the compiled application code is required prior executing through the framework.</td>
</tr>
<tr>
<td>&lt;application-options&gt;</td>
<td>Options used by the subject application. These are passed exactly as they are to the application when it is launched through the framework.</td>
</tr>
</tbody>
</table>

Table 5.10: Command line arguments for launching PANDArch.

Given below is an example command for launching PANDArch. In this example, the SecondHop program is executed for testing conformance of Jackrabbit, which itself has been specified in the class path option. As shown, instrumentation probes have been enabled.

```
java -jar pandarch.jar -probes -cp jackrabbit-standalone-2.4.2.jar jackrabbit_core.grasp SecondHop
```
5.8 Evaluation of the Implementation

Similar to the strategies adopted in the design of PANDArch, various programming techniques employed for implementing the framework also contribute towards meeting the guiding principles. These are summarised in the following table.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Realisation in the Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pluggable</td>
<td>Satisfying the pluggability principle at the implementation level is primarily achieved by following the design, which provides the necessary foundation to make PANDArch pluggable. As detailed in Section 5.7, the Java implementation of the framework has ensured pluggability by allowing the original (i.e. unmodified) compiled code of the application to be launched through the framework whenever conformance checking is needed. Furthermore, as the required details for executing the subject application is given to PANDArch only at launch time, the framework has no prior knowledge of the application. Once conformance monitoring is complete, the same executable code can be deployed to operate the application as usual.</td>
</tr>
<tr>
<td>Non-intrusive</td>
<td>This principle, too, is primarily satisfied by following the design. The approach for implementing runtime probes makes use of the Java agent technology, which allows dynamic instrumentation of compiled Java code. This technology enables non-persistent runtime instrumentation of bytecode which conforms to the non-intrusive principle. Additionally, as can be gathered from Section 5.7, PANDArch does not require access to the application source code at any point.</td>
</tr>
<tr>
<td>Automated</td>
<td>This principle is mostly satisfied by following the design. However, the choice of architecture modelling notation, which is Grasp, also provides support for complying the framework implementation with this principle. The annotation capability, which is harnessed for embedding mapping information, along with the architecture graph produced by the Grasp compiler enable automated extraction of architecture properties and mappings from a Grasp model.</td>
</tr>
<tr>
<td>Specification-driven and notation-independent</td>
<td>The current implementation of PANDArch, which includes a rules generator for Grasp, is capable of parsing architectures specified in Grasp and generating conformance rules for checking conformance of running Java software. Therefore, the framework implementation can considered specification-driven. Nonetheless, the Java objects implementing conformance rules do not carry any Grasp specific information, making the framework implementation notation-independent.</td>
</tr>
</tbody>
</table>

Table 5.11: The evaluation of the implementation of PANDArch against guiding principles (part 1).
Chapter 5. Implementation

Principle Realisation in the Implementation

<table>
<thead>
<tr>
<th>Principle</th>
<th>Realisation in the Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompleteness-tolerant</td>
<td>Tolerance for incomplete architecture specifications is achieved mainly with the include and exclude directives that are part of the mapping scheme developed for Grasp. These two directives respectively instruct PANDArch to monitor only certain portions of the implementation and refrain from monitoring certain portions of the implementation. The provision for incremental conformance checking of a developing software system, or when only a partial specification of the architecture is available. The implemented mapping configurations and the use of wildcards also permit checking conformance at the level of detail available in the architecture. For instance, if an architecture specifies only the top-level components and not their sub-components, then conformance is checked only for those top-level components.</td>
</tr>
<tr>
<td>Self-contained</td>
<td>As dictated by the design, PANDArch does not make use of any external tools for implementing the required capabilities. Although the implementation employs a third-party library for instrumenting Java code, this library is packaged with the rest of the framework and therefore not a prerequisite for deploying the framework.</td>
</tr>
<tr>
<td>Performance-centric</td>
<td>The framework implements and extends the performance specific strategies outlined in the design. Most of the performance related techniques are applied in the implementation of monitor components, and in particular, the Java runtime monitor. These are discussed in detail in Section 5.3.2.</td>
</tr>
<tr>
<td>Extensible</td>
<td>The implementation satisfies the extensible principle by complying with the component-oriented design of the framework. All major components including events, implementation descriptors, conformance rules, monitors, evaluators, and conformance rule generators have published interfaces that they expose. Any new component in these categories must also implement their respective interfaces, enabling the framework as a whole to be easily extended to support new event sources, architecture notations, etc.</td>
</tr>
</tbody>
</table>

Table 5.12: The evaluation of the implementation of PANDArch against guiding principles (part 2).
5.9 Conclusions

The detailed design and programming strategies adopted to implement the PANDArch framework are presented in this chapter. The chapter also discusses the approaches taken to ensure that the implementation conforms to the prescribed architecture. Internal workings of the important components in the framework are discussed along with their extensibility aspects.

A key implementation strategy is the use of Java generics to bind together, at the language level, components that work in a process pipeline. All major components in the framework including monitors, evaluators and rules generators follow this designed pattern.

The framework implementation pays particular attention to minimizing the performance impact on the application being monitored. The techniques employed in the implementation of monitor components, in particular, play a pivotal role in achieving this design objective. The Java runtime monitor component employs both the feature-rich JDI as well as high-performance instrumentation probes to capture runtime events of Java software. The framework gives the user the option to configure the Java monitor to use both these event gathering mechanisms in tandem, striking a balance between performance and comprehensive mining of implementation metadata.

The current PANDArch implementation contains a runtime evaluator component that works closely with the Java runtime monitor. Along with the evaluator, the framework also implements five conformance rules for checking conformance of Java applications at runtime. The conformance rules themselves are generated from Grasp architecture specifications which are processed using the Grasp rules generator component. The annotations capability in the Grasp ADL provides the means for embedding implementation mappings. The chapter details current mapping configurations used for linking various architecture elements to their code constructs.

The chapter concludes with an evaluation of the implementation strategies against the core principles of the framework.

The next chapter presents the evaluation of the currently implemented framework with the help of a test harness as well as three well regarded open source software products.
An important aspect in the development of PANDArch is the evaluation of the capabilities of the implemented framework. The chapter describes the overall approach taken for conducting evaluation experiments, along with their limitations and trade-offs. These experiments are performed with the help of a sample Java application and with non-trivial Java software. The results of each experiment are presented in the form of an evaluation report, covering both the detection of conformance violations and decline in performance due to runtime conformance monitoring with PANDArch. An analysis of the observed behaviour follows each report.

6.1 Introduction

The thesis takes a two phased approach for evaluating PANDArch. The first phase entails unit testing conformance rules, to ensure they produce the correct output for a given input. A purpose built sample Grasp architecture and its corresponding Java application are employed to facilitate validating each conformance rule. The second phase involves deploying PANDArch with real-world software implementations and their architectures for evaluating its conformance checking capabilities as well as performance. The open source community offers a large selection of mature Java software from which suitable candidates can be chosen for experimenting with PANDArch. An added advantage with open source software is the unrestricted access to source code, which is crucial for some of the evaluation strategies discussed later in this chapter.

The next two sections introduce the approaches for unit testing and evaluating the framework with non-trivial software. They are followed by the evaluation report for unit testing in Section 6.4, while sections 6.5 to 6.7 present results from evaluating PANDArch with each of the chosen open source Java software products. The chapter concludes with a discussion of the effectiveness of PANDArch with respect to this evaluation.
6.2 Unit Testing Conformance Rules

The first step of the evaluation process involves unit testing the conformance checking mechanism. Each implemented conformance rule is tested with possible architecture properties and their implementation mappings, and in certain cases, with different programming constructs. An architecture of a simple multi-layered application and its corresponding Java implementation are used as the testbed. Each conformance rule has a schedule of test cases that cover both the Grasp specification as well as the Java implementation. Test cases related to the architecture check that the correct conformance rules are generated for properties in the architecture, whereas the test cases for the implementation verify that these generated conformance rules are able to correctly detect conformance violations.

However, measuring the impact on the performance of the subject software is not included in this phase of the evaluation. The performance characteristics of the sample application supporting these tests is not representative of those software developed in practice, and therefore is not suitable for deriving reasonable conclusions. Performance related tests are instead conducted with the open source software in phase two of the evaluation.

It is also noted that the test cases developed for validating conformance rules are not exhaustive as they do not test all possible inputs to a conformance rule. Therefore correctness of conformance rules is not guaranteed through these tests alone. However, the chosen test cases for each conformance provide coverage to a large number of common Java programming techniques that can be used for implementing the architecture property checked by that conformance rule. On the other hand, employing a formal software verification technique such as model checking [155] could guarantee correctness of conformance rules. Such testing is envisaged as part of future work when additional conformance rules are introduced.

The unit test cases of each conformance rule and their outcomes are presented in Section 6.4.

6.3 Testing Conformance Detection and Performance

The major portion of evaluating PANDArch consists of testing the framework with nontrivial, production ready software. Three well adopted open source software products are chosen for evaluating the conformance checker. The remainder of this section details the procedure followed through for selecting, configuring, and deploying these software with PANDArch prior to commencing the experimentation.
6.3.1 Scope of Conformance Checking

The evaluation of PANDArch with external software is constrained by the following:

- **Employs software developed only in Java.** Although the current implementation of the framework can potentially monitor architecture conformance of software written in other programming languages targeting the Java VM (e.g. Scala), the implementation mappings required to support these languages need to be understood. Furthermore, at present the framework implementation provides a monitor only for Java specific runtime events.

- **Configures the framework only for runtime conformance checking.** As discussed in the preceding chapters, the fundamental design of PANDArch supports numerous event sources. The framework currently implements a functional Java runtime monitor and a corresponding evaluator for runtime conformance checking. Although certain parts of the current framework implementation may process some static code analysis events, a functional static monitor for Java and a static evaluator component have not yet been developed.

- **Monitors Java code hosted in a single instance of the Java VM.** A Java distributed events monitor is required for simultaneous monitoring of multiple Java VM instances hosting a single application. This component has not yet been implemented.

- **Deploys with JVM 7.** Due to subtle differences in the Java VM across different product versions and vendor implementations, PANDArch currently supports only the Oracle HotSpot VM or any other Java VM implementation that fully complies with the JVM 7 standard. This restriction applies to both the framework as well as the target application.

6.3.2 Choice of Applications

The evaluation of PANDArch is conducted with three well established open source software products that are of reasonable size and complexity. In order to increase the exposure of the conformance checker, the chosen systems come from three distinct classes of software. These software products are:

1. **Apache Jackrabbit:** A content repository and a content server application.

2. **jEdit:** A programmer’s text editor with a graphical user interface.

3. **Hibernate:** A framework for building systems with object relational mapping technology.

The strategies employed for evaluating PANDArch with each of these software systems are discussed in detail under their respective evaluation reports later in this chapter.
6.3.3 Methodology

6.3.3.1 Modelling the Architecture

A key challenge facing PANDArch evaluation is the lack of open source Java projects having well-documented, up to date and sufficiently detailed architectures useful for checking conformance. Due to this reason, an architecture recovery approach is taken to build an architecture specification of the subject software using a commercial product named Structure101 [68]. This tool is capable of generating a component view of a Java application by statically analysing its bytecode. The tool is primarily targeted towards detecting undesirable dependencies such as cyclic references among components in the implemented architecture. However, Structure101 is also useful for identifying the implemented component architecture, i.e. component implementations and their interactions as they appear in the compiled Java code. The steps involved in extracting the component architecture of a Java application using Structure101 and adapting this architecture for checking conformance with PANDArch are explained next.

1. **Initialise the static analyser.** Provide the compiled Java code of the target application to Structure101 to begin its static analysis process. This can be in the form of a single jar file, a folder containing a number of jar files, or the folder hierarchy of the application class files.

2. **Analyse the Java bytecode.** Set up Structure101 to perform a detailed analysis of the Java bytecode that also includes exploring method invocations among Java types. Although the tool is capable of detecting only static (i.e. non-reflective) method invocations, they are helpful for understanding runtime interactions among components.

3. **Demarcate implementation artefacts for architecture recovery.** Select the root namespace of the Java implementation for which conformance checking should be performed. A typical non-trivial Java application may have a number of top-level namespaces that map to external libraries, test harnesses, etc. These are generally not part of the software architecture of the application and therefore should be omitted when checking for conformance.

4. **Recover static architecture structures.** Request Structure101 to generate a component view of the architecture for the selected namespace. The tool uses package, type and method relationships it discovers in Java class files to build this view. This architecture view serves as the basis for building a corresponding Grasp architecture specification of the software.
5. **Recover architecture relationships.** Request Structure101 to generate the dependency matrix for components identified in the previous step. A dependency matrix produced by this tool shows various relationships among a set of components in the implementation. These relationships are derived from the underlying classes in each component. For example, a class in component $A$ referring to a class in component $B$ is shown as a “references” relationship from $A$ to $B$ in the dependency matrix. In this case, $A$ is considered to be dependent on $B$. Similarly, if a method in $B$ invokes a method in $A$, the dependency matrix shows this as a “calls” relationship from $B$ to $A$. Besides these relationships, the two components in this example may simultaneously have other relationships between them. Each of these relationships can either make $A$ dependent on $B$ or vice versa.

6. **Describe the recovered architecture in Grasp.** Use the component view and the dependency matrix produced by Structure101 to manually construct a Grasp model of the implemented architecture. While every component in the recovered architecture is included in the Grasp specification, the “calls” relationships in the dependency matrix are modelled as runtime interactions using `link` statements in Grasp. A `link` element in Grasp connects a `requires` interface of a component to a matching `provides` interface of another component. The recovered component structure, however, lacks sufficient information to reliably deduce the two interfaces at either end of an interaction. Therefore, to overcome this limitation, every interaction is modelled as a generic, non-interface method call. Grasp supports this scenario by equipping every component with an intrinsic `out` interface to model outgoing calls and an equivalent `in` interface to model incoming calls. Moreover, components in real-world software do not always communicate through strict interfaces, and PANDArch has been designed to accommodate these practices. Other component relationships, such as “refers to” dependencies, are not modelled in Grasp.

7. **Map components in the Grasp architecture specification to Java code structures.** The component view generated by Structure101 shows the Java package names from which the tool inferred those components. This information is used to associate the components in the Grasp model back to the Java implementation.

8. **Set up directives for the conformance checker within the Grasp specification.** The conformance checker is provided with directives such as namespaces that should be excluded from the evaluation process. These directives embedded within the Grasp specification along with implementation mappings. The required directives are decided upon by manually examining the the component view generated by Structure101.
6.3.3.2 Inducing Architecture Violations

The notion of architecture conformance checking centres around the ability to compare software implementations against their intended architectures. As described in the previous section, architectures employed for evaluating the framework are implemented architectures derived from the Java bytecode of their corresponding applications. Conformance in this case is guaranteed, since the extracted architecture reflects the implementation. On the other hand, the implemented architecture of mature software often differs from the intended architecture of the software, which leads to architecture erosion. Nevertheless, in the absence of accurate architecture documentation or input from an authoritative source, building a useful Grasp model of the intended architecture is practically not feasible.

In light of this limitation, the approach taken to evaluate the framework against the chosen open source software is to artificially implant conformance violations. Two methods are employed in this regard. In both cases, the baseline architecture is the implemented architecture recovered from the code at the beginning of the experiment.

1. **Make changes in the Java code to deliberately breach the architecture.** Selected points in the Java implementation are altered in a manner that breaks conformance with the baseline architecture. Particular attention is given to making code changes that result in conformance violations detectable only at runtime. However, due to the lack of detailed design documentation, these modifications purposely do not alter the functionality of the software in order to avoid introducing defects.

2. **Change the architecture to intentionally diverge from the Java implementation.** The baseline architecture is altered to cause a discrepancy with the implementation.

Evaluation results are discussed under both these categories for each application tested with the framework.

6.3.3.3 Evaluating Performance Impact

To evaluate the effects of the framework on their performance, the applications are executed in three modes.

1. **Application executes outwith the framework.** In this scenario the test cases are executed without coupling the application to the PANDArch framework. Hence the application executes under normal conditions.
2. Application executes with the framework plugged in, which has been configured to use only the JDI. The tests are performed with the application coupled with the framework. In this case, the framework is configured to listen to runtime events through the JDI alone.

3. Application executes with the framework plugged in, which has been configured to use both probes and the JDI. This scenario is almost identical to the above, except that the framework is set up to use both instrumentation probes and the JDI. In this mode, the probes supply the Java monitor components with interface-load, class-load, and method-entry events, while rest of the events come from the JDI.

In all three cases, performance measurements are presented in both tabular and graphical forms to understand the variation of performance impact with the number of architecture violations, as well as illustrate the difference between the two Java event gathering mechanisms.

6.3.3.4 Verifying Non-intrusiveness

Testing whether PANDArch satisfies the non-intrusive principle is included in this evaluation. While the source code of the subject application is not made available to the framework, the compiled Java classes are tested whether they have been modified during conformance checking by comparing the MD5 digests (i.e. hashes) of the JAR packages taken before and after the experiments. These digests are reported with the rest of the results.

6.3.4 Test Environment

The work for evaluating the framework using the previously noted open source systems are executed on the same hardware and software environment. A specification of this environment containing the most relevant attributes is given Table 6.1. Results of the performance evaluation tests presented later in this chapter should be related to this system configuration.

<table>
<thead>
<tr>
<th><strong>Main processor</strong></th>
<th>Intel Core 2 Duo, 2.26 GHz (2 cores).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graphics processor</strong></td>
<td>NVIDIA GeForce 9400M with 256 MB of video memory.</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>8GB of DDR3 RAM, 1067 MHz.</td>
</tr>
<tr>
<td><strong>Hard disk</strong></td>
<td>128 GB SSD drive using SATA.</td>
</tr>
<tr>
<td><strong>Operating system</strong></td>
<td>OS X version 10.8.4 (64-bit).</td>
</tr>
<tr>
<td><strong>Java runtime</strong></td>
<td>JRE version 1.7 update 25 with Java HotSpot Server VM (64-bit).</td>
</tr>
</tbody>
</table>

Table 6.1: System specification of the environment employed for executing evaluation tests.
CHAPTER 6. EVALUATION

6.3.5 Limitations

Some limitations of the overall approach taken for evaluating the conformance monitoring framework with the three software systems are highlighted below.

- **Execution coverage.** A challenge faced by runtime conformance monitoring is ensuring sufficient execution coverage of the code in order to detect as many conformance violations as possible. Software testing techniques such as using various metrics generated from code coverage tools to improve test procedures can be adopted for increasing code coverage in conformance checking as well. However, these techniques require a thorough understanding of the system requirements. Nevertheless, guaranteeing complete executing coverage in non-trivial software can be considered unlikely. Therefore, the evaluation experiments presented in this chapter are subjected to execution coverage of the test cases used.

- **Limitations due to induced violations.** The approach taken for evaluating PANDArch involves artificially introducing conformance violations by either modifying the implementation or the recovered architecture to disagree with each other. Modifying the code of a well-tested software product requires careful prior analysis, which in turn needs considerable time and effort. However, due to time constraints and lack of reliable documentation, the changes made in the code to induce architecture violations are trivial by nature. They neither contribute to nor alter functionality. Hence, conformance violations resulting from this exercise could be considered somewhat superficial.

- **Limitations due to the use of reverse engineered architectures.** As a consequence of employing the recovered architecture for conformance checking, the implementation is not validated against the intended architecture of the software. This means that although PANDArch can be tested with the three evaluation candidates, it is unable to provide any assurances regarding compliance to their original architectures.

These aspects of the evaluation strategy are assessed in detail in the subsequent chapter along with suggestions for overcoming these challenges.
6.4 Evaluation Report: Unit Testing

The following subsections detail the unit test cases employed for validating each conformance rule. Each test case is assigned a reference, while the tests are ordered in increasing complexity. Every test case considers both the architecture and the implementation aspects of a conformance rule.

6.4.1 Conformance Rule: EvalNamespace

This rule checks whether the namespace from which an event originates is a valid namespace associated with the architecture. A valid namespace is one that has been mapped to an element in the architecture or otherwise has been specifically excluded from conformance checking.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1T1</td>
<td>Introduce a component to the architecture and map it to a namespace and a single class. Implement the class within the mapped namespace.</td>
</tr>
<tr>
<td>R1T2</td>
<td>Same as R1T1 above, but implement another class within an unmapped namespace. Make a reference to this class from the mapped class in the mapped namespace.</td>
</tr>
<tr>
<td>R1T3</td>
<td>Same as R1T2 above, but include a directive in the Grasp model to ignore the unmapped namespace during conformance checking.</td>
</tr>
</tbody>
</table>

Table 6.2: Test cases for verifying the EvalNamespace rule.

6.4.2 Conformance Rule: EvalType

This rule checks whether the type (i.e. a class) from which an event originates is a valid type associated with the architecture. A valid type is one that has been mapped to an element in the architecture or otherwise has been specifically excluded from conformance checking.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2T1</td>
<td>Introduce a component to the architecture and map it to a namespace and a single class. Implement the class within the mapped namespace.</td>
</tr>
<tr>
<td>R2T2</td>
<td>Same as R2T1 above, but implement another class within the existing namespace. Make a reference to this class from the mapped class.</td>
</tr>
<tr>
<td>R2T3</td>
<td>Same as R2T2 above, but include a directive in the Grasp model to ignore the unmapped class during conformance checking.</td>
</tr>
</tbody>
</table>

Table 6.3: Test cases for verifying the EvalType rule.
6.4.3 Conformance Rule: EvalProvidesInterface

This rule checks whether the type (i.e. class) from which the given event originates matches one of the types configured with this rule, and if so, whether the type satisfies a given provides interface constraint in the architecture. The constraint is derived from the component or connector element to which the type maps. If the element maps to more than one type, then at least one of those types should implement the provides interface configured with this rule.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3T1</td>
<td>In the Grasp model, create a component with a single provides interface, and map this component to a single class in the implementation. This class implements a matching interface as that of the component.</td>
</tr>
<tr>
<td>R3T2</td>
<td>Same as R3T1 above, but in this case the class does not implement an interface.</td>
</tr>
<tr>
<td>R3T3</td>
<td>Same as R3T1 above, but instead of a matching interface, the class implements an incompatible interface.</td>
</tr>
<tr>
<td>R3T4</td>
<td>Same as R3T1 above, but in this case the class implements two interfaces. One is a matching interface and other is an incompatible interface.</td>
</tr>
<tr>
<td>R3T5</td>
<td>Update the Grasp model in R3T1 above to map the component to two additional classes in the implementation. Create these two classes. Make the original class implement a matching interface as that of the component.</td>
</tr>
<tr>
<td>R3T6</td>
<td>Same as R3T5 above, but make a second mapped class implement a matching interface as that of the component. At this point two mapped classes implement the same interface.</td>
</tr>
<tr>
<td>R3T7</td>
<td>Same as R3T6 above, but make the remaining mapped class implement an incompatible interface. At this point two mapped classes implement a matching interface, while the third implements a different interface.</td>
</tr>
<tr>
<td>R3T8</td>
<td>Implementation remains same as that of R3T6 above, but update the Grasp model re-specifying the class mappings for the component using the ‘*’ wildcard.</td>
</tr>
<tr>
<td>R3T9</td>
<td>Same as R3T8 above, but remove interface implementations from both classes.</td>
</tr>
<tr>
<td>R3T10</td>
<td>The Grasp model remains the same as that of R3T8 above. In the implementation, create a new namespace (i.e. Java package) at the same level as the three classes. Create a fourth class within the new namespace. Make the new class implement a compatible interface to that of the component.</td>
</tr>
<tr>
<td>R3T11</td>
<td>Same as R3T10 above, but the Grasp model is updated by adding a new provides interface (i.e one that is different to the current interface) to the component.</td>
</tr>
<tr>
<td>R3T12</td>
<td>The interface added in R3T11 is implemented by one of the classes mapped to the component.</td>
</tr>
</tbody>
</table>

Table 6.4: Test cases for verifying the EvalProvidesInterface rule.
6.4.4 Conformance Rule: EvalRequiresInterface

This rule checks whether the calling type (i.e. class) contained in the given method-entry event matches one of types configured with this rule, and if so, whether the type satisfies a requires interface constraint in the architecture. The constraint is derived from the component or connector element to which the type maps. The called method must be an interface method, and the declaring interface is compared against the requires interface configured with this rule.

**Prerequisite:** In order to test a conformance rule for a required interface, the sample application must have a component with a compatible provides interface. For this purpose, a mock component with a single provides interface is specified in the Grasp model, after which the interface of the mock component and the one described in the test cases are linked. The mock component is also mapped to a single Java class that implements a matching interface having at least one method.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4T1</td>
<td>In the Grasp model, create a component with a single requires interface, and map this component to a single Java class in the implementation. The class invokes the interface method in the mock component.</td>
</tr>
<tr>
<td>R4T2</td>
<td>Same as R4T1 above, but the class mapped to the mock component does not implement the interface. Both the interface definition in the mock component and the link are removed. The test class invokes an ordinary public method in the class mapped to the mock component.</td>
</tr>
<tr>
<td>R4T3</td>
<td>Same as R4T1 above, but instead of a matching interface, the class mapped to the mock component implements an incompatible interface. The interface definition in the mock component is changed accordingly and the link removed for this test to be executed.</td>
</tr>
<tr>
<td>R4T4</td>
<td>Update the Grasp model in R4T1 above to map the component to two additional Java classes in the implementation. Create these two classes. Make all three classes invoke the interface method in the mock component.</td>
</tr>
<tr>
<td>R4T5</td>
<td>Same as R4T4 above. However, introduce an additional interface to the mock component and implement this interface in its mapped class.</td>
</tr>
<tr>
<td>R4T6</td>
<td>Implementation remains same as that of R4T5 above, but update the Grasp model re-specifying the class mappings for the component using the ‘*’ wildcard.</td>
</tr>
<tr>
<td>R4T7</td>
<td>The Grasp model remains the same. Create a new namespace in the implementation at the same level as the three classes. Create a class within this namespace. The new class invokes the interface of the mock component.</td>
</tr>
<tr>
<td>R4T8</td>
<td>Same as R4T7, but remove the requires interface and its links from the model.</td>
</tr>
</tbody>
</table>

**Table 6.5:** Test cases for verifying the EvalRequiresInterface rule.
6.4.5 Conformance Rule: EvalLinkInteraction

This rule checks whether both the calling and called types (i.e. classes) in the given method-entry event satisfy a link interaction constraint in the architecture. A single conformance rule of this type may hold a number of interaction constraints derived from link elements or implicit constraints related to connectors and component elements in the architecture. For instance, types implementing a single component or connector are always allowed to interact with each other.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5T1</td>
<td>In the Grasp model, create component A with a requires interface, and component B with a provides interface of the same type as the interface of A. Map both components to two classes each in the implementation and let one class of component B implement its interface. Allow one of the classes mapped to component A invoke the other class of the same component.</td>
</tr>
<tr>
<td>R5T2</td>
<td>Same as R5T1, but update the Grasp model by linking the two components through their interfaces.</td>
</tr>
<tr>
<td>R5T3</td>
<td>Same as R5T2, but update one of the classes mapped to component A invoke an interface method in the mapped classes of component B.</td>
</tr>
<tr>
<td>R5T4</td>
<td>Retaining R5T3 as the basis, add a third component C to the Grasp model. Equip this new component with a provides interface of the same type as the interface of A. Map C to a single Java class, ensuring that it implements the interface of C. Make one of the classes of component A invoke an interface method in the class mapped to component C.</td>
</tr>
<tr>
<td>R5T5</td>
<td>Same as R5T4, but make one of the classes of component A invoke a non-interface public method in the class mapped to component C.</td>
</tr>
<tr>
<td>R5T6</td>
<td>Same as R5T5, but update the Grasp model by specifying a link from the implicit out interface of component A to the implicit in interface of component C.</td>
</tr>
<tr>
<td>R5T7</td>
<td>Same as R5T6, but update the Grasp model by re-specifying the class mappings for components A and B using the ‘*’ wildcard.</td>
</tr>
<tr>
<td>R5T8</td>
<td>Baseline remains R5T7. In the Grasp model, create the nested child component D within component B. In the code, create a new Java namespace that is at the same level as the classes of component B. Map component D to a class within this new namespace. Allow component A to invoke a public method in component D.</td>
</tr>
<tr>
<td>R5T9</td>
<td>Same as R5T8, but specify a link in the Grasp model from the implicit out interface of component A to the implicit in interface of component D.</td>
</tr>
<tr>
<td>R5T10</td>
<td>Same as R5T9, but specify another link in the Grasp model from the out interface of component D to the in interface of component C. Allow the class mapped to component D invoke a public method in the class mapped to component C.</td>
</tr>
</tbody>
</table>

Table 6.6: Test cases for verifying the EvalLinkInteraction rule.
### 6.4.6 Results of Unit Testing

The following table shows a summary of the results for the unit test cases described in the previous section.

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Expected Result</th>
<th>Test Outcome</th>
<th>Test Case #</th>
<th>Expected Result</th>
<th>Test Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1T1</td>
<td>No violation</td>
<td>Passed</td>
<td>R4T1</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R1T2</td>
<td>Violation</td>
<td>Passed</td>
<td>R4T2</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R1T3</td>
<td>No violation</td>
<td>Passed</td>
<td>R4T3</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R2T1</td>
<td>No violation</td>
<td>Passed</td>
<td>R4T4</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R2T2</td>
<td>Violation</td>
<td>Passed</td>
<td>R4T5</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R2T3</td>
<td>No violation</td>
<td>Passed</td>
<td>R4T6</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T1</td>
<td>No violation</td>
<td>Passed</td>
<td>R4T7</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T2</td>
<td>Violation</td>
<td>Passed</td>
<td>R4T8</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T3</td>
<td>Violation</td>
<td>Passed</td>
<td>R5T1</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T4</td>
<td>Violation</td>
<td>Passed</td>
<td>R5T2</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T5</td>
<td>No violation</td>
<td>Passed</td>
<td>R5T3</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T6</td>
<td>No violation</td>
<td>Passed</td>
<td>R5T4</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T7</td>
<td>Violation</td>
<td>Passed</td>
<td>R5T5</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T8</td>
<td>No violation</td>
<td>Passed</td>
<td>R5T6</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T9</td>
<td>Violation</td>
<td>Passed</td>
<td>R5T7</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T10</td>
<td>No violation</td>
<td>Passed</td>
<td>R5T8</td>
<td>Violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T11</td>
<td>Violation</td>
<td>Passed</td>
<td>R5T9</td>
<td>No violation</td>
<td>Passed</td>
</tr>
<tr>
<td>R3T12</td>
<td>No violation</td>
<td>Passed</td>
<td>R5T10</td>
<td>No violation</td>
<td>Passed</td>
</tr>
</tbody>
</table>

*Table 6.7: Summarised results of unit testing.*
6.5 Evaluation Report: Jackrabbit

Apache Jackrabbit [157] is a content repository application that runs as a server. It also provides direct programmatic access to its data store through an API packaged alongside the server code. The evaluation tests for this application is executed in the server mode, whereby a sample client program is made to communicate with an instance of the Jackrabbit server running locally. Although the client relies on the Jackrabbit API to communicate with the server, only the server is plugged in with the PANDArch framework during conformance checking. The following table describes the deployment configuration of Jackrabbit used for evaluating PANDArch.

<table>
<thead>
<tr>
<th>Jackrabbit deployment mode</th>
<th>As a server running in the local machine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>2.4.2</td>
</tr>
<tr>
<td>Deployment JAR files</td>
<td>jackrabbit-standalone-2.4.2.jar</td>
</tr>
<tr>
<td>Root namespace used in conformance checking</td>
<td>org.apache.jackrabbit.core</td>
</tr>
</tbody>
</table>

Table 6.8: Jackrabbit deployment configuration used for evaluating the framework.

The root namespace describes the portion of the application that is considered for checking conformance. In the case of Jackrabbit, the server engine and most of the core functionality is implemented within this namespace. As discussed in Section [6.3.3], the root namespace also demarcates the implementation for static analysis through Structure101.

6.5.1 Recovered Architecture

The recovered component architecture of the Jackrabbit application is given in Figure [6.1] When Structure101 generates the component view of an architecture, it also arranges the components in architectural layers by applying its own heuristics over the explicit dependency relationships discovered in the code. However, these layering arrangements are not considered for evaluating the framework, and therefore not modelled in the corresponding Grasp specifications.

For the purpose of retaining clarity, the component view in Figure [6.1] does not show the relationships among components. These relationships are instead described in the dependency matrix in Figure [6.2]. The relationships in this matrix have directionality. As such, for a given relationship the component in the column signifies its source and the row is treated as the destination. An intersection with a check mark (✓) denotes the existence of at least one “calls” relationship between the two components involved. These are the only component relationships currently supported in Grasp and therefore used subsequently for checking conformance.
Though currently not used, the matrix also captures two other types of dependencies. The blue boxes imply the presence of one or more constructor calls between two components. This means a constituent class in the source component instantiates class that is part of the destination component. Invocations of class constructors are not treated as ordinary method calls, and therefore not modelled as component interactions in Grasp architecture models. The relevance of class instantiations in architecture models, however, remains a topic of future research.

The intersections coloured in red indicate other dependencies that are neither generic “call” relationships nor constructor invocations. These relationships consist of general references such as inheritance hierarchies, method arguments, and return values. However, similar to constructor invocations, these relationships are not included in conformance checking due to the unavailability of modelling support in Grasp.
Figure 6.2: The component dependency matrix for the recovered Jackrabbit architecture.
6.5.2 Grasp Architecture

The Grasp specification of the extracted Jackrabbit architecture is shown in Listing 6.1.

```java
@confmon(include="org.apache.jackrabbit.core")
@confmon(exclude="org.apache.jackrabbit.core.query.lucene")
@confmon(prefixNs="org.apache.jackrabbit")
architecture Jackrabbit {
    // Component templates
    template NamespaceComponent() {

    // Component architecture
    system Core {
        // Components
        @confmon(ns=".core.xml", classes="?")
        component xml = NamespaceComponent();
        @confmon(ns=".core.lock", classes="?")
        component lock = NamespaceComponent();
        @confmon(ns=".core.query", classes="*")
        component query = NamespaceComponent();
        @confmon(ns=".core.version", classes="?")
        component version = NamespaceComponent();
        @confmon(ns=".core.retention", classes="?")
        component retention = NamespaceComponent();
        @confmon(ns=".core.session", classes="?")
        component session = NamespaceComponent();
        @confmon(ns=".core.jndi", classes="*")
        component jndi = NamespaceComponent();
        @confmon(ns=".core.security", classes="*")
        component security = NamespaceComponent();
        @confmon(ns=".core.config", classes="?")
        component config = NamespaceComponent();
        @confmon(ns=".core.nodetype", classes="*")
        component nodetype = NamespaceComponent();
        @confmon(ns=".core.observation", classes="?")
        component observation = NamespaceComponent();
        @confmon(ns=".core", classes="?")
        component core = NamespaceComponent();
        @confmon(ns=".core.persistence", classes="*")
        component persistence = NamespaceComponent();
        @confmon(ns=".core.virtual", classes="?")
        component virtual = NamespaceComponent();
        @confmon(ns=".core.state", classes="?")
        component state = NamespaceComponent();
        @confmon(ns=".core.cluster", classes="?")
        component cluster = NamespaceComponent();
        @confmon(ns=".core.value", classes="?")
        component value = NamespaceComponent();
        @confmon(ns=".core.data", classes="*")
        component data = NamespaceComponent();
        @confmon(ns=".core.fs", classes="*")
        component fs = NamespaceComponent();
        @confmon(ns=".core.journal", classes="?")
    }
}
```

component journal = NamespaceComponent();
@confmon(ns=[".core.util"], classes=["*"])

component util = NamespaceComponent();
@confmon(ns=[".core.cache"], classes=["?"])

component cache = NamespaceComponent();
@confmon(ns=[".core.id"], classes=["?"])

component id = NamespaceComponent();
@confmon(ns=[".core.jmx"], classes=["?"])

component jmx = NamespaceComponent();
@confmon(ns=[".stats"], classes=["?"])

component stats = NamespaceComponent();

// Interactions
link xml.out to version, session, security, config, nodetype, core, state,
  value, util, id;
link lock.out to session, security, observation, core, state, cluster, value,
  fs, id;
link query.out to session, security, nodetype, core, state, cluster, value,
  fs, journal, id, stats;
link version.out to session, security, nodetype, observation, core,
  persistence, state, cluster, value, fs, id;
link retention.out to security, core, fs, id;
link session.out to security, observation, core, state, cluster, stats;
link jndi.out to config, core;
link security.out to xml, config, nodetype, core, cluster, fs, util, id;
link config.out to query, state, data, fs, journal, util;
link nodetype.out to session, security, observation, core, virtual, cluster,
  value, fs, util;
link observation.out to security, nodetype, core, state, cluster, value,
  journal, id;
link core.out to xml, lock, query, version, retention, session, security,
  config, nodetype, observation, persistence, state, cluster, value,
  data, fs, util, id;
link persistence.out to state, value, data, fs, util, cache, id, stats;
link virtual.out to nodetype, state, value, cache;
link state.out to nodetype, observation, core, persistence, virtual, cluster,
  value, cache, id;
link cluster.out to xml, config, observation, state, value, journal;
link data.out to persistence, state, value, util;
link fs.out to util;
link journal.out to version, util, id;
link util.out to config;
}

Listing 6.1: Grasp specification of the recovered Jackrabbit architecture.

An important aspect of Grasp specifications used for conformance checking with PANDArch
is the substantial use of annotations (see Section 5.6.2). These annotations deliver both
directives and mapping configurations to the conformance checker. The Grasp architecture
of Jackrabbit begins with three annotations with instructions for PANDArch to customise the conformance checker for Jackrabbit. The `include` directive specifies the root namespace of the implementation that maps to the described architecture, allowing the checker to safely ignore events originating outside this namespace.

The `exclude` directive in line 2, requests the framework to omit the given namespace, although it resides within the root namespace noted with the previous directive. The omitted namespace maps to Apache Lucene [158], an external component used by Jackrabbit to implement its search capability. Since Lucene is wrapped within a Jackrabbit component named `query` that has been specified in the Grasp architecture, failing to block out Lucene does not break the conformance checking process. However, masking sections of the implementation that do not contribute towards checking conformance can potentially improve performance of the application when the framework is plugged in. The remaining directive in the Jackrabbit model informs the framework to use the accompanying namespace to prefix any partially qualified namespaces in subsequent mapping configurations.

As can be seen in line 9, the component view of the Jackrabbit architecture is modelled within a Grasp system block. Each component has an associated attribute statement holding the mapping configuration that links the component to the implementation. These mapping configurations detail the namespace the classes involved in implementing the components. A complete explanation of supported directives and mapping configurations for Grasp can be found in Section 5.6.2.

After the components in the architecture have been defined, the Grasp description also models their interactions using a number of `link` statements. These statements connect components through compatible interfaces. However, as noted in Section 6.3.3, an architecture recovered using Structure101 does not detail interfaces in components, if any. Hence the corresponding Grasp specification links components through their intrinsic `out` and `in` interfaces to represent the ordinary “call” relationships in the dependency matrix. Each column in the dependency matrix translates to a single `link` statement. Components in columns are treated as consumers, and for each one of them, components in rows that are checked are treated as their providers. The Grasp syntax for modelling this relationship is:

```
link <consumer>.out to <provider1> [, <provider2> [, <provider3> [, ... ]]] ;
```

The `link` statements in the Jackrabbit model in Listing 6.1 exemplifies the above. This Grasp model serves as the basis for evaluating PANDArch with Jackrabbit.
6.5.3 Test Approach

All tests are carried out using a modified version of the SecondHop program distributed with Jackrabbit. This program signs in to the content repository, performs a few content operations and signs out. The sequence of content operations carried out are creating nodes, setting node properties, retrieving nodes and their properties, and removing nodes.

6.5.3.1 Induced Conformance Violations

In accordance with the evaluation methodology presented earlier in Section 6.3.3, architecture conformance in Jackrabbit is broken as follows. It should be noted that the two schemes employed for inducing violations are administered independent of each other.

Changes to the code: For testing a single conformance violation, the AddNodeOperation class in the session component is modified to instantiate the ReferenceChangeTracker class in the util component and invoke its clear() method using Java reflection. The AddNodeOperation class is used for creating nodes, while the interaction between session and util is not specified in the architecture and therefore should not be allowed. In addition, the fabricated reflective method invocation cannot be easily detected, if at all, through static analysis. Multiple violations in the code are made by invoking the same clear() method in the ReferenceChangeTracker class from other classes in the session component in a similar fashion.

Changes to the architecture: The link from component data to component state (line 89) is removed. The detailed dependency matrix generated by Structure101 shows that there are three “call” relationships that map to this link. Therefore, the conformance checker should flag each one of them as a conformance violation whenever those invocations are detected during execution.

6.5.3.2 Measuring Performance Impact

Tests for evaluating performance impact are carried out using the modified code where conformance violations are artificially planted. A total of eleven test cases are used, where the number of violations in the code increased incrementally from zero to ten. While the first test run with zero violations represents the case of a conformant implementation, a maximum of ten violations are chosen due to the limited knowledge of the Jackrabbit code base. Performance is measured by executing ten iterations of the test scenario during a single run of the test harness, and making four such runs for each test case. Execution time of each iteration is recorded with the help of standard Java timers placed within the test harness. The average execution time is used for analysis.
6.5.4 Results

Results from evaluating the framework with Jackrabbit are presented next.

6.5.4.1 Detecting Conformance Violations

As expected, the framework does not report any violations when the test harness is launched prior to modifying the Jackrabbit code or its extracted architecture. In this case all components and their interactions in the implementation are compliant with the architecture. In the case where the Java code is modified, PANDArch is able to identify the violation in the code. The framework flagged the violation with the following error message:

Conformance error <link violation> : component 'session' not allowed to communicate with component 'util' (AddNodeOperation.perform -> ReferenceChangeTracker.clear)

Multiple violations in the implementation are also identified similarly.

In the case where the architecture is modified to deviate from the Java implementation, the framework correctly reports the three non-conformant points in the code that corresponds to the broken link in the architecture. Note that, even though an interaction between two given end points can occur more than once, PANDArch reports only the first instance of that interaction.

Conformance error <link violation> : component 'data' not allowed to communicate with component 'state' (GarbageCollector.scanPersistenceManagers -> NodeState.getPropertyNames)
Conformance error <link violation> : component 'data' not allowed to communicate with component 'state' (GarbageCollector.scanPersistenceManagers -> PropertyState.getType)
Conformance error <link violation> : component 'data' not allowed to communicate with component 'state' (GarbageCollector.scanPersistenceManagers -> PropertyState.getValues)

The framework also notifies of component interactions that are specified in the architecture but did not occur during application execution. There are two possible reasons for undetected interactions. First is because some aspects of the implementation are non-compliant with the architecture, while the other is because certain parts of the implementation is not tested. In the case of Jackrabbit, the reason for these interactions not to occur is because those parts of the code are not executed by the test harness. In either case, PANDArch reports these messages once the application terminates. At this point the framework is able to conclusively determine any conformance rules that were not evaluated due to the absence of associated runtime events. A sample of the concluding messages issued by the framework for Jackrabbit is shown below.
6.5.4.2 Performance Impact

The results of performance tests are shown in Table 6.9.

<table>
<thead>
<tr>
<th>Run</th>
<th>Framework unplugged (µs)</th>
<th>Using probes (µs)</th>
<th>Using JDI (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>108,644</td>
<td>143,002</td>
<td>488,018</td>
</tr>
<tr>
<td>2</td>
<td>107,929</td>
<td>141,185</td>
<td>486,951</td>
</tr>
<tr>
<td>3</td>
<td>107,319</td>
<td>141,274</td>
<td>482,206</td>
</tr>
<tr>
<td>4</td>
<td>106,054</td>
<td>142,333</td>
<td>479,477</td>
</tr>
<tr>
<td>Average</td>
<td>107,487</td>
<td>141,949 (+30.9%)</td>
<td>484,163 (+345.6%)</td>
</tr>
</tbody>
</table>

Table 6.9: A comparison of performance impact between the use of JDI and instrumentation probes to capture runtime events. Execution times are in microseconds.

These numbers show that a significant performance gain is achieved by using instrumentation probes in place of JDI. Although Jackrabbit still runs almost 31% slower with probes, this may be an acceptable compromise given that the framework can be easily unplugged when conformance testing is not required.

Evaluating PANDArch with Jackrabbit also includes measuring variations of execution time as the number of architecture violations in Jackrabbit increase. The number violations in the code is increased incrementally up to a maximum of ten. Execution times are recorded for both JDI and probe configurations after each induced violation, results of which are shown in Figure 6.3. As shown in this graph, the performance impact in both cases remain almost constant as the number of violations rise.

6.5.4.3 Non-intrusiveness

The following table shows the MD5 digests of the Jackrabbit executable before and after checking for conformance. As indicated, the Java bytecode is not modified by the framework.

<table>
<thead>
<tr>
<th>JAR file</th>
<th>MD5 Digests (Before and After)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>jackrabbit-standalone-2.4.2.jar</td>
<td>2617389e1342a157d5177f0903c172fc, 2617389e1342a157d5177f0903c172fc</td>
<td>Unmodified</td>
</tr>
</tbody>
</table>

Table 6.10: The MD5 digests of the Jackrabbit code verifying the non-intrusiveness of PANDArch.
6.5. EVALUATION REPORT: JACKRABBIT

![Graph showing the performance of Jackrabbit as the count of architecture violations increases.](image)

**Figure 6.3:** Graph showing the performance of Jackrabbit as the count of architecture violations increases.

### 6.5.4.4 Discussion

Evaluation of PANDArch with a real-world server system has shown that architecture conformance checking at runtime is possible with this type of software. Performance is critical in server applications, and therefore any approach taken for monitoring architecture compliance of a running server must not allow the resulting impact on performance to alter the operating characteristics of the software. For example, a client should not timeout awaiting responses from a server that is being monitored. In the case of Jackrabbit, the 31% drop in performance due to PANDArch does not appear to cause behavioural changes. However, thorough loads testing, perhaps using a tool such as LoadRunner [159], is needed to reach a conclusion in the matter.

This evaluation test also demonstrates the framework’s ability to perform when only a partial architecture description of the target software is available. The architecture used in this experiment describes only the core functional parts of the Jackrabbit server. A number of other areas in the system such as the layer that implements a standards-based API for accessing the content repository is not covered in the architecture. While these unspecified parts may still execute during a monitoring session, PANDArch is able to ignore events originating from them. Namespace filtering makes it possible to mask out sections of the implementation not considered in the architecture as well as those areas that should be treated as black boxes. Excluding Lucene from the Jackrabbit conformance test exemplifies the latter case.
6.6 Evaluation Report: jEdit

The second application employed for evaluating the framework is jEdit [160], an open source programmer’s editor implemented entirely in Java. This application deployed as a single Jar file, which makes it both easy to extract the implemented architecture and test with PANDArch. As jEdit is a graphical user interface (GUI) application, evaluation is carried out without the help of an external test harness. Instead, a set of pre-designed test scenarios are used to manually carry out specific operations in the application through its user interface. Details of the jEdit configuration used in this evaluation are shown below.

<table>
<thead>
<tr>
<th>Deployment mode</th>
<th>As a GUI-based desktop Java application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>5.1.0</td>
</tr>
<tr>
<td>Deployment JAR file</td>
<td>jedit.jar</td>
</tr>
<tr>
<td>Root namespace used in conformance checking</td>
<td>org.gjt.sp.jedit</td>
</tr>
</tbody>
</table>

Table 6.11: jEdit deployment configuration used for evaluating the framework.

The root namespace specified above encompasses almost the complete implementation of jEdit.

6.6.1 Recovered Architecture

The recovered architecture of jEdit using Structure101 is shown in Figure 6.4. Although this component arrangement looks conceptually similar to the recovered architecture of Jackrabbit, a key difference is the presence of nested components in the jEdit architecture. As can be seen, component gui is a composite component that houses two other components, namely statusbar and tray. Note that although a child component called gui is shown within the top-level component of the same name, this due to a quirk in the way Structure101 presents the component view of an architecture. The child gui in reality represents the implementing classes of its parent component.

The dependency relationships among modules of the jEdit architecture are presented in Figure 6.5. The larger of the two matrices shows interactions of the top-level components while the smaller matrix has those of the nested components in gui. However, the interactions between the top-level components and the nested components are not considered here. Although such interactions can be easily modelled in Grasp, this information is not readily obtainable through Structure101. Instead the two dependency matrices show interactions between gui and its top-level counterparts, as well as between gui and its child components.
6.6. EVALUATION REPORT: JEDIT

6.6.2 Grasp Architecture

The Grasp specification of the extracted JEdit architecture is shown in Listing 6.2. The specification begins with the directive requesting the conformance checker to consider events arriving only from the namespace implementing the core architecture, namely the \texttt{org.gjt.sp.jedit} namespace. This is followed by a directive specifying the namespace prefix for fully qualifying relative namespaces found elsewhere in the specification.

The directives are followed by component declarations and their corresponding implementation mappings that have a similar pattern to that of the Jackrabbit architecture. However, a noteworthy extension is the declaration of nested components in lines 29–34. The statement declaring the \texttt{gui} component is a composite statement within which the two child component declarations are placed. Akin to other components, annotations with implementation mappings are associated to both the parent component and well as the two child component. The mapping configuration assigned to the \texttt{gui} component declares that the component is implemented using all the classes in the \texttt{jedit.gui} namespace, including those within its sub-namespaces. This works well for monitoring interactions between the \texttt{gui} component and other top level components. Furthermore, as noted earlier, the architecture does not specify communication paths between top level components and child components of \texttt{gui}.

Figure 6.4: The recovered component architecture of JEdit.
What the architecture does specify, however, are two interactions between gui and its child component named statusbar. These are declared in the Grasp specification using the two link statements in lines 89–92. Note that unlike other link statements, these two declarations have additional mapping configurations attached to them. The extra details are needed because the earlier mapping configuration for gui considers every class within its associated namespace, including those within sub-namespaces. As a result, the conformance checker is unable to distinguish interactions between gui and its child components, which are also contained within the namespace of gui. To overcome this difficulty, the Grasp specification provides additional clues in the form of linkFrom and linkTo configuration mappings. They identify classes implementing only the parent gui component enabling the conformance checker to correctly evaluate interactions between the parent and child components.

**Figure 6.5:** The component dependency matrices for the recovered jEdit architecture.
6.6. EVALUATION REPORT: JEDIT

architecture JEdit {
    // Component templates
    template NamespaceComponent() {}

    // Component architecture
    system Core {
        // Components
        @confmon(ns=[".jedit.pluginmgr"], classes=["?"])
        component pluginmgr = NamespaceComponent();
        @confmon(ns=[".jedit.help"], classes=["?"])
        component help = NamespaceComponent();
        @confmon(ns=[".jedit.menu"], classes=["?"])
        component menu = NamespaceComponent();
        @confmon(ns=[".jedit.browser"], classes=["?"])
        component browser = NamespaceComponent();
        @confmon(ns=[".jedit.options"], classes=["?"])
        component options = NamespaceComponent();
        @confmon(ns=[".jedit.search"], classes=["?"])
        component search = NamespaceComponent();
        @confmon(ns=[".jedit.bufferset"], classes=["?"])
        component bufferset = NamespaceComponent();
        @confmon(ns=[".jedit.input"], classes=["?"])
        component input = NamespaceComponent();
        @confmon(ns=[".jedit.datatransfer"], classes=["?"])
        component datatransfer = NamespaceComponent();
        @confmon(ns=[".jedit.gui"], classes=["*"])
        component gui = NamespaceComponent() {
            @confmon(ns=[".jedit.gui.statusbar"], classes=["?"])
            component statusbar = NamespaceComponent();
            @confmon(ns=[".jedit.gui.tray"], classes=["?"])
            component tray = NamespaceComponent();
        }
        @confmon(ns=[".jedit.msg"], classes=["?"])
        component msg = NamespaceComponent();
        @confmon(ns=[".jedit.print"], classes=["?"])
        component print = NamespaceComponent();
        @confmon(ns=[".jedit.textarea"], classes=["?"])
        component textarea = NamespaceComponent();
        @confmon(ns=[".jedit.indent"], classes=["?"])
        component indent = NamespaceComponent();
        @confmon(ns=[".jedit.io"], classes=["?"])
        component io = NamespaceComponent();
        @confmon(ns=[".jedit.buffer"], classes=["?"])
        component buffer = NamespaceComponent();
        @confmon(ns=[".jedit.bufferio"], classes=["?"])
        component bufferio = NamespaceComponent();
        @confmon(ns=[".jedit.proto.jeditresource"], classes=["?"])
        component jeditresource = NamespaceComponent();
        @confmon(ns=[".jedit"], classes=["?"])
        component jedit = NamespaceComponent();
        @confmon(ns=[".jedit.bsh"], classes=["*"])
    }
}
CHAPTER 6. EVALUATION

54  component bsh = NamespaceComponent();
55  @conmon(ns=[".jedit.syntax"], classes=["?"])
56  component syntax = NamespaceComponent();
57  @conmon(ns=[".jedit.visitors"], classes=["?"])
58  component visitors = NamespaceComponent();
59
60  // Interactions - top level components
61  link pluginmgr.out to browser, gui, io, jedit;
62  link help.out to gui, msg, io, jedit;
63  link menu.out to browser, gui, msg, io, jedit;
64  link browser.out to search, gui, msg, io, jedit, bsh;
65  link options.out to pluginmgr, bufferset, gui, textarea, buffer, jedit,
66  syntax;
67  link search.out to bufferset, gui, msg, textarea, io, buffer, jedit, bsh,
68  syntax;
69  link bufferset.out to msg, io, jedit;
70  link input.out to gui, textarea, buffer, jedit;
71  link datatransfer.out to textarea, io, buffer, jedit, syntax;
72  link gui.out to pluginmgr, browser, bufferset, input, msg, textarea, io,
73  buffer, jedit, bsh, syntax;
74  link msg.out to jedit;
75  link print.out to textarea, jedit, syntax;
76  link textarea.out to browser, bufferset, input, datatransfer, gui, io,
77  buffer, jedit, syntax;
78  link indent.out to buffer, jedit;
79  link io.out to browser, msg, bufferio, jedit;
80  link buffer.out to textarea, indent, jedit, syntax;
81  link bufferio.out to io, jedit;
82  link jeditresource.out to jedit;
83  link jedit.out to pluginmgr, browser, options, search, bufferset, input,
84  datatransfer, gui, msg, textarea, io, buffer, bsh, syntax, visitors;
85  link syntax.out to jedit;
86  link visitors.out to jedit;
87
88  // Interactions - nested components of 'gui'
89  @conmon(linkFrom=["?"])
90  link gui.out to gui.statusbar;
91  @conmon(linkTo=["?"])  
92  link gui.statusbar.out to gui;
93  }
94}

Listing 6.2: Grasp specification of the recovered jEdit architecture.

6.6.3 Test Approach

The following operational scenarios are used for evaluating PANDArch with jEdit. These
operations are performed manually through the jEdit user interface and each test scenario is
executed with the framework plugged in. Additionally, jEdit is launched afresh before executing
each scenario.
6.6. EVALUATION REPORT: JEDIT

<table>
<thead>
<tr>
<th>Scenario #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Once jEdit is launched, ensure that a blank new document is shown. Otherwise use <strong>File ▶ Close All</strong> to close any open documents. Type a line of arbitrary text on the blank document and save the document giving Scenario1.txt as its file name. Close the document.</td>
</tr>
<tr>
<td>S2</td>
<td>Launch jEdit and open the Scenario1.txt file saved in scenario S1. Position the cursor at the very beginning of the document. Use the <strong>Search ▶ Find...</strong> menu option to search for a word already contained within the text of the document. Close the document.</td>
</tr>
<tr>
<td>S3</td>
<td>Launch jEdit and open a simple Java source file (or any other type of source file supported by the syntax colouring feature). Print the Java file using the <strong>File ▶ Print...</strong> menu option.</td>
</tr>
<tr>
<td>S4</td>
<td>Launch jEdit and select the <strong>Utilities ▶ Global Options...</strong> menu option. Under the <strong>General</strong> section, check the <strong>Sort recent file list</strong> setting and click <strong>OK</strong> to save changes.</td>
</tr>
</tbody>
</table>

Table 6.12: Test scenarios used for evaluating the framework with jEdit.

6.6.3.1  **Induced Conformance Violations**

The above operational scenarios give coverage to the induced architecture violations in jEdit. All four scenarios are tested with the unmodified implementation and the architecture first to ensure that there are no spurious conformance violations.

**Changes to the code:** Modifications are made to the implementation to test scenarios S1 and S2 described above. As with the previous evaluation with Jackrabbit, new code is added to make non-conforming method invocations among classes using Java reflection. In the first such code update, targeting S1, the `save()` method in the `Buffer` class in the `jedit` component is modified to invoke the `nextTabStop()` in the `print` component. This method is implemented in the `PrintTabExpander` class, which happens to be a static inner class of the `BufferPrintable` class, and therefore a good candidate for testing the framework’s ability to correctly interpret this Java specific programming construct. Accessibility of the inner class has to be changed from package-private to public to make this work. Theses modifications violate conformance as the architecture does not specify a link from `jedit` to `print`.

In the second code change, architecture conformance is broken by making the `search` component communicate with the `pluginmgr` component. To achieve this, the `setSearchString()` method in class `SearchAndReplace` is modified to invoke the `getInstance()` method in class `PluginManager`. An interesting aspect of this code change is that both the caller and callee are static methods. It thereby provides an opportunity to investigate if PANDArch is capable of understanding component interactions that do not involve type instantiation (i.e. objects). This violation is tested by scenario S2.
Changes to the architecture: The link from component print to component syntax (line 75) is broken. This violation affects six method calls (or “call” relationships as labelled in Structure101) and is covered by test scenario S3. In addition, the link from jedit to options (line 83) is also removed, affecting eight method invocations. This conformance breakage is tested through scenario S4.

6.6.3.2 Measuring Performance Impact

Due to the manually operated, scenario-driven nature of the tests, evaluation of PANDArch with jEdit does not include measuring execution performance.

6.6.4 Results

Results from evaluating the framework with jEdit are given next.

6.6.4.1 Detecting Conformance Violations

Error messages issued by PANDArch showing conformance violations are presented under each operational scenario for which these notifications.

Scenario S1:

Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'print' (Buffer.save -> BufferPrintable.PrintTabExpander.nextTabStop)

As expected, the framework correctly identifies the non-conforming method invocation made through reflection. Although the scenario is executed a number of times during a single run of jEdit, the framework reports only the first occurrence of the violation, as per its implementation.

Scenario S2:

Conformance error <link violation> : component 'search' not allowed to communicate with component 'pluginmgr' (SearchAndReplace.setSearchString -> PluginManager.getInstance)

In this test too, the framework alerts of the noncompliant interaction forced into the implementation. The result also demonstrates that static method invocations (both end points are static in this specific case) in the implementation has no bearing on the ability to detect an invalid interaction.
Scenario S3:

Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.<init> -> SyntaxStyle.getFontColor)
Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.<init> -> SyntaxStyle.getFontBackgroundColor)
Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.<init> -> SyntaxStyle.getFont)
Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.printPage -> DisplayTokenHandler.init)
Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.printPage -> Chunk.paintChunkBackgrounds)
Conformance error <link violation> : component 'print' not allowed to communicate with component 'syntax' (BufferPrintable.printPage -> Chunk.paintChunkList)

Here, the framework reports that the detected interactions between components print and syntax are invalid. This is consistent with the change made to the Grasp model for this scenario.

Scenario S4:

Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.<init> -> GutterOptionPane.isGutterEnabled)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.<init> -> GutterOptionPane.getMinLineNumbers)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.<init> -> GutterOptionPane.isSelectionAreaEnabled)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.getPropertiesChanged -> GutterOptionPane.isGutterEnabled)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.getPropertiesChanged -> GutterOptionPane.getMinLineNumbers)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.getPropertiesChanged -> GutterOptionPane.isSelectionAreaEnabled)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.getPropertiesChanged -> GutterOptionPane.getSelectionAreaBackground)
Conformance error <link violation> : component 'jedit' not allowed to communicate with component 'options' (EditPane.getPropertiesChanged -> GutterOptionPane.getSelectionAreaWidth)
The first three conformance errors are reported the moment the application is launched, prior to executing scenario S4. This is expected as the accompanying error details reveal that the origin of the offending method invocation is the constructor of the `EditPane` class, which implements the core functionality of the editor. The remaining messages are shown only after the settings change is made in the `Global Options` dialogue box.

In addition to the conformance errors shown above, PANDArch also reports component interactions that were not encountered at the conclusion of each of the above tests.

### 6.6.4.2 Non-intrusiveness

The following table shows the MD5 digests of the jEdit executable before and after checking for conformance. As with Jackrabbit, the Java bytecode has not been modified by the framework.

<table>
<thead>
<tr>
<th>JAR file</th>
<th>MD5 Digests (Before and After)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>jedit.jar</td>
<td>3710a3534ba12146c9dbe33343aeeca2c</td>
<td>Unmodified</td>
</tr>
<tr>
<td></td>
<td>3710a3534ba12146c9dbe33343aeeca2c</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13: The MD5 digests of the jEdit code verifying non-intrusiveness during conformance checking.

### 6.6.5 Discussion

The preceding experiment exemplifies an approach for conformance testing non-trivial GUI applications. The strategy taken in this case is a manual scenario-driven process, similar to how software testing is generally performed with these types of applications. However, an alternative approach is to drive the application through its own scripting mechanism, if the capability exists, or use a test automation tool with the ability to record and playback GUI operations. A automated approach not only makes the testing process more productive and consistently repeatable, but it also measuring executing times useful for analysing performance implications. Nevertheless, due to large number of test cases needed simulate possible user inputs, checking architecture conformance in GUI applications can be more challenging compared to a server application or a software library.
6.7 Evaluation Report: Hibernate

The final software with which PANDArch is evaluated for this thesis is the Java implementation of Hibernate [161]. Hibernate is an object relational mapping (ORM) framework that facilitates persisting domain objects in a relational data store. Hibernate itself is not a database management system, but can be configured to work with numerous commercial and public domain database products. For this evaluation test, the chosen database is H2 [162]. H2 is a pure Java database engine that can be embedded in other Java software and requires no installation. Together with these characteristics and its ability to support in-memory databases, H2 provides an easily deployable database solution that can be coupled with Hibernate.

Unlike the candidates enlisted in the two previous evaluation tests, Hibernate is not a standard-alone application. Therefore, evaluating PANDArch with Hibernate requires a test harness. The test program developed for this purpose builds a simple domain object model which it then passes to Hibernate to be persisted as a relational database in H2. Although simple in scope, the test program exercises the four basic operations of a database, i.e. create, read, write and delete (CRUD), in order to achieve reasonable execution coverage of Hibernate.

The Hibernate configuration deployed for this evaluation test is as follows.

<table>
<thead>
<tr>
<th>Deployment mode</th>
<th>As a callable library linked to a test harness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4.2.2</td>
</tr>
<tr>
<td>Deployment JAR files</td>
<td>hibernate-core-4.2.2.Final.jar</td>
</tr>
<tr>
<td></td>
<td>hibernate-commons-annotations-4.0.2.Final.jar</td>
</tr>
<tr>
<td></td>
<td>hibernate-jpa-2.0-api-1.0.1.Final.jar</td>
</tr>
<tr>
<td></td>
<td>jboss-logging-3.1.0.GA.jar</td>
</tr>
<tr>
<td></td>
<td>jboss-transaction-api_1.1_spec-1.0.1.Final.jar</td>
</tr>
<tr>
<td></td>
<td>javassist-3.15.0-GA.jar</td>
</tr>
<tr>
<td></td>
<td>dom4j-1.6.1.jar</td>
</tr>
<tr>
<td></td>
<td>antlr-2.7.7.jar</td>
</tr>
<tr>
<td>Database system</td>
<td>H2 (configured as an in-memory database)</td>
</tr>
<tr>
<td>Root namespace used in conformance checking</td>
<td>org.hibernate.engine</td>
</tr>
<tr>
<td>Other</td>
<td>Logging is disabled</td>
</tr>
</tbody>
</table>

Table 6.14: Hibernate deployment configuration used for framework evaluation.

As with the two other evaluations, the conformance checker in this test is made to focus on the core architecture encompassed within the root namespace noted above.
6.7.1 Recovered Architecture

The component view of the recovered Hibernate core architecture is given in Figure 6.6, while its accompanying dependency matrices are shown in Figure 6.2.

Figure 6.6: The recovered component architecture of Hibernate.
A salient characteristic of the extracted Hibernate architecture is the presence of a number of nested components. In some cases the depth of nesting spans multiple levels. For example, the top-level component jdbc has three child components, one of which, i.e. the batch component, holds two child components of its own. As shown, dependencies of these nested components are described in the smaller matrices that are broken out from the top-level matrix. Also, similar to those of jEdit, component interactions in this Hibernate architecture take place only among siblings (i.e. components at the same nesting level) and between parents and their immediate child components. This limitation is due to the manner in which Structure101 presents views of the recovered architecture, though it does not hamper the conformance checker.
6.7.2 Grasp Architecture

The Grasp representation of the Hibernate architecture is consistent with those of the two previous evaluation tests. The specification contains a number of nested component statements to model structure of the discovered architecture. As with the jEdit specification, link statements that specify parent-child interactions have been assigned additional mapping configurations. An explanation in this regard is given in Section 6.6.2.

```plaintext
@confmon(include=["org.hibernate.engine"]) @confmon(prefixNs="org.hibernate") architecture Hibernate {
  // Component templates
  template NamespaceComponent() {

  // Component architecture
  system Core {
    // Components
    @confmon(ns=[".engine"], classes=["?"]) component engine = NamespaceComponent();
    @confmon(ns=[".engine.internal"], classes=["?"]) component internal = NamespaceComponent();
    @confmon(ns=[".engine.query.spi"], classes=["*"]) component query_spi = NamespaceComponent() {
      @confmon(ns=[".engine.query.spi.sql"], classes=["?"]) component sql = NamespaceComponent();
      component transaction = NamespaceComponent() {
        component jdbc = NamespaceComponent();
        @confmon(ns=[".engine.transaction.spi"], classes=["?"]) component spi = NamespaceComponent();
      }
    }
    @confmon(ns=[".engine.loading.internal"], classes=["?"]) component loading_internal = NamespaceComponent();
    @confmon(ns=[".engine.transaction"], classes=["*"]) component transaction = NamespaceComponent() {
      component internal = NamespaceComponent() {
        @confmon(ns=[".engine.transaction.internal.jdbc"], classes=["?"]) component jdbc = NamespaceComponent();
        @confmon(ns=[".engine.transaction.internal.jta"], classes=["?"]) component jta = NamespaceComponent();
      }
      @confmon(ns=[".engine.transaction.synchronization"], classes=["*"]) component synchronization = NamespaceComponent() {
        @confmon(ns=[".engine.transaction.synchronization.internal"], classes=["?"]) component internal = NamespaceComponent();
        @confmon(ns=[".engine.transaction.synchronization.spi"], classes=["?"]) component spi = NamespaceComponent();
      }
    }
    @confmon(ns=[".engine.spi"], classes=["?"]) component spi = NamespaceComponent();
  }
  @confmon(ns=[".engine.jdbc"], classes=["*"]) component jdbc = NamespaceComponent();
```

Listing 6.3: Grasp specification of the recovered Hibernate architecture.
6.7.3 Test Approach

As noted earlier, the evaluation of PANDArch with Hibernate is carried out with the help of a test harness. This simple Java program is linked with the required Hibernate libraries and the H2 database engine to form a standalone Java executable. The harness does not rely on a preconfigured database schema but instead instructs Hibernate to programatically create a new schema each time the code executes. The test program saves a few domain objects, modifies some that have already been saved, and also deletes a few. A Hibernate query command is issued after each of these operations, the results of which are displayed at end of the program. Standard Java timers are used for recording the execution time of a single run of the test program.

In terms of complexity, Hibernate is built upon a substantial Java code base. Identifying parts of the code for inducing architecture violations is a challenging task in the absence of developer guidance or detailed documentation. Hence the evaluation strategy is somewhat revised in this case. The test does not involve altering the code but rather only changes the Grasp model to make the architecture divergent from the implementation. The approach taken is to systematically remove component links in the Grasp model and executing the test program after each change. For this purpose links in the model are arranged into ten groups. The model is updated by removing one group of links at a time, until all the links in the Grasp specification are broken. This process is followed through twice with PANDArch plugged-in, once with instrumentation probes and the other with JDI.

The aforementioned link groups are described in tabular form in Figure 6.8. The right most column shows the total number of “call” relationships in the code associated with the links in each group. For a given group, this number translates to the number of unique conformance violations in the code, if all the links of that group are removed from the Grasp specification. The first group in the list, i.e. $G_0$, has no associated links and therefore used for reporting the case where no links in the architecture are broken. The ten subsequent groups labelled $G_1$ to $G_{10}$ all have one or more links included in each one of them. As can be seen, the arrangement of links into these ten groups is done in a manner that the distribution of “call” relationships attributed to each group gradually increases. For instance, $G_1$ may produce one conformance violation when removed, while the removal of $G_2$ could result in three such violations. On the other end of the scale, removing $G_{10}$ may cause 215 conformance violations in the implementation. The reason for arranging links into groups having increasingly large number of implementation mappings is to investigate the behaviour of the conformance checker with exponential increase of conformance violations. However, the numbers in the table do not follow a strict logarithmic trend.
### Figure 6.8: Grouped component links for evaluating the PANDArch with Hibernate.

The objective of this evaluation is to measure performance implications on Hibernate in both configurations of PANDArch, i.e. probes and JDI, while seeking to quantify conformance detection coverage for a given application (i.e. the test harness in this case). To this end, the test harness is first run without the framework to measure its unencumbered execution time. This figure is used as the baseline against with other execution times are compared. Thereafter, the harness is executed a number of times with the framework plugged-in and configured to gather events through instrumentation probes. Prior to each run, links in the Grasp model are removed one group at a time, in a cumulative manner. For instance, the first run has links from group G1 removed, while in the second run links from both G1 and G2 are taken out from the Grasp specification. After each run, the reported execution time as well as the detected number of unique conformance violations are recorded for analysis. The process is repeated after reconfiguring PANDArch operate in the JDI mode.
6.7.4 Results

The measurements taken from running the Hibernate test program with PANDArch are shown in Table 6.15. Each row represents a run after the removal of the corresponding group of links in the Grasp model. The reported execution times for both PANDArch configurations are given for each run of the test program. Shown alongside are the cumulative number of detected violations reported at the end of each run. It should be noted that the conformance checker produces the same number of violation detections in both its operating configurations. Furthermore, each execution time noted in the table is an average of four runs. The last row of the table shows the execution times averaged across all the runs for both modes of PANDArch. Given alongside these numbers are the performance impact as percentage of the unencumbered execution time.

The graph in Figure 6.9 plots the average execution times for the test program in all three conditions under which it is executed. Except for the slight increase in the presence of high numbers of conformance violations with the JDI setup, the execution time remains fairly constant in all other cases.

<table>
<thead>
<tr>
<th>Group (run)</th>
<th>Discrete violations</th>
<th>Cumulative violations</th>
<th>Detected violations</th>
<th>Exec. time: unplugged</th>
<th>Exec. time: probes</th>
<th>Exec. time: JDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>381</td>
<td>1,116</td>
<td>34,671</td>
</tr>
<tr>
<td>G1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>381</td>
<td>1,113</td>
<td>34,772</td>
</tr>
<tr>
<td>G2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>381</td>
<td>1,128</td>
<td>35,339</td>
</tr>
<tr>
<td>G3</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>381</td>
<td>1,098</td>
<td>35,052</td>
</tr>
<tr>
<td>G4</td>
<td>12</td>
<td>22</td>
<td>20</td>
<td>381</td>
<td>1,117</td>
<td>35,273</td>
</tr>
<tr>
<td>G5</td>
<td>24</td>
<td>46</td>
<td>41</td>
<td>381</td>
<td>1,117</td>
<td>35,092</td>
</tr>
<tr>
<td>G6</td>
<td>33</td>
<td>79</td>
<td>72</td>
<td>381</td>
<td>1,132</td>
<td>35,253</td>
</tr>
<tr>
<td>G7</td>
<td>47</td>
<td>126</td>
<td>116</td>
<td>381</td>
<td>1,124</td>
<td>35,237</td>
</tr>
<tr>
<td>G8</td>
<td>64</td>
<td>190</td>
<td>177</td>
<td>381</td>
<td>1,140</td>
<td>35,475</td>
</tr>
<tr>
<td>G9</td>
<td>154</td>
<td>344</td>
<td>322</td>
<td>381</td>
<td>1,154</td>
<td>35,426</td>
</tr>
<tr>
<td>G10</td>
<td>215</td>
<td>559</td>
<td>518</td>
<td>381</td>
<td>1,138</td>
<td>35,819</td>
</tr>
</tbody>
</table>

| Average    | 1,125 (+295%)       | 35,219 (+9,241%)     |

Table 6.15: Performance results from executing the Hibernate test harness with PANDArch. The harness outwith PANDArch is run once, with the execution time from this test used as the baseline for comparing others. All times are in milliseconds.
6.7.4.1 Non-intrusiveness

The MD5 digests for all the JAR files in the Hibernate deployment are shown below. As with the two previous experiments, the code is not modified by the framework.

<table>
<thead>
<tr>
<th>JAR file</th>
<th>MD5 Digests (Before and After)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>hibernate-core-4.2.2.Final.jar</td>
<td>38be1ba7cd1a9cf4ca46d69a7ae0586e 38be1ba7cd1a9cf4ca46d69a7ae0586e</td>
<td>Unmodified</td>
</tr>
<tr>
<td>hibernate-commons-annotations-4.0.2.Final.jar</td>
<td>916d4ddfb26db16da75ee8f973fd08ad 916d4ddfb26db16da75ee8f973fd08ad</td>
<td>Unmodified</td>
</tr>
<tr>
<td>hibernate-jpa-2.0-api-1.0.1.Final.jar</td>
<td>d7e7d8f60fc44a127ba702d43e71abec d7e7d8f60fc44a127ba702d43e71abec</td>
<td>Unmodified</td>
</tr>
<tr>
<td>jboss-logging-3.1.0.GA.jar</td>
<td>735bcea3e47fd715900cfb95ec68b50f 735bcea3e47fd715900cfb95ec68b50f</td>
<td>Unmodified</td>
</tr>
<tr>
<td>jboss-transaction-api_1.1_spec-1.0.1.Final.jar</td>
<td>679cd909d6130e6bf467b291031e1e2d 679cd909d6130e6bf467b291031e1e2d</td>
<td>Unmodified</td>
</tr>
<tr>
<td>javassist-3.15.0-GA.jar</td>
<td>2fca06eedcd3d3e5b0fe32416f99e1c 2fca06eedcd3d3e5b0fe32416f99e1c</td>
<td>Unmodified</td>
</tr>
<tr>
<td>dom4j-1.6.1.jar</td>
<td>4df851d3fe3900efc6e395be48030d6d 4df851d3fe3900efc6e395be48030d6d</td>
<td>Unmodified</td>
</tr>
<tr>
<td>antlr-2.7.7.jar</td>
<td>f8f1352c52a4c6a500b597596501fc64 f8f1352c52a4c6a500b597596501fc64</td>
<td>Unmodified</td>
</tr>
</tbody>
</table>

Table 6.16: The MD5 digests of the Jackrabbit code verifying the non-intrusiveness of PANDArch.
6.7.5 Discussion

The evaluation of PANDArch with Hibernate focuses mainly on understanding its performance aspects, particularly when the number of detectable conformance violations in the target application is high. An interesting outcome of this experiment is the considerably larger performance impact the framework causes on Hibernate as compared with the Jackrabbit experiment. When configured with instrumentation probes, the framework slows down Hibernate by a factor of almost three times, and with JDI this increases to over 92 times. In the case of Jackrabbit, the impact factors are 0.31 times and 3.5 times respectively. A possible reason for the larger performance penalty experienced with Hibernate is that an application using Hibernate is more processor intensive than a server application, such as Jackrabbit, servicing a single client request. On the other hand, the Jackrabbit implementation has a considerably larger code base than that of Hibernate. Therefore, a number of other factors can contribute to this significant performance parity, and an exact reason cannot be determined easily. Nonetheless, a noteworthy observation is the ratios between these performance factors in the two experiments. Jackrabbit works approximately 11 times faster with the probes mechanism than with JDI, whereas with Hibernate the probes are 31 times faster than its counterpart.

Another aspect of this experiment is investigating the number violations detected out of the total number of induced violations. The number of unique violations in the code for each broken link in the Grasp model is derived from the information made available by Structure101. However, runtime detection of these violations is subject to coverage of those points in the code during program execution. The detection numbers in Table 6.15 are therefore specific to the test harness (and its implemented functionality) deployed in this experiment. The results show that the rate of detection is closer to 100% when links have a few associated interactions in the code. The detection rate reaches 92% when every link is removed in the Grasp model.
6.8 Conclusions

The preceding sections of this chapter present the approach taken to evaluate the PANDArch framework. The first phase of the evaluation exercise performs unit testing on each conformance rule to ensure it produce the expected output. Each implemented conformance rule is individually validated with the help of a sample application, using a number of test cases that feed them with different combinations of input. The tests are designed in a manner that both the architecture model and the implementation contribute towards verifying these rules.

In the second phase of the evaluation, the framework is coupled with three substantial open source software products for the purpose of understanding the behaviour and the effectiveness of the conformance checker. Due the lack of reliable and suitably detailed architectures of the chosen software, a reverse engineering approach is taken to discover their architectures from compiled code. As a consequence, conformance is broken by either introducing violations to the code or altering the extracted architecture. Particular attention is given to implanting conformance violations in the code that are not detectable through static analysis methods. The process is applied to all three test subjects and the results show that PANDArch is able to identify most conformance violations from the executing code. It is thought that the violations the framework did not catch are in areas the code did not execute.

An important part of experimenting the framework with open source software is observing the performance characteristics of the subject software during conformance checking, an area not covered in the unit testing phase. In this regard, their execution times are measured under three conditions; without the framework and with the framework configured to collect events through instrumentations probes and the JDI. In addition, performance is measured against the number of architecture violations. Two of the three software products are subjected to performance tests. Their results show a large disparity in the performance impacts caused by PANDArch. While more experiments are needed to generalise a conclusion, the experiments presented here show that, in comparison to the JDI, the performance of instrumentation probes is better.

As a result of employing discovered architectures, PANDArch is only able to verify component interactions that are non-interface method invocations. The extraction tool is unable to recognise interactions arising through interfaces, if they exist at all. Therefore, the evaluation in this regard is somewhat limited to checking only one type of architecture conformance. Furthermore, with this exercise the framework is evaluated in terms of its capabilities, but not exactly with regard to taming architecture erosion. The extracted architectures have no clues as to how divergent they are from the intended architectures, nor do they provide design intent. These aspects are discussed in detailed in an overall assessment of the PANDArch framework in Chapter 7.
This chapter provides a critical assessment of the work presented in this thesis. The main research output, which includes the conceptual foundations, the design, and the subsequent implementation of the PANDArch framework, is first assessed against the core principles that guides the work. Other aspects related to the development and evaluation of the implemented framework are thereafter discussed along with areas of future improvement.

7.1 Adherence to Guiding Principles

The objective of this thesis is to propose, develop and evaluate a practical and effective runtime architecture conformance monitoring approach that could contribute to controlling software architecture erosion. In order to guide the research activities and, more importantly, be able to qualitatively assess the research outcome, eight core principles are established at the onset of the process. The following sections assess PANDArch against each of these principles.

7.1.1 Pluggable

This principle states that a runtime architecture conformance monitoring framework must be pluggable, so that such a framework is not a permanent fixture of the software it monitors. Neither should it require recompiling the target software, either before starting the monitoring process or afterwards. PANDArch satisfies both these criteria. The framework operates as an independent entity, while allowing users to launch the software they wish to monitor through the framework. No prior modifications are required at source code level and no restrictions are imposed on launching the application with the framework. Although instrumentation probes are injected on the fly as the application loads, these probes do not alter the state or behaviour of the software. Once monitoring is complete, the binary code of the application continues to remain in
its original form. The details for executing PANDArch as a standalone Java application given in Section 5.7 and the case studies presented in the Evaluation chapter exemplify the ease with which the framework can be coupled and decoupled with the subject software. Accordingly, PANDArch can be considered to have achieved the goal of pluggability both in its design and implementation.

### 7.1.2 Non-intrusive

As they are related, the design and implementation of PANDArch consider this core principle in tandem with the above principle of a pluggable framework. The non-intrusiveness principle requires the framework, in particular, to refrain from altering the source code of the application it needs to monitor. As noted above, PANDArch does not modify the source code or the compiled code prior to executing the application or upon its termination. When the framework is configured to use instrumentation instead of the JDI, it does so by injecting probes, where necessary, to the in-memory image of a Java class, thereby avoiding changing the class files on disk. Furthermore, the stateless probes do not cause any behavioural change in the application being monitored, besides the expected degradation in performance. Once the application has terminated, these in-memory instrumentation probes cease to exist, which is evidenced by the unmodified Java executables in the evaluation experiments. Therefore, as the implemented framework does not make permanent changes to the application, the PANDArch approach is regarded non-intrusive.

### 7.1.3 Automated

This principle dictates that the process of extracting architecture properties and implementation mappings from a specification, and the subsequent generation of conformance rules should be automated. Just as importantly, the detection of conformance errors in the implementation and their notification should not require any manual intervention. All these conditions are addressed in both the design and implementation of PANDArch. In its present state, the framework is capable of processing a Grasp architecture specification, extracting its various architecture properties along with mapping configurations embedded in annotations, to produce the corresponding conformance rules. These rules are then fed to the runtime evaluator component which validates each conformance rule against events occurring at runtime in the monitored software. A conformance rule that fails to validate against some property in the implementation is reported immediately as a conformance violation. The experiments evaluating the framework with the three open source products demonstrate that the only manual step involved in the PANDArch process is describing the architecture and mappings in Grasp.
7.1.4 Specification-driven and Notation-independent

Observing the specification-driven part of this principle obliges PANDArch to acquire all the required information for checking conformance from a specification of the subject architecture. The framework design caters to this requirement by stipulating that conformance rules be derived from architecture specifications through rules generator components. In the current framework implementation a Grasp rules generator performs this task. With the help of this component, the framework accepts Grasp architecture models containing implementation mappings and they are the only input besides the executable code of the corresponding software.

The remainder of the principle, i.e. making the framework notation agnostic is satisfied in the design of PANDArch but not so in the implementation. However, although Grasp is the supported notation in the implemented framework at this time, it does not preclude the framework from supporting other ADLs or modelling notations. The extensible design of PANDArch can accommodate different components for processing varying types of ADLs, while insulating other parts of the framework from these notation dependencies. For example, conformance rules created from a specific modelling notation have no information related to that notation, allowing the possibility for two entirely different notations to produce identical conformance rules. In addition, conformance rules can be hand-crafted, with no reference to an architecture specification. While this capability is mostly useful for debugging purposes, it does demonstrate that the framework design is indeed neutral with respect to architecture modelling notations. However, its implementation presently supports only Grasp.

7.1.5 Incompleteness-tolerant

To honour this principle PANDArch should utilise whatever properties available in the given architecture for effective conformance checking. The framework has this capability, for which evidence is presented in the jEdit and Jackrabbit evaluations. The extracted architectures in both these experiments do not have sufficient information to describe interactions among child components and sibling components of their parents. Although these interactions exist in the concrete implementations, the architecture models them as links between the parent component and its siblings. When checking for conformance, the framework is able to remap interactions in the code to the abstracted links in the architecture. In the event the architecture is updated with more complete interactions, the conformance checker is able to use this information accordingly. PANDArch also allows specifying exclusion areas in the implementation as well as configuring the conformance checker to work on specific regions. Exclusions are useful for marking sections of the implementation as “black boxes” where conformance checking may not be required, for example, an external library used in the application. Similarly, focusing the conformance checker
to operate on specific regions of the implementation is useful for ignoring, for instance, parts
that are of little architectural concern or still under development.

7.1.6 Self-contained

The principle for self-containment states that the framework should not depend on other software
tools for its deployment or operations. Both the design and the Java implementation of PANDArch
have respected this principle in their respective approaches. While the design follows a Java-
based strategy that can be easily adapted for another similar technology such as .NET, the
implementation expands this design to build the framework using the standard Java technology
stack. Although the Java agent module which injects instrumentation probes relies on the ASM
library for modifying Java bytecode streams, the use of this library is not considered as breaking
the principle. The library is packaged together with other Java code artefacts and is completely
invisible to the user. Therefore, the framework does not demand any prerequisites for deployment
besides the availability of the Java runtime environment.

7.1.7 Performance-centric

This principle, which mostly applies to runtime conformance monitoring, dictates that the
framework should, as much as possible, minimise the negative influence on the performance of
the application being monitored. As described in the previous chapters, a number of strategies
are adopted in both the high-level design and the implementation of the framework to meet
this objective. In particular, the use of instrumentation probes as an alternative to the JDI
to capture certain Java runtime events has proven to be effective in significantly lowering the
performance impact. However, results observed from the two performance experiments described
in Chapter 6 show that a large variance in the performance penalties inflicted by PANDArch.
At the same time, a benchmark against which performance levels can be compared does not
exist, nor are quantitative measurements from similar work available in literature. Therefore,
under these circumstances, a qualitative approach is taken to make a reasonable determination as
to whether the framework has realised its performance objectives. The two test programs used
in performance experiments involving Jackrabbit and Hibernate remained testable throughout
the experiments, while producing their expected output. Although a decline in performance
is observed in comparison to running the tests without the framework, the slowdown does not
impede their successful executions. Furthermore, the delays caused by instrumentation probes
are significantly smaller than those resulting from the JDI. It is therefore reasonable to assume
that PANDArch can be deployed in software projects to facilitate conformance testing alongside
regular testing without modifying existing test procedures. However, performance characteristics
of the framework under heavy load is not evaluated in this thesis, and therefore remains area of future work. Regardless, runtime conformance checking is expected to have some impact on the execution performance of the subject. PANDArch, while attempting to minimise this impact, is also pluggable, allowing conformance checking to be an on-demand activity in the software life cycle.

7.1.8 Extensible

Supporting the extensible principle allows the PANDArch framework to work with disparate mechanisms for mining implementation metadata as well as process architecture specifications of varying notations. In this regard, the design of PANDArch follows of a number of approaches. In the first instance, the use of multiple monitor components permits the framework to support different event sources related to Java software, such as Java VMs and static analysers. The framework design may also host components capable of monitoring events generated from applications developed using other technologies. Along with monitors, the framework design caters for multiple evaluator components for evaluating conformance rules against different classes of events. By coupling monitors together with evaluators, the design facilitates the use of different event types such as runtime events, static events and events from multiple VMs to perform conformance checking. Yet another aspect of extensibility is being able to support different architecture modelling notations, which is discussed in Section 7.1.5. However, although its design supports considerable extensibility, the implemented framework has not yet realised this principle in full. It has only a single monitor for catching Java runtime events, a single evaluator for checking conformance rules, and a single rule generator component for processing Grasp specifications. Therefore, PANDArch should implement a few more monitors, evaluators and rule generators before the extensibility objective for the framework as whole can be considered realised.
7.2 Limitations and Weaknesses

The following sub-sections discuss some of the limitations and weaknesses in both the overall strategy and the current implementation of the framework.

7.2.1 Concerns Related to Specifying Architectures

A fundamental aspect of the PANDArch approach is that its driver is a specification of the software architecture. This is expected because for any technique that checks architecture conformance, an architecture specification is the starting point. However, there are significant variations among languages, notations, and techniques employed for modelling software architectures. There also is the case where in practice architectures are often described informally and in diagrammatic form. Therefore, in these conditions, an immediate first concern is whether PANDArch is too tightly coupled with Grasp for the framework to be adopted in non-trivial software development. This concern is addressed earlier in the discussion related to achieving notation independence. While PANDArch is notation agnostic by design, the implemented framework can be considered coupled with the conceptual component-connector modelling paradigm adopted in Grasp. Modelling software architectures using discrete components connected through interfaces is a popular technique. Furthermore, the extracted architectures of the three open source systems in the earlier case studies show that component-oriented designs can be easily, if not loosely, implemented by a mainstream language like Java. However, other architecture paradigms such as service-orientation [163] are also popular in the industry, and being a generic conformance monitoring framework, PANDArch should extend support for these paradigms as well.

At the very foundation of the PANDArch approach are conformance rules. An important quality of conformance rules is that, although they are derived from some architecture property, the rules do not contain any architecturally significant information. A conformance rule specifies some constraint, dynamic or otherwise, that should hold only in the implementation, rather that in the architecture. The framework generates conformance rules for a number of architecture properties, but once created, the rules are independent of the architecture (except that they may refer back to corresponding architecture elements for error reporting purposes). As a consequence of this detachment, it should be possible to generate conformance rules for any architecture property (or concept), provided that such a property can be implemented by a programmer. Furthermore, new conformance rules can be introduced to support additional architecture properties. Therefore, it can be reasonably assumed that the design of PANDArch is not only independent of architecture modelling notations, but it is also independent of architecture paradigms. The conformance checker, which operates only on conformance rules, expects a rule generator capable of processing some architecture specification to extract the appropriate
conformance rules from that specification. The conformance verification process from that point onwards remains unchanged. However, until the framework implementation has been extended with these capabilities and sufficiently evaluated, this assumption is treated as conjecture.

Another challenge related to describing architectures is its lack of prevalence. As argued in Chapter 2, ADLs and other specification techniques are not well adopted in the industry [147]. One possible reason for this is the precision and the completeness required, when only a few architectural concepts are practised in the industry [147]. Furthermore, contemporary development methodologies such as agile techniques employ very little high-level design, and the design is continually refined as development progresses. In this context, PANDArch has the potential to motivate the adoption of architecture specifications as they could prove useful in conformance verification. Not requiring a full description of the architecture or accepting partial specifications could also make the framework useful for agile software projects.

### 7.2.2 Concerns Related to Implementation Mappings

Besides specifying the architecture of the target application, mapping elements in the specified architecture to programming constructs in the implementation is a crucial aspect of PANDArch. Without implementation mappings the framework has no means of checking whether the implemented software conforms to what is prescribed in the architecture. With Grasp, annotations provide the means for associating various mapping configurations with elements in the architecture. This approach complies with the principle of being specification-driven, where mapping information is required to be part of the architecture specification. However, embedding implementation mappings may not always be possible with other ADLs or architecture modelling techniques. They may not have a capability similar to Grasp annotations or other means to link architectural constructs to program code. Therefore, even if PANDArch offers an extensible design to deal with architectures modelled with notations other than Grasp, implementing this capability is likely to be challenging in some ADLs. A possible solution in such cases is to use another artefact to hold the mapping details needed for bridging the architecture and the implementation. However, doing so might require devising a new notation.

A shortcoming in the way PANDArch currently deals with implementation mappings is that it views the entire implementation as a logical entity. As a result, architectural concepts cannot be mapped to physical elements of the implementation. For instance, mapping a component in the architecture to a Java JAR file that implements the component is currently not possible. Instead the component in this case needs to be mapped to every class packaged within the JAR file. However, doing so may restrict the conformance checker from verifying the physical deployment constraints prescribed in the architecture, as information related to their physical deployment are
not available in Java classes. For instance, a Java class does not carry the name of the JAR file in which it has been packaged. Since verifying whether an implementation satisfies the physical constraints specified in the intended architecture is also important, the framework should be extended to incorporate this capability.

From a practical standpoint, embedding implementation mappings within an architecture specification could sometimes lead to readability and maintainability problems in the model. These issues could be particularly adverse with textual ADLs such as Grasp. Noise created by mapping descriptions can make the model harder to read and comprehend. An obvious solution to address this problem is to use modelling tools or special editors that filter out mappings in the specification when required. However, a larger related concern is the effort required for defining and maintaining implementation mappings. Updating these mappings has the potential to impose a considerable burden, particularly during the early stages of development when programmers develop code at a fast pace. The overhead required to keep the architecture mapped to the implementation under these conditions may discourage the use of PANDArch, and therefore remains a weakness of this approach.

7.3 Comparisons to Other Similar Approaches

A number of approaches for examining software architectures at runtime can be found in literature. This section compares the PANDArch approach with some of those approaches. Although some of the approaches are already featured in Chapter 2, the emphasis of the following discussion is comparing their techniques with that of PANDArch.

DiscoTect [62] is a technique for discovering software architectures from executing systems. It uses runtime events, state information and rules for known architectural styles to discover the architecture of an running Java program. A mapping language called DiscoSTEP is employed to bridge the abstraction gap between architecture and implementation. The primary focus of DiscoTect is runtime architecture discovery, and conformance is checked manually. In contrast, PANDArch automatically checks rules extracted from an architecture against the implementation.

An adaptation of DiscoTect has been developed by Ganesan et al. [64]. This approach collects runtime traces from applications with the help of probes that are inserted using aspect weaving techniques. It replaces DiscoSTEP specifications with Coloured Petri nets to link architecture to implementation and extracts sequence diagrams from the system in order to check behavioural compliance, which is not supported by DiscoTect. This technique is pluggable, non-intrusive and some aspects of the discovered architecture can be verified automatically. However, the required probes have to be defined manually, while mappings are maintained separate from
the architecture specification and properties corresponding to specific architectural styles must be manually pre-configured in the checker. The approach also appears tightly coupled to the OSGi framework. On the other hand, PANDArch uses a single architecture specification with integrated mappings from which rules for checking architecture conformance are automatically derived. The framework is also largely capable of monitoring most types of Java software.

Popescu and Medvidovic [164] suggest combining mappings and code injection techniques in a semi-automatic approach for checking behavioural compliance of an event-based software implementation with its architecture. This approach injects probes and recorders into components, extracts and filters runtime event data, and then compares them to prescriptive sequences of events. It focuses on event-based systems. Rules for causality and matching are defined separately by the architect, and some human interpretation is required to verify conformance.

The SAVE tool [165] is capable of checking conformance of software implemented in Java, C, C++ or Delphi with their corresponding structural and behavioural architecture views. For checking structural conformance, components in the architecture are manually mapped to source code constructs after which the code is statically analysed to discover points of disparity. Behavioural conformance checking is done by comparing runtime traces against sequence diagrams prescribing the intended behaviour of the system. However, behavioural checks are performed offline, i.e. when the application is not executing, whereas with PANDArch runtime conformance monitoring is a runtime activity.
8.1 Summary

Software architectures form the crucial foundation upon which non-trivial software system are built. Architecture is the primary outcome of the process where important system requirements are transformed into a technical design, therefore a cornerstone for the successful delivery, functional performance and maintenance of software. Nonetheless, architecture often remains an invisible or intangible entity within the implemented software, as mainstream programming languages are unable to represent architectural concepts. Due to this invisibility and a number of other reasons, an evolving software system has the tendency to deviate from its intended architecture. Architecture erosion has the likelihood to severely affect the performance of the software, cripple its functionality, not deliver on the requirements and force the software into premature retirement.

The background survey in Chapter 2 of the thesis investigates the numerous attempts made to address architecture erosion. These approaches are placed in a classification framework for ease of comparing the different classes of strategies. While each strategy has its own benefits and challenges, a common factor that becomes apparent from the survey is the adoption of these techniques is not as prevalent as the problem itself. The chapter discusses a number of possible contributing factors for this lack of take up.

The research in the thesis investigates the possibility of incorporating runtime conformance monitoring as part of an overall strategy aimed at controlling architecture erosion. The thesis defines architecture conformance monitoring as the process of verifying conformance of both static and dynamic properties of a software implementation against the properties of the architecture. The proposed approach focuses on checking conformance at runtime, i.e. when the subject software is executing. The approach is guided by a number of core principles, which
themselves are motivated by the shortcomings seen as reasons behind the low adoption of other approaches. Of these principles, those that stress the need for detachability, non-intrusiveness, automation and a specification-driven approach are brought together to form a notion of an architecture environment capable of moulding the software implementation. This notion forms the conceptual basis for developing the PANDArch conformance monitoring framework.

The process of conformance monitoring begins with the specification of the architecture. In this work, the Grasp ADL provides means for describing architectures along with implementation mappings within a single artefact. An essential aspect of conformance monitoring, implementation mappings correlate elements in the architecture to code constructs implementing those architecture elements. Implementation mapping strategies, however, vary among different approaches, and PANDArch adopts a flexible technique that tags the architecture with names of code constructs. The framework parses architecture properties and mappings in the Grasp model to generate a number of conformance rules describing properties in the implementation that must hold. These conformance rules form the basis for checking architecture conformance.

The design of PANDArch relies on runtime events to gather details about the target software implementation. Each conformance rule extracted from the architecture is associated with a specific event that provides the most relevant information for evaluating that rule. Most conformance rules are evaluated as and when events are received, while some rules are evaluated after the target software terminates execution. Besides supporting runtime events of Java applications, the design can be extended to accept events from other sources, both static and dynamic. Furthermore, a number of design strategies are put in place to minimise the performance impact on the software being monitored. In the current Java-based implementation of the framework, two mechanisms are provided for listening to runtime events of other Java software. The mechanism that relies on the JDI captures larger number of event types though its performance impact is extremely high. Alternatively, instrumentation probes detect a smaller subset of Java runtime events while keeping the performance cost considerably lower.

In this thesis the PANDArch framework is evaluated both for the effectiveness of its conformance checking capabilities and its performance impact on the application being monitored. Each conformance rule is first validated using a number of unit tests to ensure that it produces the expected results. Thereafter, the framework as a whole is evaluated with the help of three popular Java open source products of different types, namely Jackrabbit, jEdit and Hibernate. Due to insufficient architecture documentation, the implemented architectures of these software are extracted from the compiled code. Conformance is broken by both altering the code and making changes to the Grasp model of the architecture. In either case, PANDArch is able to detect the induced conformance violations at runtime. Performance testing is included in the
8.2 Future Work

The design and implementation PANDArch framework presented in this thesis can be expanded in a number of useful ways. One such extension is the support for different architectural views and their associated conformance rules. A limitation in the current framework implementation is that it can only use runtime structural views for checking architecture conformance. While this is mostly because of a similar limitation in Grasp, the framework itself needs to be extended with conformance rules related to other types of architectural views used commonly in practice, such as the physical view. Such a capability should be developed in conjunction with support for extracting architecture properties from an ADL that supports these views. A useful notation to support in this regard is UML, as industry surveys show significant use of UML for describing certain architectural views [147].

Incorporating functionality to process multiple architectural views requires validating the consistency among these views. Although automated consistency checking of architectural models using formal methods [166] is somewhat different to architecture conformance checking, it can be considered a necessary pre-processing step for ensuring there are no discrepancies among views given to PANDArch. For instance, a runtime view and a physical view of an architecture should convey the same component interactions, failing which the conformance checker could produce conflicting notifications during conformance monitoring. While existing approaches may be used for consistency verification, they should be made an inherent part of the framework so that the automation principle is not violated.

Besides providing the capacity to deal with multiple architectural views, PANDArch should also be capable of supporting varying configurations of software systems. Extending the framework to check conformance of distributed architectures is considered a priority in this regard, as a significant portion of contemporary software are considered to be distributed systems.
Another important area of future work is building a static conformance checking capability to supplement dynamic conformance monitoring available at present. Checking conformance by statically analysing the implementation is a fundamental aspect of the PANDArch approach as discussed in Chapter 3. It can be carried out during the development phase, even before the software is ready to be tested. Although some static architecture properties can be checked for conformance during program execution, certain others such as references to external libraries cannot be checked at runtime. These external references are often lost during the build process when the program is statically linked to those libraries. Therefore, the inclusion of static conformance checking will allow PANDArch to increase the coverage of detectable architecture violations. Static conformance checking may also contribute towards minimising the performance impact by limiting runtime conformance checking to only dynamic architecture properties.

The approach taken for evaluating PANDArch can be further improved as part of future work. The strategy presented in this thesis for evaluating the framework with open source software involved extracting the implemented architecture of those systems followed by artificially inducing architecture violations. This was required due to the lack of accurate and up to date architectures of the chose open source products. While such an approach is effective in evaluating whether the framework performs as expected from a functional point of view, it does not represent its use case in practice. Therefore, in order to measure its effectiveness in detecting architecture violations PANDArch needs to be deployed within an on going software development project. The framework should use the existing software architecture to analyse the implementation for conformance violations while conformance checking is made an integral activity of the development process. A useful extension to the framework prior to carrying out this work is enabling static conformance checking as discussed in the previous paragraph.

An important area of further exploration is runtime conformance checking of applications that execute native code outwith a virtual machine. Applications written in languages such C or C++ fall under this category and are in widespread use. A fundamental premise in the PANDArch approach is that an executing application is able to provide some information about changing state in the form of events. However, applications that execute natively do not meet this premise because they are hosted by an operating system that does not track the internal state of the application to the extent done by a virtual machine. Therefore, an elaborate mechanism needs to be devised to extract runtime information from executing native code. Some compilers allow embedding runtime-type information, which can serve as a starting point, although retrieving that information during execution may require injecting probes into the compiled code. However, as PANDArch will remain pluggable, the negative consequences runtime-type information and code injection can be contained to the duration when conformance is monitored.
8.3 Final Thoughts

The thesis presents a novel approach for verifying architecture conformance that could be incorporated as part of a holistic solution for controlling software architecture erosion. The pluggable, automated, non-intrusive and dynamic framework developed for this purpose takes the path of runtime conformance checking to identify architecture violations in the implementation. Evaluation of the framework with nontrivial software demonstrates that the strategy is indeed effective though some limitations exist. However, runtime conformance checking alone may not detect every point of architectural non-conformance in the code. While guaranteeing sufficient code coverage for runtime conformance checking itself is a challenging task, verifying the conformance of certain architecture properties may require static analysis of the source code. Similarly, a distributed application may have certain architecture constraints that apply to interactions among its nodes, and to check whether these constraints hold, the framework needs to collect events from multiple virtual machines. Although PANDArch can be extended to support these scenarios, such features are not yet implemented. But they will most likely be needed if the framework is to be promoted as a comprehensive architecture conformance monitor.

An essential prerequisite for any approach that attempts to control architecture erosion is the availability of a specification of the intended architecture. Without an architecture, there is no other benchmark against which the implementation can be validated. As experienced with the three open source systems deployed in the evaluation experiments, documented architectures are often rare and unreliable in the software engineering world. Therefore, while researchers continue to spend considerable effort exploring avenues that effectively deal with architecture erosion, their work is bound to be hindered by the very socio-technical issue of unwillingness to document architectures on the part of those who design software. In this regard, finding a practical solution for improving architecture specification should be an intrinsic part of the larger strategy for addressing architecture erosion.
A comprehensive reference of Grasp architecture description language is given in this appendix. Grasp is the choice of ADL for specifying architectures of applications used for evaluating the PANDArch framework described in this thesis.

### A.1 Introduction

Grasp is a textual ADL that follows the popular component and connector paradigm [5] for modelling architectures. In addition to structural properties, Grasp also supports specifying rationale as part of an architecture description. Rationale is an intrinsic aspect of software architecture, representing various alternatives, trade-offs, assumptions, constraints and others factors considered during its design [4]. However, unlike other architectural features, rationale often remains unspecified and inaccessible to tools. Grasp employs a simple model for capturing rationales as part of an architecture specification and attaching them to elements in the architecture. Both formal expressions and natural language can be used for specifying rationale. The bidirectional links between rationales and elements enable forward and backward traceability, two forms of analysis useful in evaluating the robustness of the architectural design.

This reference first presents the language constructs of Grasp, followed by a discussion of the underlying concepts and details related to modelling architectures with Grasp. The appendix concludes with the presentation of a formal grammar for the language.
A.2 Language Basics

A.2.1 Specification Structure

The simplest Grasp architecture specification consists of an architecture statement and a system statement. The architecture statement specifies the entire architecture within which the system statement declares the runtime structure of the architecture. A Grasp source is a text file that contains a single architecture specification and only one system declaration is allowed within that architecture. By convention the name of a source file has the “.grasp” extension, however, this is not mandatory.

```plaintext
architecture HelloWorld {
    system HelloWorldSystem {
    }
}
```

Besides the mandatory system statement, a Grasp specification may contain any number of requirement, quality_attribute, rationale and template statements within the body of the architecture statement. These are all top-level statements in the architecture and in-turn they contain other Grasp statements. The following example describes a simple architecture consisting of system, rationale and template elements.

```plaintext
architecture HelloWorld {
    rationale HelloWorldRationale() {
    }
    template HelloWorldTemplate() {
    }
    system HelloWorldSystem {
    }
}
```

A.2.2 Comments

Grasp supports two types of comments similar to those found in languages such as Java or C++. An inline comment is preceded with the character pair ‘//’ and extends until end of line. A block comment starts with the character pair ‘/*’ and continues until a matching ‘*/’ pair is found.
A.2.3 Case Sensitivity

Grasp is a case-sensitive language. Therefore, keywords and identifiers that are alphabetically identical but vary only by case are treated differently. For example, `link` is a reserved keyword in Grasp and cannot be used as an identifier. However, `Link` is a valid identifier. So are `LINK`, `linK` and `LInk`, but all distinct from each other.

A.2.4 Statements

A statement in Grasp is a declaration of a first-class architecture element or object. A simple Grasp statement is terminated by the semicolon character (‘;’). A block statement (also known as a compound statement) is one where the declaration is followed by a list of additional statements enclosed within a pair of opening and closing braces (‘{…}’). A block statement defines the body of an element. Therefore, a statements that uses the semicolon form or a block statement with no nested statements declares an element with an empty body. Grasp objects that do not allownesting of other elements support only the simple form of declaration. Those that allow nesting support both forms, i.e. simple and block statements, though most of these elements may individually restrict what can be included in the body. A formal description of these restrictions can be found in the Grasp grammar specification detailed in Section A.5.

A block statement in Grasp (or the body of an element) also defines a scope. A scope is a namespace consisting of unique names. Therefore, two or more elements declared within the same scope cannot share a name. However, elements in different scopes are allowed to have the same name. The code snippet below shows examples of both simple and block statements.

```plaintext
architecture Simple { // block statement
  template SimpleComponent() {} // block statement-empty body
  system SimpleSystem {
    component compA = SimpleComponent(); // block statement
    component compB = SimpleComponent() {
      check true; // simple statement
    }
  }
}
```

A.2.5 Identifiers

Identifiers in Grasp consist of a sequence of characters of which the first character must be an alphabetical character or the underscore. The rest of the characters can be any combination of alphabetical, numeric or underscore characters. Reserved keywords of the language cannot be used as identifiers. An identifier can be of arbitrary length.
A.2.6 Keywords

The following keywords are currently used in Grasp. These keywords are reserved by the Grasp compiler and therefore, cannot be used as regular identifiers.

<table>
<thead>
<tr>
<th>architecture</th>
<th>because</th>
<th>check</th>
<th>component</th>
<th>connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>extends</td>
<td>false</td>
<td>in</td>
<td>inhibits</td>
<td>layer</td>
</tr>
<tr>
<td>link</td>
<td>over</td>
<td>out</td>
<td>provides</td>
<td>property</td>
</tr>
<tr>
<td>quality_attribute</td>
<td>rationale</td>
<td>reason</td>
<td>requirement</td>
<td>requires</td>
</tr>
<tr>
<td>subsetof</td>
<td>supports</td>
<td>system</td>
<td>template</td>
<td>to</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Reserved keywords in Grasp ADL.

A.2.7 Types and Values

Grasp provides a simple type system consisting four basic types and one collection type. These are described in the following subsections.

A.2.7.1 Integer Type

The integer data type is represented as a 64-bit signed two’s complement integer value in the range of \(-2^{63}\) to \(2^{63} - 1\). An integer literal is expressed as a sequence of decimal digits. The sequence may precede with an optional plus (‘+’) or minus (‘-’) character denoting sign.

A.2.7.2 Real Type

The real data type is represented in the 64-bit double-precision IEEE 754 floating point format. The precise range of a value represented by the real type is dependent on a specific implementation of the Grasp compiler. A real literal is expressed as a sequence of decimal digits and a single
A.2. LANGUAGE BASICS

period (‘.’) character. The sequence may precede with an optional plus (‘+’) or minus (‘-’) character denoting sign.

A.2.7.3 Boolean Type

The boolean data type specifies a logical true or false value. Boolean literals are specified using the Grasp true and false keywords.

A.2.7.4 String Type

The string type represents a sequence of characters. A string literal in Grasp is a sequence of characters enclosed within a matching pair of double quotes or single quotes.

A.2.7.5 Set Type

The set type represents a collection of unique values of any of the four basic types explained above. A set literal is expressed as a list of integer, real, boolean or string literals contained within a pair of ‘[‘ and ‘]’ characters. Items in the list are separated by comma (‘,’) characters.

The following code snippet shows examples of the different types in Grasp.

```plaintext
architecture Simple
{
  quality_attribute qaInteger = 25;
  quality_attribute qaReal= 0.32;
  quality_attribute qaString = "Gui guidelines";
  quality_attribute qaAnotherString = 'Performance evaluation';
  quality_attribute qaBoolean = true;
  quality_attribute qaSet = ["Ok", false, -10.52, 1024, 'False'];
  SimpleSystem {}
}
```

A.2.8 Expressions

Grasp expressions are used for partly describing properties, constraints and rationale in architecture specifications. An expression may include any of the operators offered by the language. These operators are described in the following table in the descending order of precedence.
A few examples of Grasp expressions are demonstrated below.

```plaintext
architecture Simple {
  // QAs
  quality_attribute qa1 = 25 * 64;
  quality_attribute qa2 = true && false;
  quality_attribute qa3 = "Gui guidelines" + " other guidelines";

  // Template
  template BasicComponent(other) {
    check other.properties() subsetof ["Ok", "OK", "ok", "oK"]; 
    check other.connected() == true;
  }
  SimpleSystem {};
}
```
A.3 The Conceptual Model

An intrinsic aspect of Grasp is modelling rationale as part of an architecture specification. This section explains the steps involved in describing an architecture with rationale using an example. The conceptual model behind Grasp consists of three primary entities. They are architecture element (AE), rationale and reason. An AE has zero or more associated rationales justifying its purpose. A rationale is a conjunction of one or more reasons. A reason is a logical expression that evaluates to a Boolean outcome. If a reason in a rationale evaluates to false, then every dependent AE is considered to have failed its rationale. Any reason expressed in natural language alone is treated as a logically true statement. The diagram in Figure A.1 formalises this model in UML.

Figure A.1: The Grasp conceptual model formalised in UML.

A rationale and an AE have a bi-directional association with each other. A rationale motivates the existence or behaviour of one or more AEs. Conversely, an AE can be attached to zero or more rationale entities. Since an AE is justified by a rationale associated with it, the term because is used to brand this association. As a rationale can cross-cut many AEs, the model caters for this possibility by allowing a single rationale to be associated with multiple AEs.

An AE \( X \) can also act as a reason for the rationale of another AE \( Y \) since \( X \) can cause or require the existence of \( Y \) in the same architecture. However, direct or indirect cyclic references are not permitted. Thus, relationships between rationales and AEs form an acyclic graph similar to that of AREL. The \{unique\} constraint applied to the AE entity ensures that each AE instance is unique and therefore does not become a reason for its own rationale.

A rationale may extend another, effectively inheriting its collection of reasons. However, multiple-inheritance is not permitted in the interest of simplicity and clarity.

This model also allows associating quality attributes (QAs) as motivating reasons for rationales.
A QA is a constraint that should hold when a system implements and delivers its services. Typical drivers for QAs are non-functional requirements. QAs that can be described quantitatively are specified as a collection of properties in this model. Since design choices that support certain QAs may negatively impact others, the association between reasons and QAs are twofold. A reason may support one or more QAs. A reason may also hinder one or more QAs while in the process of justifying some design decision.

Lastly, reasons and QAs may be attributed to requirements, which in this model are simply references to an external requirement specification.

### A.4 Describing Architectures with Grasp

To demonstrate architecture specification in Grasp, a case study of an electronic fund transfer system (EFT) presented with the AREL approach is reused as an example here. The outcome of applying Grasp to a portion of this example architecture, namely the asynchronous messaging subsystem, is shown in Listing A.1.

```plaintext
architecture Example {
  // Requirements and quality attribute specifiers
  requirement Rq_AckProcessing;

  quality_attribute MultiConnections = true;
  quality_attribute MaxVolumePerBank = 50;
  quality_attribute MinTrxPerDay = 8000;
  quality_attribute NoLostPayments = true;
  quality_attribute NoDuplicates = true;

  // Rationale descriptors
  @(desc="OptimalMsgProcPerformance")
  rationale AR10() {
    reason supports MultiConnections, MaxVolumePerBank, MinTrxPerDay;
    reason supports Rq_AckProcessing;
  }

  @(desc="ProcessingSequence")
  rationale AR13(M) {
    reason ["Async"] subsetof M.properties();
  }

  @(desc="NoLossTransaction")
  rationale AR14(M) {
    reason supports NoLostPayments, NoDuplicates;
    reason ["Async"] subsetof M.properties();
  }

  @(desc="TimeOutMechanism")
  rationale AR15(M) {
    reason ["Async"] subsetof M.properties();
  }
}
```
The EFT system is designed to execute high-value online fund transfers between local banks and a central bank. A key requirement for the messaging subsystem is performance and hence, the designers chose to use an asynchronous messaging strategy to achieve this critical quality attribute. Subsequent design decisions and outcomes were directly or indirectly influenced by the decision to build an architecture that supports asynchronous messaging. The next few subsections discuss different aspects of a Grasp model with the help of this example.

### A.4.1 Modelling Rationale

The Grasp description of the messaging subsystem begins with explicit declarations of QAs and references to external requirements. The reference Rq_AckProcessing points to an existing requirement which states that all messages should be acknowledged. The Grasp code also define five QAs related to performance and reliability qualities that should be exhibited by the messaging subsystem. Once defined, these QAs and the requirement become motivating reasons for describing rationale.

The Grasp model declares four rationales named AR10, AR13, AR14 and AR15, each with its own set of reasons. Each rationale is tagged with a descriptive name using the annotation feature in Grasp. Rationale AR10 has two reasons behind it: to achieve qualities MultiConnections, MaxVolumePerBank and MinTrxPerDay, and to satisfy the requirement Rq_AckProcessing. The keyword supports relates reasons to QAs and requirements. The reason for rationale AR13, on the other hand, is an expression that references parameter M representing an AE given to AR13 at invocation. This expression checks whether the set of properties of M includes property Async using the subsetof operator. The remaining two rationales are declared in a similar manner.
A.4.2 Modelling System Structure

For specifying the static runtime structure of an architecture, Grasp follows the popular components and connectors paradigm. Along with these two primitives, Grasp supports layer, interface, link and check elements as its primary building blocks. An abstract reusable construct called template helps to define composite structures from which components and connectors are instantiated.

The chosen example consists of only components, all of which are instantiated from one of two templates, namely AsyncComponent and AlarmService. The asyncMarker property defined in AsyncComponent identifies all instantiating components as part of the design dealing with asynchronous communication aspects.

The system block in the Grasp specification contains a number of component statements instantiating components from templates. However, the interconnections (i.e. wirings) between these components are not included as this information is missing in the original case study. Grasp provides a link primitive to specify the wiring among components (as well as connectors) through their interfaces.

A.4.3 Binding Rationale to Architecture Elements

The next step in building a Grasp specification is to associate rationale to various elements in the architecture. As shown in Listing A.1, each component statement has a because clause that attaches a previously declared rationale to that component. Some rationales in this example accept arguments that refer to other components within the same namespace. A rationale can be attached in a similar manner to any type of AE.

This example also demonstrates the dependency chain that forms with AEs and rationales in a Grasp model. For instance, the alarm component is bound to rationale AR15, which in turn is tied to errDetect which is passed as an argument to AR15. In turn errDetect is associated with AR14, which depends on msgPro, which depends on AR10.

A.4.4 Evaluating Rationale

A rationale is evaluated in the context of the AE to which it is attached. The application context of a rationale is an important aspect in this model. A rationale cannot be evaluated as a free-standing entity and therefore must be associated with at least one AE for this purpose. At the same time, not all AEs associated with a given rationale may satisfy that rationale. Hence, a rationale is evaluated within the context of each associated AE, and independent of other affiliated AEs.
In the given example, rationale AR10 is tested within the context of msgProc, AR13 within the context of mcpDriver, AR14 within both errDetect and errRecover, and AR15 within alarm. In the case of rationale AR14, which attaches to two components, it may possibly pass with one component and fail with the other.

### A.5 Grammar

Grasp grammar rules are described below in the Extended Backus–Naur Form (EBNF) notation.

\[
\text{⟨grasp_specification⟩} ::= \text{⟨architecture_statement⟩}
\]

\[
\text{⟨architecture_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘architecture’} \text{⟨identifier⟩} \{ \text{⟨architecture_body⟩} \}^\star
\]

\[
\text{⟨architecture_body⟩} ::= \text{⟨declaration_statement⟩}^\star \text{⟨system_statement⟩} \text{⟨declaration_statement⟩}^\star
\]

\[
\text{⟨declaration_statement⟩} ::= \text{⟨requirement_statement⟩} | \text{⟨qattribute_statement⟩} | \text{⟨rationale_statement⟩} | \text{⟨template_statement⟩}
\]

\[
\text{⟨requirement_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘requirement’} \text{⟨identifier⟩} \{ \text{‘=’} \text{⟨string_literal⟩} \}? \text{‘;’}
\]

\[
\text{⟨qattribute_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘quality_attribute’} \text{⟨identifier⟩} \{ \text{‘=’} \text{⟨expression⟩} \}? \text{⟨supports_clause⟩}? \text{‘;’}
\]

\[
\text{⟨rationale_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘rationale’} \text{⟨identifier⟩} \{ \text{⟨parameter_list⟩}? \} \text{⟨extends_clause⟩}? \{ \text{⟨reason_statement⟩} \}^\star \text{‘{’} \text{⟨reason_statement⟩}^\star \text{‘} \}
\]

\[
\text{⟨reason_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘reason’} \{ \text{⟨expression⟩} \} \text{⟨supports_clause⟩} \text{⟨inhibits_clause⟩}? \text{‘;’}
\]

\[
\text{⟨template_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘template’} \text{⟨identifier⟩} \{ : \text{⟨integer_literal⟩} \}? \text{‘{’} \text{⟨template_constituent⟩}^\star \text{‘} \}
\]

\[
\text{⟨template_constituent⟩} ::= \text{⟨template_statement⟩} | \text{⟨provides_statement⟩} | \text{⟨requires_statement⟩} | \text{⟨property_statement⟩} | \text{⟨check_statement⟩}
\]

\[
\text{⟨provides_statement⟩} ::= \text{⟨annotation⟩}^\star \text{‘provides’} \text{⟨identifier⟩} \{ : \text{⟨integer_literal⟩} \}? \text{⟨identifier⟩}? \text{⟨because_clause⟩}? \{ \text{‘{’} \text{⟨provides_constituent⟩}^\star \text{‘} \}
\]

\[
\text{⟨provides_constituent⟩} ::= \text{⟨check_statement⟩}
\]
\[\text{requires}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘requires’} \langle\text{identifier}\rangle \langle\text{identifier}\rangle^? \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{requires}_\text{constituent}\rangle \text{‘Y’} \text{‘;}’)
\]

\[\text{requires}_\text{constituent} \quad ::= \quad \langle\text{check}_\text{statement}\rangle
\]

\[\text{property}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘property’} \langle\text{identifier}\rangle (’=’ \langle\text{expression}\rangle)? \langle\text{because}_\text{clause}\rangle^? \text{‘;’}
\]

\[\text{check}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘check’} \langle\text{expression}\rangle \langle\text{because}_\text{clause}\rangle^? \text{‘;’}
\]

\[\text{system}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘system’} \langle\text{identifier}\rangle \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{system}_\text{constituent}\rangle \text{‘Y’}
\]

\[\text{system}_\text{constituent} \quad ::= \quad \langle\text{compositional}_\text{statement}\rangle
\]

\[\text{compositional}_\text{statement} \quad ::= \quad \langle\text{layer}_\text{statement}\rangle
\]

\[\text{layer}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘layer’} \langle\text{identifier}\rangle \langle\text{layer}_\text{over}\rangle^? \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{layer}_\text{constituent}\rangle \text{‘Y’} \text{‘;’})
\]

\[\text{layer}_\text{over} \quad ::= \quad \text{‘over’} \langle\text{identifier}\rangle (’,’ \langle\text{identifier}\rangle^*)
\]

\[\text{layer}_\text{constituent} \quad ::= \quad \langle\text{compositional}_\text{statement}\rangle
\]

\[\text{component}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘component’} \langle\text{identifier}\rangle ‘=’ \langle\text{identifier}\rangle
\]

\[\langle\text{argument}_\text{list}\rangle? \text{‘Y’} \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{instantiable}_\text{constituent}\rangle \text{‘Y’} \text{‘;’})
\]

\[\text{connector}_\text{statement} \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘connector’} \langle\text{identifier}\rangle ‘=’ \langle\text{identifier}\rangle
\]

\[\langle\text{argument}_\text{list}\rangle? \text{‘Y’} \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{instantiable}_\text{constituent}\rangle \text{‘Y’} \text{‘;’})
\]

\[\text{instantiable}_\text{constituent} \quad ::= \quad \langle\text{compositional}_\text{statement}\rangle
\]

\[\langle\text{link}_\text{statement}\rangle \quad ::= \quad \langle\text{annotation}\rangle^\ast \text{‘link’} \langle\text{identifier}\rangle^? \langle\text{link}_\text{consumer}\rangle \text{‘to’} \langle\text{link}_\text{providers}\rangle \langle\text{because}_\text{clause}\rangle^? \quad (‘t’ \langle\text{link}_\text{constituent}\rangle \text{‘Y’} \text{‘;’})
\]

\[\text{link}_\text{consumer} \quad ::= \quad \langle\text{member}_\text{expression}\rangle
\]

\[\text{link}_\text{providers} \quad ::= \quad \langle\text{member}_\text{expression}\rangle (’,’ \langle\text{member}_\text{expression}\rangle)^*
\]

\[\text{annotation} \quad ::= \quad \text{‘@’} \langle\text{identifier}\rangle^? \quad (‘t’ \langle\text{namevalue}\rangle (’,’ \langle\text{namevalue}\rangle)^* \text{‘;’})
\]

\[\text{supports}_\text{clause} \quad ::= \quad \text{‘supports’} \langle\text{identifier}\rangle (’,’ \langle\text{identifier}\rangle)^*
\]

\[\text{inhibits}_\text{clause} \quad ::= \quad \text{‘inhibits’} \langle\text{identifier}\rangle (’,’ \langle\text{identifier}\rangle)^*
\]
A.5. GRAMMAR

(extends_clause) ::= 'extends' (identifier)

(because_clause) ::= 'because' (because_causalcontext) (',' (because_causalcontext))*

(because_causalcontext) ::= (identifier) ('(' (argument_list) '?')'

(parameter_list) ::= (identifier) (',' (identifier))*

(argument_list) ::= (member_expression) (',' (member_expression))*

(namevalue) ::= (identifier) '={' (expression)

(expression) ::= ((alpha_character) '|' '.') ((alpha_character) (decimal_digit) '|'('.'))*

(subsetof_expression) ::= (logicalOr_expression) ('subsetof' (logicalOr_expression))*

(logicalOr_expression) ::= (logicalAnd_expression) ('||' (logicalAnd_expression))*

(logicalAnd_expression) ::= (bitwiseAnd_expression) ('&&' (bitwiseAnd_expression))*

(bitwiseAnd_expression) ::= (bitwiseXor_expression) ('!' (bitwiseXor_expression))*

(bitwiseXor_expression) ::= (bitwiseOr_expression) ('~' (bitwiseOr_expression))*

(bitwiseOr_expression) ::= (bitwiseOr_expression) ('|' (bitwiseOr_expression))*

(bitwiseOr_expression) ::= (bitwiseXor_expression) ('|' (bitwiseXor_expression))*

(bitwiseXor_expression) ::= (bitwiseAnd_expression) ('&' (bitwiseAnd_expression))*

(bitwiseAnd_expression) ::= (equality_expression) ('&' (equality_expression))*

(equality_expression) ::= (relational_expression) ('==','!==' (relational_expression))*

(relational_expression) ::= (additive_expression) ('>' ' '<' ' >=' ' <=' (additive_expression))*

(additive_expression) ::= (multiplicative_expression) ('+' '-' (multiplicative_expression))*

(multiplicative_expression) ::= (unary_expression) ('*'/ '/' '%%' (unary_expression))*

(unary_expression) ::= ('!' ' ~' '+' '-' (unary_expression)

(primary_expression) ::= ('(' (expression) ')

(member_expression) ::= (member_part) ('.' (member_part))*

(member_part) ::= (identifier) (identifier) ('(' (member_arguments) ')') 'in' 'out'

(member_arguments) ::= (expression) (',' (expression))*

(literal) ::= (set_literal) (integer_literal) (real_literal) (boolean_literal) (string_literal)

(set_literal) ::= ['[' (literal) (',','literal')]* ']

(integer_literal) ::= (decimal_digit) '+' (hex_digit)'

(real_literal) ::= (decimal_digit) '*' (decimal_digit)
\langle boolean_literal \rangle ::= '#' \langle singlequote_text \rangle ' ' '#' \langle doublequote_text \rangle ' ' | 'true' | 'false'

\langle string_literal \rangle ::= ' ' \langle singlequote_text \rangle ' ' | ' ' \langle doublequote_text \rangle ' ' 

\langle singlequote_text \rangle ::= (! (' ' | ' ' | \langle newline_character \rangle ) | \langle escape_sequence \rangle ) *

\langle doublequote_text \rangle ::= (! (' ' | ' ' | \langle newline_character \rangle ) | \langle escape_sequence \rangle ) *

\langle escape_sequence \rangle ::= ' \backslash ' ( 't' | 'b' | 'n' | 'r' | 'f' | ' ' | ' ' | '\u' \langle hex_digit \rangle \langle hex_digit \rangle \langle hex_digit \rangle \langle hex_digit \rangle ) 

\langle alpha_character \rangle ::= ('a'..'z') | ('A'..'Z')

\langle hex_digit \rangle ::= \langle decimal_digit \rangle | ('a'..'f') | ('A'..'F')

\langle decimal_digit \rangle ::= '0'..'9'

\langle whitespace \rangle ::= ( \langle whitespace_character \rangle | \langle newline_character \rangle ) *

\langle inline_comment \rangle ::= '//' ( ! ( \langle newline_character \rangle ) ) *

\langle block_comment \rangle ::= '/*' ( ' ' | ' ' | ' ' ) ' */'

\langle whitespace_character \rangle ::= '\u0009' | '\u000B' | '\u000C' | ' ' | '\u0020'

\langle newline_character \rangle ::= '\u000A' | '\u000D' | '\u0085' | '\u2028' | '\u2029'

\langle keyword \rangle ::= 'architecture' | 'requirement' | 'quality_attribute' | 'rationale' | 'reason' | 'template' | 'provides' | 'requires' | 'property' | 'check' | 'system' | 'layer' | 'component' | 'connector' | 'link' | 'supports' | 'inhibits' | 'extends' | 'because' | 'over' | 'to' | 'in' | 'out' | 'subsetof' | 'true' | 'false'

\langle escape_sequence \rangle ::= ' \backslash ' ( 't' | 'b' | 'n' | 'r' | 'f' | ' ' | ' ' | '\u' \langle hex_digit \rangle \langle hex_digit \rangle \langle hex_digit \rangle \langle hex_digit \rangle )
REFERENCES


REFERENCES


REFERENCES

[106] Oracle, “Java BluePrints: Guidelines, patterns, and code for end-to-end applications.”

[107] Microsoft, “Patterns & practices: Developer Center Home.”


REFERENCES


