PANDArch: A Pluggable Automated Non-intrusive Dynamic Architecture Conformance Checker

Lakshitha de Silva and Dharini Balasubramaniam

School of Computer Science, University of St Andrews, St Andrews, KY16 9SX, UK
lakshitha.desilva@acm.org, dharini@st-andrews.ac.uk

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 be fully checked until deployment. Routine maintenance and changing requirements can also lead to a deployed system deviating from this architecture over time. Both static and dynamic checks are thus required to ensure that an implementation conforms to its prescriptive architecture throughout its lifespan. However, runtime conformance checking strategies typically alter the implementation of an application, increasing its size and affecting its performance and maintainability. In this paper, we describe the design of a novel dynamic conformance-checking framework that is pluggable and non-intrusive, thereby limiting any overheads to those periods when checking is activated. An implementation of this framework with Java as the target language and its early evaluation are also presented.

1 Introduction

A key benefit of software architectures [9] is that they establish the basis for system implementation. The essential structure, interactions and quality attributes captured at the architectural level can guide the development of the system [4].

Architecture-driven development methodologies can ensure that a software system conforms to its prescribed static architecture at the outset. However, verifying the compliance of dynamic features of an implementation is not always possible until the system is 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. Such erosion of the architecture [9] can lead to vital properties being violated and the software becoming unfit for use [14]. Both static and dynamic conformance checks are therefore required to ensure that an implementation and its architecture remain consistent with one another.

Dynamic architectural features include runtime instantiations, reflective method invocations, dynamic linking, online updates and patches, and quality of service measures. Detecting runtime violations requires system execution to be monitored and relevant runtime data be extracted, abstracted and checked against architectural constraints.

Most existing work in dynamic architectural conformance checking involves incorporating extra functionality, such as aspect weaving, or source code annotations of architectural properties in the target system. In both cases an external monitoring system reconstructs a runtime view of the architecture using data gathered from the added code

K. Drira (Ed.): ECSA 2013, LNCS 7957, pp. 240–248, 2013.

or annotations. The extracted architecture is then used for checking conformance against a prescriptive architecture using other tools [13]. Such conformance checking techniques are tightly coupled with the software product and cannot be invoked only when required. This limitation can lead to permanent degradation of application performance, inflexibility in conformance checking and poor maintainability [13].

This paper introduces PANDArch, a framework for checking conformance between software architectures and implementations that aims to solve these problems. The framework is designed to be automated, customisable, non-intrusive and pluggable, thus minimising overheads on applications. We outline its design, implementation and some early evaluation using architectures specified in the Grasp [12] architecture description language (ADL) and implementations in Java. The key contribution of this work is making dynamic architecture conformance checking a viable option for developers.

The paper is organised as follows. Section 2 outlines the concept of architectural conformance while Section 3 describes design principles that guided the framework. Key implementation details are discussed in Section 4 and preliminary evaluation results using an open source application are presented in Section 5. Section 6 describes related work and the paper concludes with thoughts on future work in Section 7.

2 Architecture Conformance Checking

An implementation that satisfies the constraints specified in its prescribed architecture is said to conform to it. Conformance can relate to a number of architectural properties relating to structure, interactions and quality of service (QoS) requirements. Our framework aims to check all these properties captured in the form of conformance rules.

2.1 Static and Dynamic Checking

An architecture specification may contain multiple views associated with static (such as code or development) or dynamic (execution) aspects of the system. As explained in Section 1, both static and dynamic checks are required to ensure full conformance.

Static checks are done while a system is being built or when taken offline for maintenance. They may relate to code structures and aspects of communication integrity.

Dynamic checks are carried out while the system executes and thus require access to runtime state and operations which are validated against architectural constraints. Such checks may relate to structures, communication integrity, component instances and quality of service thresholds at runtime. As method-level granularity is often required for dynamic checking, a key challenge is capturing relevant and useful data from system execution while still keeping any performance impact to a minimum.

2.2 Mapping between Architecture and Implementation

An architecture may be used to derive multiple system implementations. It exists at a higher level of abstraction and hence a single architectural feature can be implemented using a combination of programming constructs. Many implementation details are not

significant at the architectural level. Therefore, a mapping between architectural and implementation abstractions is required to perform conformance checking. We categorise mechanisms for specifying such mappings as follows:

- **Naming Conventions from Architectural to 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 behavioural and QoS properties as well. This technique can be further extended by supplementing architectural elements with annotations having rich information about corresponding implementation constructs. Conversely, annotations in source code could identify relevant architectural entities. An annotation for a class, for example, can identify the component implemented by that class. Source code annotations, however, could be easily lost due to programmer activity.
- **Combining Architecture and Implementation in a Single Artefact, as in ArchJava [1] or ArchWare [7].** Conformance checks are minimised or not required in such systems since architecture and implementation are combined in one specification. However, current approaches either require a permanent runtime platform or that the application be implemented in a language that is not widely adopted in industry.
- **External (Outwith both Architecture and Implementation) Specification of the Mapping.** While this mechanism does constrain architecture and implementation representations, it does require a separate artefact having explicit mapping of all architecturally significant features. The DiscoTect [11] technique uses this approach.

PANDArch adopts the first strategy for dynamic conformance checking. This decision avoids a complex mapping scheme that could hinder adoption, while still allowing the dynamic checks noted earlier to be carried out. It also makes the framework more readily adaptable to both existing modelling notations and programming languages.

3 Design Principles

As motivated in Section 1, the following principles guided the design of the framework.

- **Pluggable.** Target application can execute with or without the monitoring framework; when the framework is unplugged, the binaries have no instrumentation or other code,
- **Automated.** Generation of conformance rules and their checking are automated,
- **Non-intrusive.** The source code of the target application is not changed,
- **Performance-centric.** Performance impact on target application is minimised as far as possible and limited to the period when the framework is plugged in, and
- **Extensible.** The framework can accommodate modifications to conformance rules.

We hypothesise that these principles lead to a more viable compliance checking framework that aligns well with industry practices. The conceptual process for checking conformance using the proposed framework is illustrated in Figure 1 below.

An architecture specification, containing required mapping annotations, drives both system development and the conformance process. Architectural constraints and mapping information are extracted from the specification after compilation, and used by

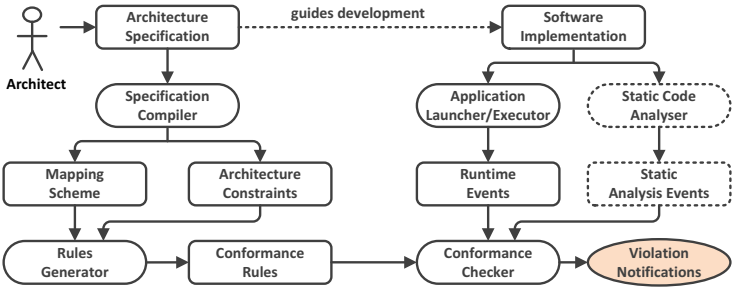

Fig. 1. Proposed conceptual process for checking architectural conformance

the rules generator to create a set of conformance rules. The mapping scheme is customised to the specific technology or language used for the target system. For instance, if the system is implemented in Java, then the mapping scheme is specific to Java and object-orientation. Rules are specified as Java objects exposing a specific interface.

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 architectural rule. The same process is applicable to events generated through static analysis. Although currently not implemented, the design can accommodate a static analyser using an adaptor to transform code inspection notifications to PANDArch events.

4 Implementation

The architecture of the conformance checking framework reflects the design principles listed in section 3. The layered architecture of the framework is shown in Figure 2.

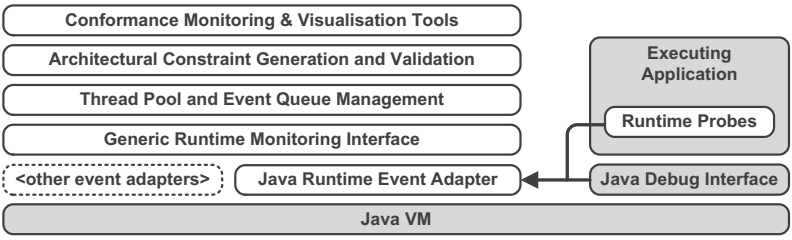

Fig. 2. Layered logical architecture of the conformance checking framework

PANDArch is implemented in Java and can currently check conformance of applications executed in the Java Virtual Machine (JVM). However, it is extensible to handle other event sources. Initially the JDI platform alone was used to capture events emitted

by the JVM. However, due to their impact on performance, Class-Load and Method-Entry events are now captured using a Java instrumentation agent [8]. The agent injects optimised probes into the byte code whenever the JVM loads an application class. These probes are both additive and stateless and therefore do not alter application behaviour. Byte code streams are modified only in-memory, hence changes are not persistent. Raw data from probes are sent to the framework through an asynchronous socket channel.

Our implementation uses architectures specified in the Grasp ADL [12], though the framework design is not tied to this notation. Besides common architectural concepts such as components, connectors and layers, Grasp also supports annotations. These are name-value pairs useful for supplementing architectural elements with additional data without altering semantics. In our case, annotations carry crucial mapping information linking architectural entities to their implementation, as explained further in Section 5.

5 Evaluation

The initial evaluation focuses on two aspects: the ability of PANDArch to detect conformance violations and its impact on the performance of target applications. We chose version 2.4.2 of Apache Jackrabbit [2], a Java content repository application, for the initial evaluation of the framework primarily because it includes some architecture documentation. As a server application, Jackrabbit is also suitable for testing performance impact. However, the published runtime architecture is neither complete nor up to date, particularly with respect to interactions among architectural elements [3]. In order to generate useful and sufficient conformance rules, the source code was manually examined to discover interactions among a few key components. Filtering capabilities of PANDArch were configured to monitor only these components at runtime. For these parts, conformance is guaranteed since the architecture reflects the implementation. However, where appropriate conformance was deliberately broken to test the effectiveness of the framework. The extracted architecture is shown in Figure 3.

The Grasp specification for the extracted portion of the architecture is shown in Listing 1. The whole architecture is contained within an architecture block while the runtime view is described within the system block. Components are described using the component keyword and in this example, each component declaration has an associated annotation that begins with @confomn. These annotations map components to implementation. For example, the annotation attached to component Data specifies that it has been implemented using all the classes found in the Java namespace org.apache.jackrabbit.core.data. Similarly, the two annotations attached to the root architecture statement identify namespaces that should be included and excluded from conformance monitoring.

Interactions among components are specified using the Grasp the link construct. A link connects a *requires* (i.e. consumer) interface in one component to one or more *provides* interfaces in other components. However, in real-world software component interactions are not always through interfaces, as exemplified in Jackrabbit. Grasp overcomes this by equipping every component with an intrinsic *out* interface to model outgoing, non-interface method calls to other components. This is evinced in Listing 1.

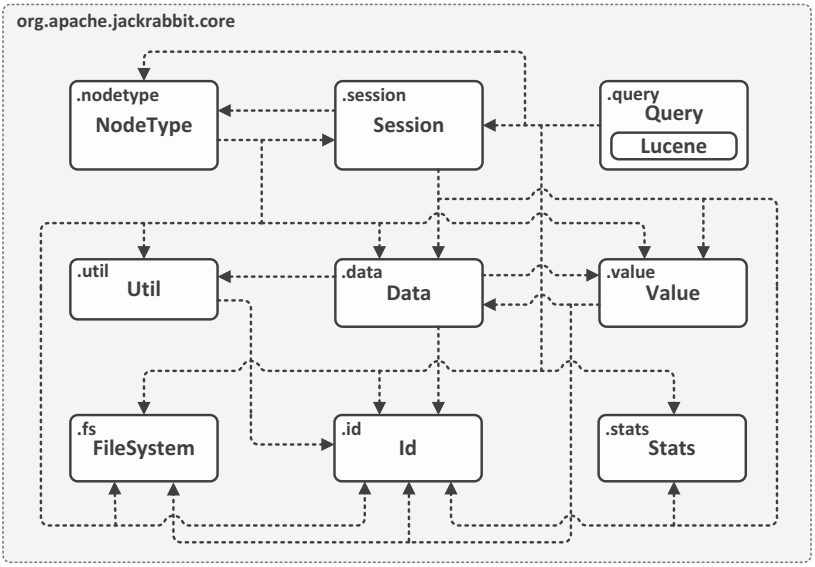

Fig. 3. Extracted architecture of Jackrabbit showing interactions among a few key components

5.1 Detecting Architecture Violations

The ability of the framework to detect architecture violations was evaluated using the above Grasp specification. Namespace filters were set to ignore components and interactions not included in this specification. Initial runtime tests were 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. As expected, the framework did not report any violations in the first instance as all components and their interactions were compliant with the architecture. The architecture and the implementation were then changed to cause mismatches. Particular attention was given to violations that could be detected only at runtime. For example, the AddNodeOperation class in the Session component was modified to instantiate a class in the Util component and invoke one of its methods using Java reflection. This interaction is not specified in the architecture and therefore should not be allowed. In addition, this reflective method invocation cannot be easily detected, if at all, through static analysis. As in all other cases, the framework correctly identified this violation when the test program was executed.

5.2 Performance Impact

The SecondHop program was also used to evaluate the performance of PANDArch. The program executes for ten iterations during a single run, and makes four such runs for each test case. The results of these tests are shown in Table 1. A significant performance gain is achieved by using instrumentation probes instead of JDI. Although

Listing 1. Grasp specification of modules and interactions shown in Figure 3

```
@confmon(include=["org.apache.jackrabbit.core"])
@confmon(exclude=["org.apache.jackrabbit.core.query.lucene"])
architecture Jackrabbit {
    template NamespaceComponent() {}
    system Core {
        // Components
        @confmon(ns=["org.apache.jackrabbit.core.nodetype"], classes=["*"])
        component NodeType = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.session"], classes=["*"])
        component Session = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.query"], classes=["*"])
        component Query = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.util"], classes=["*"])
        component Util = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.data"], classes=["*"])
        component Data = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.value"], classes=["*"])
        component Value = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.fs"], classes=["*"])
        component FileSystem = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.id"], classes=["*"])
        component Id = NamespaceComponent();
        @confmon(ns=["org.apache.jackrabbit.core.stats"], classes=["*"])
        component Stats = NamespaceComponent();
        // Interactions
        link NodeType.out to Session, Util, Data, Value, FileSystem, Id;
        link Session.out  to NodeType, Data, Value, Stats, Id;
        link Query.out    to NodeType, Session, FileSystem, Id, Stats;
        link Util.out     to Id, Util, Id, Value;
        link Value.out    to Data, FileSystem, Id;
    }
}
```

our test application 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.

The framework allows users to choose between JDI or probes for the purpose of fine tuning conformance checks. In comparison to probes, the JDI mode offers a more thorough conformance validation at the cost of reduced performance. The choice between the two may be dependent on a number of factors including whether dynamic conformance checking takes place as part of system testing prior to deployment or while the system is in operation, and how often such checks are carried out.

We also verified that PANDArch generates conformance rules automatically, does not affect the source code, and can be unplugged without recompiling the target.

6 Related Work

DiscoTect [11] uses runtime events, state information and rules for known architectural styles to discover the architecture of an executing Java program. It uses a mapping language to bridge the abstraction gap between architecture and implementation and conformance is checked manually. In contrast, PANDArch automatically validates constraints extracted from an architecture, irrespective of style, against the implementation.

A later work by Ganesan et al. [6] adapts DiscoTect by replacing its mapping language with Coloured Petri nets to link architecture to implementation. This technique

Table 1. A comparison of performance impact between JDI and instrumention probes. Tests were executed in a system with 2.26 GHz Core 2 Duo processors and 8GB of memory.

Run	Framework unplugged (μs)	Framework using probes (μs)	Framework using JDI (μs)
1	107,762	139,091	472,472
2	107,254	137,622	486,169
3	101,573	135,551	449,602
4	104,677	139,057	468,763
Average	105,317	137,830 (+30.9%)	469,251 (+345.6%)

is pluggable, non-intrusive and some aspects of the discovered architecture can be verified automatically. However, its mappings are also distinct from the architecture specification and stylistic architectural properties must be manually pre-configured in the checker. PANDArch uses a single architecture specification with in-built mappings, from which constraints used by the conformance checker are automatically generated.

Popescu and Medvidovic [10] propose a semi-automatic approach for checking dynamic compliance between an event-based system and its architecture. This approach injects probes and recorders into components, extracts and filters runtime data on events and compares it to a prescriptive sequence of events. It focuses on communications in event-based systems and requires some human interpretation to decide conformance.

The SAVE tool [5] uses runtime events as well as source code to extract architectural views, though runtime compliance checking is not possible.

7 Conclusions and Future Work

We have introduced a dynamic architecture conformance checking framework that is pluggable, automated, non-intrusive, and minimises overhead on target applications. An implementation of the framework for Grasp and Java is currently being evaluated.

This work opens up many avenues for further research. Extensive evaluation using different types of applications under different loads is required to determine viability and effectiveness of the framework. Although the core design of the framework does not preclude them, the current implementation does not support distributed applications or static conformance checking. We intend to incorporate these functionalities to improve applicability. Furthermore, a challenge faced by any dynamic program monitoring tool is ensuring sufficient execution coverage. We plan to address this issue by employing static analysis to preconfigure the runtime checker, so that runtime architectural violations can be meaningfully interpreted with relation to the amount of code covered during execution.

Acknowledgment. This work is supported through a PhD studentship awarded by Scottish Informatics and Computer Science Alliance (SICSA) and University of St Andrews.

References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM (2002)
2. Apache Software Foundation: Apache Jackrabbit (2010), http://jackrabbit.apache.org/ (accessed April 2013)
3. Apache Software Foundation: Jackrabbit Architecture (2010), http://jackrabbit.apache.org/jackrabbit-architecture.html (accessed April 2013)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley (2003)
5. Duszynski, S., Knodel, J., Lindvall, M.: SAVE: Software architecture visualization and evaluation. In: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, pp. 323–324. IEEE (2009)
6. Ganesan, D., Keuler, T., Nishimura, Y.: Architecture compliance checking at run-time. Information and Software Technology 51(11), 1586–1600 (2009)
7. Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cimpan, S., Warboys, B., Snowdon, B., Greenwood, R.M.: Support for evolving software architectures in the ArchWare ADL. In: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture, pp. 69–78. IEEE (2004)
8. Oracle: Package java.lang.instrument (2013), http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html (accessed April 2013)
9. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
10. Popescu, D., Medvidovic, N.: Ensuring architectural conformance in message–based systems. In: Proceedings of the Workshop on Architecting Dependable Systems (2008)
11. Schmerl, B., Garlan, D., Yan, H.: Dynamically discovering architectures with DiscoTect. In: Proceedings of the 13th ACM International Symposium on Foundations Software Engineering, pp. 103–106. ACM (2005)
12. de Silva, L., Balasubramaniam, D.: A model for specifying rationale using an architecture description language. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) ECSA 2011. LNCS, vol. 6903, pp. 319–327. Springer, Heidelberg (2011)
13. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. Journal of Systems and Software 85(1), 132–151 (2012)
14. van Gurp, J., Bosch, J.: Design erosion: problems and causes. Journal of Systems and Software 61(2), 105–119 (2002)