

# Testing Complex Architectural Conformance Relations

Michal Young

University of Oregon

Department of Computer and Information Science

Eugene, Oregon USA

+1 (541) 346-4140

michal@cs.uoregon.edu

## ABSTRACT

Architectural conformance testing can help detect inconsistencies between an architectural model that has been verified and a system that has been implemented. Architectural descriptions suitable for powerful static analyses are necessarily abstractions and idealizations of the actual implementation structure of software systems, making the relation between architectural model and as-built structure complex. We argue that an explicit representation of complex conformance relations is a necessary basis for architectural conformance testing.

## Keywords

Conformance testing, model checking, specification-based testing

## 1 Introduction

Static verification of certain critical properties is one of the main attractions of replacing informal white-board sketches by formal representations of software architectures. The value of static verification is magnified if there is some assurance that the actual software system is consistent with the architectural descriptions that have been verified. We make the following arguments:

- Safe (conservative) verification of architectural conformance is unlikely to make architectural conformance testing redundant, even when large parts of a system are generated directly from architectural descriptions and therefore putatively “correct by construction.”
- Although directly testing an implementation for properties that have been verified in an architectural model is probably advisable as a redundant check, it is a poor replacement for testing for conformance to the architectural model.
- Requiring simple relations between architectural models and implementations may result either in poor implementations or in architectural models that are unsuitable for some useful kinds of analysis. Conversely, crafting architectural descriptions that are suitable for powerful static analyses will tend to make the relation between architectural

model and implementation less straightforward, and therefore make architectural conformance testing less straightforward also.

- Explicit maintenance of a many-to-many relation between architectural components and implementation modules is necessary as a basis for architectural conformance testing when the relation between architecture and implementation is not straightforward.

While we expect that this general argument might apply with respect to many kinds of architectural representations and architecture-level analysis, our experience is primarily in analysis of concurrent process interaction, and we therefore illustrate the argument with examples from that domain.

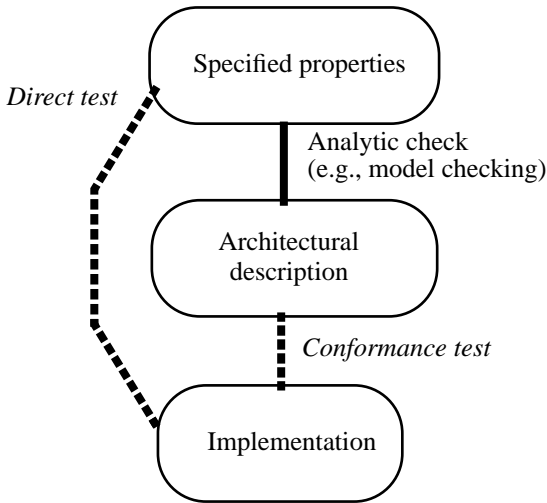
## 2 Why Conformance Testing?

Testing is inherently imperfect. One would prefer an analytic guarantee of architectural conformance if one were available, but it is unlikely that such a guarantee can be obtained.

If an application were completely generated from an architectural specification, then the architecture description language would be a programming language, and generation would be equivalent to compilation. We are in the habit of trusting compilers, and not worrying overmuch about conformance between our high level programs and assembly code; likewise testing would be redundant in the case of applications completely generated from verified architectural descriptions. In practice, though, the architectural descriptions that are simple enough for verification are not those that are detailed enough to for complete generation of practical implementations. More commonly, the architectural description is a model of program that may be partly generated but also includes some components that are either built manually or are imported from (unverified) libraries.

When parts of an application are not mechanically derived from the architectural description, the verified architectural description is a model of the system, and we must find a way to gain some confidence that the model accurately reflects reality (or, equivalently, that

**FIGURE 1. Direct vs. conformance testing**



the implementation conforms to the architectural description). Analytic methods may play some part in gaining this confidence, but they are unlikely to be sufficient. The correspondence between analysis and synthesis is such that, when automatically synthesizing components is not feasible, fully automatic analysis is also unlikely to be feasible.

### 3 Direct Testing vs. Conformance Testing

Implementations can be tested directly against the same high-level specifications which have been verified in a high-level architectural model, using for example the specification-based testing methods of O'Malley et al. [1]. We can represent this “direct testing” approach as illustrated in Figure 1.

Direct testing, in which test oracles are derived from properties that both the architectural model and implementation must obey, can accept some behaviors that deviate from the architectural model without violating the specified properties. In that case, the implementation behavior is “accidentally correct” (for at least those cases; it may be incorrect for other behaviors that have not been observed in testing).

Even if all possible behaviors obeyed the specified properties, “accidental correctness” is not a desirable situation. To see why, consider what happens when the implementation is enhanced in maintenance. A maintenance programmer is justified in changing the implementation in ways that preserve its correctness by preserving its conformance to the architectural model. However, if the implementation does not in fact conform to the architecture, these supposedly correctness-preserving changes can in fact introduce faults.

Conformance checking can be represented as follows:

A conformance test oracle derived from a verified architectural model must reject every behavior that would be rejected by a direct test oracle, and may also

reject some behaviors (those we have called “accidentally correct”) that would not be rejected by a direct test oracle. For example, suppose one of the specified properties is mutual exclusion between a pair of conflicting operations, and suppose that in the architectural design that property is achieved by performing the operations only while a semaphore is held. Suppose further that the semaphore operations are omitted in one of the processes making up the implementation. Direct testing would recognize a violation of the mutual exclusion property only when the actual conflicting operations were interleaved. Conformance testing would recognize the far more frequent situation in which the faulty process failed to perform the semaphore operations, even if (as happens often) the mutual exclusion condition was satisfied by pure luck.

### 4 Simple vs. Complex Architectural Relations

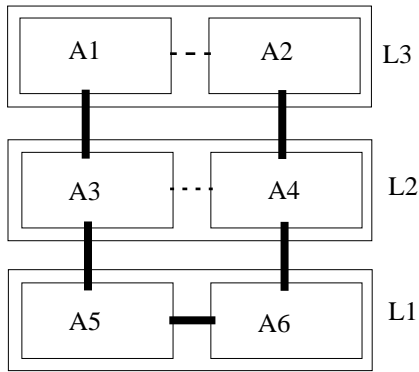
Conformance testing may be fairly straightforward if the relation between an architectural model and an implementation is characterized by a function from implementation components to architectural components.<sup>1</sup> We call such a function a simple architectural relation, because each architectural component is associated with some aggregation of implementation components.

It may happen, though, that components of the implementation cannot be simply aggregated to correspond to components of the implementation. For example, a logical (architectural) representation of a distributed system often has a layered structure, but layers may be combined in an implementation for reasons of performance, or simply because components without layered structure have been reused in a new application. This is illustrated in Figure 2. For example, if a process algebra were the architectural formalism used to specify behavior in the logical layered design, then the implementation component I3 might correspond to the composition  $A3||A5$  of two components in the architectural model.

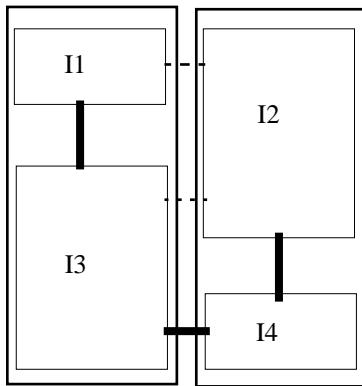
Complex relations between architectural models and implementations are not only artifacts of implementation decisions that degrade architectural structure. Limitations of model verification technology also lead to restructuring of architectural models. An architecture model which seems natural in other regards may be ill-structured for the purpose of automated analyses. For example, many researchers have applied finite-state verification methods such as temporal logic

1. It is convenient to further assume that the function is onto (i.e., there is no architectural component that corresponds to zero implementation components). Permitting architectural relations that are not onto may be useful for representing incompletely implemented systems and systems with optional parts, but it would complicate the presentation here without adding anything important to discussion of conformance testing.

**FIGURE 2. Logical layered architecture and as-built implementation architecture.**



**(a) logical layering for analysis**



**(b) as-built system with merged layers**

model checking or compositional reachability analysis to architectural models. These methods are very sensitive to the structure of models to which they are applied, and often suffer the well-known state space explosion problem for models that are not carefully crafted for analysis. While it would be pleasant to imagine that the structure imposed to achieve practical verification corresponds to a structure that is otherwise attractive, this is not always the case. It (subjectively) seems to be the case that an efficiently analyzable structure is often an easily understood structure, but it may be far from the structure that one would prefer to implement.

A particular case of restructuring for efficient finite-state verification occurs when a system has several identical processes. A typical approach is to build a model with fewer processes, on the grounds (whether formally justified or not) that the property violations that occur in a full-size system will also occur in the scaled-down version. This is a kind of ersatz induction on the number of processes in the system. Of course we would rather use real induction, but that is often very

difficult.

In at least some cases, a system for which a straightforward architectural description does not provide opportunity for induction can be restructured as a logical architecture for which (real, not ersatz) induction over the number of components is possible. Our experience with this approach is so far quite limited, but we will sketch one non-trivial example for which this is the case.<sup>1</sup>

An elevator simulator has been previously used in studies of the specification-based testing approach that we have here called “direct testing.” A straightforward architectural representation of the elevator system consists of a process for each elevator car, another for a controller that schedules the cars, and a few more. We have successfully used reachability analysis tools based on process algebra to verify versions of the system with up to four elevator cars (after first correcting errors that we successfully detected in smaller versions). Although we believe that the four-car elevator system is probably “good enough” as a representation of larger versions of the problem, we had no proof that this was the case.

Process algebra is in principle quite suited to induction over the number of processes in a system: One only needs to show that a system  $S$  is congruent to the composition of  $S$  with one more process. In practice it is seldom so straightforward, and the elevator model is typical in this regard. The controller component of the elevator system varies with the number of elevator cars to be controlled. Thus we cannot simply compose one more elevator with a system having  $n$  elevators; we must also account for the changes to the controller component as elevators are added.

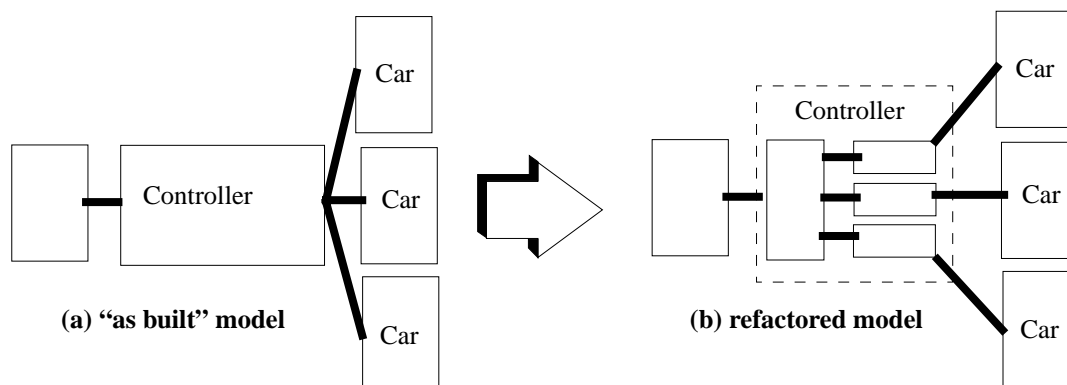
Inductive verification of the elevator system was accomplished by refactoring the controller process: It was divided into a “core” controller and a set of processes paired with elevator cars (Figure 3). The inductive step in the verification introduced both a new elevator car and the corresponding new fragment of the controller process. The breakdown of the controller component for verification is wildly different from any reasonable design for implementing the component, but by composing the right set of processes we can still derive from the verified model a black-box model of the intended behavior of the controller component.

## 5 Complex Mappings and Oracle Generation

We believe that complex mappings between architectural models and implementation structures will be the rule, not the exception. In addition to the transformations we have briefly outlined, there are a number of additional ways in which models are routinely changed to make them more amenable to automated analysis. Holzmann has argued [2] that

1. The details of this example were worked out primarily by my graduate student Yung-Pin Cheng, and will appear in more detail in a forthcoming coauthored paper.

**FIGURE 3. Refactoring the elevator model for inductive verification**



finding the right abstractions for verification is a creative process, and that starting from a straightforward representation of the actual implementation structure is almost always a bad idea. We see little reason to believe that this will be less true of architectural models. In fact it is likely that at least three different structures will be routinely constructed: A “reference” architecture which represents the architects’ understanding of the system, a model derived from the reference architecture but crafted specifically for verifiability (by whatever methods), and an implementation architecture that is also derived from the reference architecture but takes implementation constraints into account.

If we take as given that the relation between a verified architectural model and an implementation is not simple, then additional support is needed for effective conformance testing. We conjecture that it will be useful to make the structural relation between a verified architectural model and an implementation architecture explicit, and to use that relation to generate test oracles derived from the architectural model. These oracles will be transformed to correspond appropriately to components and subsystems of the implementation. With such a system, the user is still burdened with the job of describing the relation between the structures, but the user need not consider or understand how this relation is used in producing test oracles for conformance testing.

Our first attempt to provide this kind of support is based on an editor for transforming one architectural structure into another. It permits a number of useful transformations, such as aggregating processes and appropriately renaming and reexporting their interfaces. A representation of the transformations (for now we assume they are transformations from the as-built model to the verifiable model) is recorded and used in generation of test oracles for implementation components. Our experience to date is too limited to claim that this editing-based approach to capturing complex architectural relations is adequate or useful.

## 6 Summary

Our main claims are that architectural conformance testing is sometimes more effective than direct specification-based testing, and that generation of conformance test oracles is complicated by the need to account for complex relations between a structure which is designed to be efficiently verifiable and a structure which is intended to be implemented. We have given some basic examples (by no means exhaustive) of ways in which models may be restructured for analysis, and described one way in which a record of that restructuring might be captured and then used in deriving implementation test oracles from a verified architectural model.

## 7 References

1. O'Malley, T.O., S.L. Aha, and D.J. Richardson. *Specification-based test oracles for reactive systems*. In *International Conference on Software Engineering*. 1992. Melbourne, Australia.
2. Holzmann, G. *Designing executable abstractions (Keynote address)*. In *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*. 1998. Clearwater Beach, Florida: ACM Press.