Architectural Assertions:
Checking Architectural Constraints at Run-Time

Hyotaeg Jung, Carlos E. Rubio-Medrano, Eric Wong, and Yoonsik Cheon

TR #07-18
April 2007; revised May 2007

Keywords: Architectural assertion, architectural constraint, runtime assertion checks, software architecture, architectural description language, Java Modeling Language.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification—assertion checkers, class invariants, formal methods; programming by contract; D.2.11 [*Software Engineering*] Software Architectures—Languages (e.g., description, interconnection, definition); F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, pre- and post-conditions, specification techniques.

To appear in *The 6th International Workshop on System and Software Architecture, June 25-28, 2007, Las Vegas, NV.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Architectural Assertions: Checking Architectural Constraints at Run-Time

Hyotaeg Jung[1], Carlos E. Rubio-Medrano[2], W. Eric Wong[1], and Yoonsik Cheon[2]

[1]Department of Computer Science
The University of Texas at Dallas
Dallas, TX 75083

[2]Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968-1580

Abstract – *The inability to express architectural concepts and constraints explicitly in implementation code invites the problem of architectural drift and corrosion. We propose runtime checks as a solution to mitigate this problem. The key idea of our approach is to express architectural constraints or properties in an assertion language and use the runtime assertion checker of the assertion language to detect any violations of the constraints. The architectural assertions are written in terms of architectural concepts such as components, connectors, and configurations, and thus they can be easily mapped to or traced back to the original high-level constraints written in an architectural description language. We believe that our approach is effective and more practical than and complements static techniques.*

Keywords: architectural assertion, software architecture, architectural constraints, runtime assertion checking, architectural constraint language, JML language

1 Introduction

The architecture of a software system is a blueprint, abstracting from the system into constituent elements such as components, connectors, and configurations. The software architecture, described in an architectural description language (ADL), provides a great help to developers not only for the construction but also the maintenance and evolution of the system. However, even with rigorous development and maintenance, software tends to lose its original architectural structure, often referred to as *architectural drift and corrosion*, and becomes difficult to understand and modify [14].

The problem of architectural drift and corrosion becomes aggravated in reality because there are only a few programming languages available such as ArchJava [1] that directly support architectural concepts as built-in language constructs; unless a special framework such as Prosm-MW [11] is used, the architecture specified in ADL should be reified into implementation artifacts and thus gets lost in the implementation code. The inability to express architectural concepts explicitly in the code opens the door wide for architectural drift and corrosion. As the architectural information is not explicitly expressed in code, a modification to the code may cause the design of the system to begin to drift or deviate from the original architecture without being noticed by the developer.

In this position paper, we propose runtime assertion checks as a partial solution to the problem of architectural drift and corrosion. In particular, we advocate documenting architectural constraints or properties in code in a form that can be evaluated and checked at run-time that we call *architectural assertions*. An architectural assertion not only detects architectural drift at run-time but also provides an excellent document of the architecture that is always synchronized with the implementation. We expect that we can translate automatically a wide range of architectural constraints and properties from descriptions written in ADLs to architectural assertions.

We believe that architectural assertions provide a practical and effective way to detect architectural drift and complement static techniques such as formal validation and verification.

2 Research Hypotheses and Claims

In programming, an assertion is a predicate placed in a program to indicate the truth of the assertion at that place [15]. It is used to specify and reason about the correctness of a program both statically, as in Hoare-style pre and post-conditions [8], and dynamically, as in Design by Contract [12] and `assert` macros of C/C++.

We claim that assertions can be an effective and practical way to express and check at run-time the architectural constraints or properties of a software system. We further hypothesize that, with a suitable assertion framework in place, a wide class of important architectural constraints and properties can be automatically translated to executable assertions. In the next section, we explain our approach to checking architectural constraints at run-time by presenting our conceptual framework and a particular instantiation of it.

3 Our Approach

The underlying idea of our approach is to employ the runtime assertion checker of an assertion language to detect at run-time violations of architectural constraints or properties, thus architectural drift and corrosion. The key component of our approach is thus to (automatically) translate architectural constraints or properties written an architectural constraint language (ACL) X to executable assertions in an assertion language (AL) Y (see Figure 1). We call the resulting executable assertions *architectural assertions*, and there are several different types of architectural assertions, such as structural assertions, behavioral assertions (e.g., protocols), and non-functional assertions (e.g., security and performance).

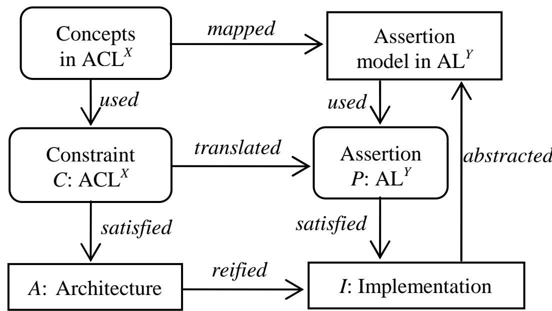

Figure 1. Conceptual framework of our approach

Two key requirements of our approach are semantic preservation and traceability. The translation from ACL to AL should preserve the semantics of ACL in that if an architecture model A satisfies a constraint C, its correct implementation I should also satisfy the translated architectural assertion P and vice versa; this is the soundness and completeness requirement of the translation. We expect to be able to prove at least the soundness of our translation: if an execution results in a violation of an architectural assertion, it means a violation of the corresponding architectural constraint.

The second requirement is to facilitate error tracking and help in preventing divergence between architectural assertions and original architectural constraints (i.e., drift of documents). It states that once a violation of an architectural assertion is detected, the assertion should be easily traced back to the original constraint or property. Our approach is to write architectural assertions not in terms of implementation artifacts but in terms of an abstraction of the implementation artifacts from which the mapping to the original constraints is clear (see an example in Section 3.1). We call this abstraction *an assertion model for architecture*, and we expect to provide a suite of them, one for each ACL, perhaps organized in a class hierarchy in the form of an assertion or specification library.

3.1 Illustration

We illustrate our approach with a small example by instantiating our conceptual framework with Armani and JML. Armani [13] is the constraint language of Acme, and JML [9] is a formal behavioral interface specification language for Java. JML allows one to document formally the behavior of Java program modules such as classes and interfaces, and a significant subset of JML assertions can be checked at run-time by the JML compiler (jmlc) [3] [4].

Suppose that we would like to constrain a specific component of an architectural model such that each of its subcomponents should have fewer than five provided interfaces. This constraint taken from [16] is written in Armani as follows.

```
invariant forall com: Component in self.Components |
  forall p: Port in com.Ports |
  size({select p: Port in com.Ports | satisfiesType(p, inputT)}) < 5;
```

The above constraint is translated to the following JML specification, assuming that the component of interest is implemented by a pair of an interface and an implementing class, InterfaceA and ClassA.

```
//@ model import org.jmlspecs.models.arch.armani.*;
public interface InterfaceA {
  //@ public model Component theModel;
  /*@ public invariant (\forall Component c;
    @    theModel.getComponents().contains(c);
    @    (\num_of  Port p; c.getPorts().contains(p); p.isInputT())
    @    < 5;
    @*/
    // the rest of definition of InterfaceA …
}

public class ClassA implements InterfaceA {
  //@ private represents theModel <- toMode(this);
  /*@ private model pure Component toModel(ClassA c) {
    @    /* definition of abstraction function */ @*/
  // the rest of definition of ClassA …
}
```

As shown in the example, JML specifications and assertions are typically annotated to a Java source code file as a special kind of comment. The first annotation imports a set of model classes for use in JML assertions and specifications. The imported classes provide an abstract model for architectural concepts found in Armani; e.g., they include such immutable classes as Component and Port. The next annotation states that the field theModel, providing an abstract model of the interface InterfaceA, is a specification-only field; it can be used only in JML specifications and doesn't have to be implemented by an implementing class. The next *invariant* clause is a direct translation of the constraint written in Armani. Note that, except for a minor syntactic difference, the structure of the invariant is the same as that of the original constraint.

The mapping from implementation artifacts to the abstract model is specified in an implementation class by using the *represents* clause. In the example, the abstraction function for the model field theModel is defined in terms of a side-effect-free specification-only method toModel. Given an abstraction function, assertions written in terms of a model field, such as the invariant in the example, can be evaluated and checked at run-time [5].

In summary, architectural assertions in JML are written in terms of model variables [5], and given a suitable model classes for Armani, the translation of architectural constraints in Armani to architectural assertions in JML is straightforward.

3.2 Research Issues

In addition to its benefits, our approach to checking architectural constraints or properties at run-time poses several challenges for research, some of which are briefly discussed below.

What types of constraints to check? We'd like to know the types or kinds of architectural constraints that can be efficiently checked at run-time. Our goal is not to be able to check all sorts of constraints at run-time but to use runtime checks as a practical, complementary technique to traditional static techniques such as formal verification. For this, we first plan to classify and categorize different architectural constraints and properties (e.g., structural, protocol, security, and performance) and then evaluate the amenability of different properties for runtime checks.

How to specify architectural constraints or properties? There are several notations or languages to describe software architectures, called ADLs (e.g., WRIGHT [2], Acme [7], xADL [6], and Darwin [10]), some with a separate ACL (e.g., Armani for Acme). Ideally, the constraints should be specified formally so that they can be automatically translated to executable assertions in assertion languages. We'd like to leverage existing research efforts in this area by adopting and, if necessary, extending an existing ACL. The work of Tibermacine, *et al.*[16] is interesting in that it proposes to unify several ACLs. The key idea is to use the same core language based on the Object Constraint Language (OCL) [17] for writing constraints and to have different ACL profiles (MOF meta models) for different ACLs.

How to translate architectural constraints to executable assertions? A key concern here is to preserve the level of abstractions in the translated assertions so that they can be easily mapped back to the original constraints. An assertion language with extensible vocabularies would be of a great help. In JML, this is done through model elements such as model fields, model methods, and model classes and interfaces. The translation rules from constraints to assertions should be sound and, ideally, complete. In addition to defining the translation rules, developing a suitable specification vocabulary for the translation (e.g., model classes in JML) will be an important research topic in this area.

What extensions to the JML language and tools? We expect to be able to translate a majority of important classes of static and dynamic structural constraints directly to JML. However, JML may need to be extended for the translation of behavioral or non-functional constraints, though a limited form of support is already provided for them[1].

How to minimize the runtime cost of checking architectural assertions? We can envision two different uses of architectural assertions, as a development tool for testing and debugging programs and as a runtime monitoring tool for deployed software systems. The second use, for example, will allow us to detect architectural deviation and corrosion due to dynamic reconfigurations or evolutions of the systems. However, for such a use to be practical, the runtime overhead of checking architectural assertions (e.g., time and space requirements) should be minimal. Another issue regarding the performance is, when measuring and checking performance constraints, how to identify and exclude the performance cost due to the runtime checking itself.

4 Contributions

The primary contribution of this work is the conceptual framework that allows one to check at run-time architectural constraints or properties and thus to detect architectural drift and corrosion automatically, one of the biggest problems in software maintenance and evolution. The seminal feature of our framework is that it leverages the recent advances of two different but related areas, software architecture and runtime assertion checks. Research in software architecture provides a formal notation (and semantics) to document key architectural properties concisely and precisely. The advances in runtime assertion checks (e.g., model variables and quantifiers) allow us to express and check a wide class of architectural assertions.

The secondary contribution is an instantiation of our conceptual framework for a particular pair of an architectural constraint language and an assertion language, i.e., Armani and JML. A nice byproduct of this instantiation is a set of JML model classes to represent the architectural concepts expressible in Acme.

[1] There are several extensions to JML that allow one to specify and check the allowed sequences of method calls, and one of these extensions may be used to check protocol properties of an architecture model. JML also provides a facility to document the time and space requirements of a program module; however, these so-called *performance contracts* are not supported by the runtime assertion checker yet.

We will complete the implementation of the above instantiation and related automatic tools (e.g., Armani-to-JML translator).

5 Evaluation

We plan to perform several case studies on the applications of our approach and automated tools. The goal of our case studies is to evaluate the effectiveness and practicality of our approach and tools. We will measure qualitatively and, if possible, quantitatively such factors as characteristics of runtime-checkable constraints, expressiveness and readability of ACL and AL (especially, our extensions to ACL and JML), degree of automation translating ACL to AL, traceability of translated assertions, and runtime costs of architectural assertions in terms of time and space requirements. Initially, we will focus on functional properties of the architecture such as structural constraints and then extend to non-functional properties such as protocol, security, and performance constraints. For the evaluation of non-functional properties, we will consider applications such as Domain Name Service (DNS) that translates domain names to IP addresses. For example, we may be able to specify and check at run-time such performance constraints as response time and throughput.

6 Conclusion

We propose architectural assertions as a practical way of detecting architectural drift and corrosion at run-time. Architectural assertions are assertions that document architectural constraints or properties in code in a form that can be evaluated and checked at run-time. The novelty of our approach lies in the fact that we take full advantage of existing runtime assertion checking facilities such as JML by automatically translating architectural constraints or properties written in architectural description or constraint languages such as Armani to assertion languages such as JML. We are currently performing a case study in JML to evaluate the effectiveness of our approach before we start full-blown development. Our case study focuses on developing an assertion library for JML that will assist us to translate architectural constraints and properties to JML assertions. The assertion library is a set of immutable Java classes that models architectural concepts such as components, connectors, ports, and roles.

Acknowledgment

Hyotaeg Jung is supported by The University of Texas at Dallas. Rubio-Medrano and Cheon are supported in part by the NSF under grant CNS-0509299.

7 References

[1] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 187-197, Orlando, Florida, May 2002.

[2] R. J. Allen, "*A Formal Approach to Software Architecture*," Ph.D. Dissertation, CMU-CS-97-144, Carnegie Mellon University, May 1997.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005.

[4] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of International Conference on Software Engineering Research and Practice*, pp. 322-328, Las Vegas, Nevada, June 2002.

[5] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software-Practice & Experience*, 35(6):583-599, May 2005.

[6] E. M. Dashofy, A. Hoek, and R. N. Tylor, "An infrastructure for the rapid development of XML-based architecture description languages," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 266-277, Orlando, Florida, May 2002.

[7] D. Garland, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference on the Center for Advanced Studies on Collaborative Research*, pp.169-183, Toronto, Canada, November 1997.

[8] C.A.R. Hoare, "An axiomatic basis of computer programming," *Communications of ACM*, 12(10):576-580, October 1969.

[9] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the 5th European Software Engineering Conference (ESEC '95)*, pp. 237-153, Barcelona, Spain, September 1995.

[11] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Transactions on Software Engineering*, 31(3):256-272, March 2005.

[12] B. Meyer, "Applying design by contract," *Computer*, 25(10):40-51, October 1992.

[13] R.T. Monroe, "*Capturing Software Architecture Design Expertise with Armani*," Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 2001.

[14] D.E. Perry and A.L. Wolf, "Foundations for the study of software architecture", *ACM SIGSOFT Software Engineering Notes*, 17(3):40-52, October 1992.

[15] D.S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, 21(1):19-31, January 1995.

[16] C. Tibermacine, R. Fleurqum, and S. Sadou, "Simplifying transformation of software architecture constraints," in *Proceedings of ACM Symposium on Applied Computing 2006*, pp. 1240-1244, Dijon, France, April 2006.

[17] J. Warmer and A. Kleppe, "*The Object Constraint Language: Precise Modeling with UML*," Addison-Wesley, 1999.