From the Proc. of the Second Inter. Software Architecture Workshop, San Francisco, California, USA, October 1996

Software Testing at the Architectural Level

Debra J. Richardson
Information and Computer Science
University of California
Irvine, CA 92697-3425
djr@ics.uci.edu

Alexander L. Wolf
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
alw@cs.colorado.edu

Abstract

This paper argues that with the advent of explicitly specified software architectures, testing can be done effectively at the architectural level. A software architecture specification provides a solid foundation for developing a plan for testing at this level. We propose several architecture-based test criteria based on the Chemical Abstract Machine model of software architecture. An architectural (integration) test plan, developed by applying selected of these criteria, can be used to assess the architecture itself or to test the implementation's conformance with the architecture. This facilitates detecting defects earlier in the software lifecycle, enables leveraging software testing costs across multiple systems developed from the same architecture, and also leverages the effort put into developing a software architecture.

1 Introduction

A number of researchers have been experimenting with architecture definition languages and architecture description models [1, 2, 7], which formally specify software architecture. A software architecture specification makes the analysis, design and construction of a complex system intellectually tractable by characterizing the system at a high level of abstraction. Such a specification enables the engineer to reason about how system requirements are satisfied in terms of the assignment of functionality to design components and the interaction of those components via their interfaces. This reasoning may uncover architectural defects, where the interaction or communication between components is incorrect. Incompatibility of the data exchanged between components, for instance, is detectable via static analysis of the component signatures. On the other hand, revealing defects in the dynamic interaction and communication behavior between components may require dynamic analysis—e.g., testing.

Effective testing requires exercising the aspects to be reasoned about. The typical manner for accomplishing this is to use test criteria that define data to cover the aspects to be tested. Based on formal notations for specifying software architectures, architecture-based test criteria can be defined, which would enable automatically defining data to cover a system's architectural aspects and also deriving architectural test plans. Thus, the anticipated, widespread use of formal architecture specification will greatly facilitate testing dynamic component interaction and afford detecting architectural defects early in the development process.

Perry and Wolf developed a framework for architectural description [12] in which a software architecture specification consists of elements (processing, data and connecting elements) and form (relationships among the elements). Based upon this framework, Inverardi and Wolf developed a model for operationally describing software architectures [8] based on viewing a software system as chemicals whose reactions are governed by rules. This metaphor was formulated as the Chemical Abstract Machine (CHAM) [3]. In this paper, we argue that the CHAM model for software architecture provides a solid foundation for defining architecture-based test criteria and developing test plans for assessing dynamic behavior at the architectural level.

2 Review of the CHAM Formalism

The Chemical Abstract Machine is a term rewriting formalism that leads to a description of an architecture as a set of static components (the "molecules") whose states and interactions are governed by transformation rules (the "reactions"). Here, we review the CHAM model, limiting ourselves to those concepts directly required for this position paper. The interested reader is referred to [8] for more detail on the use of CHAM to model software architectures and to [3] for an even more complete description of the model.

A Chemical Abstract Machine is specified by defining *molecules* $m_1, m_2, \ldots$ and *solutions* $S_1, S_2, \ldots$ of molecules. Molecules constitute the basic elements of a CHAM, while solutions are multisets of molecules interpreted as defining the states of the CHAM. A CHAM specification also contains *transformation rules* $T_1, T_2, \ldots$ each defining a *transformation relation* $S_1 \longrightarrow S_2$ dictating the way solutions can evolve (i.e., states can change) in the CHAM.

Molecules are defined as terms of a syntactic algebra that derive from a set of constants and a set of operations defined for a specific CHAM. For architecture specification, these sets would include a set D representing the data elements, a set P representing the processing elements, and a set C representing the connecting elements. See [6, 8] for examples of CHAM molecule syntaxes.

SIGSOFT 96 Workshop, San Francisco, CA USA
© 1996 ACM 0-89791-867-3/96/10...$3.50

The transformation rules can be of two kinds: general laws that are valid for all CHAMs and specific rules defined for the particular CHAM being specified. Compare, Inverardi, and Wolf [6, 8] define several CHAM-specific rules. The Reaction Law is one general law. Other general laws are provided in [8].

> *The Reaction Law.* An instance of the right-hand side of a rule can replace the corresponding instance of its left-hand side. Thus, given the rule
>
> $$M_1, M_2, \ldots, M_k \longrightarrow M'_1, M'_2, \ldots, M'_l$$
>
> if $m_1, m_2, \ldots, m_k$, and $m'_1, m'_2, \ldots, m'_l$ are instances of the $M_{1\ldots k}$ and $M'_{1\ldots l}$ by a common substitution, then we can apply the rule and obtain the following solution transformation.
>
> $$m_1, m_2, \ldots, m_k \longrightarrow m'_1, m'_2, \ldots, m'_l$$

At any given point, a CHAM can apply as many rules as possible to a solution, provided that their premises do not conflict—that is, no molecule is involved in more than one rule. Thus, it is possible to model parallel behaviors by performing parallel transformations. The CHAM makes a nondeterministic choice as to which transformation to perform when more than one rule can be applied to the same molecule or set of molecules.

3 Architecture-based Testing with CHAM

Testing is typically done in several stages. Unit and module testing analyzes the local behavior of individual components. Integration testing analyzes how individual component behaviors, and the interactions among components, contribute to the global system behavior. System testing analyzes the global behavior of the system without regard to its decomposition. Other testing stages, such as acceptance and stress testing, are specialized forms of these basic testing stages.

Unit and module test plans are derived either via structural techniques from the source code or via functional techniques from a specification or knowledge of the component's intended behavior.[1] System test plans are typically derived via functional techniques from some statement or knowledge of requirements and user needs. Integration test plans are typically derived from implemented component interfaces and some hierarchical representation (e.g., a call graph) of the static invocations present in the integrated system or subsystem, thus using a combination of functional and structural techniques, respectively.

Most approaches to integration testing are based on the implementation, which delays use of the integration test plan until system integration. Integration test plans could be derived from a system design, yet most design representations are not formal enough to do this in an automated fashion. With the advent of formal architecture specification, however, architecture-based test criteria can be defined in much the same way as have been implementation-based and specification-based test criteria. This would support algorithmically defining test data to cover the architecture and automatically developing architectural test plans—integration test plans at the architectural level.

[1] Functional, or black box, techniques are based on only a description of functionality, while structural, or white box, techniques use the internal structure to guide testing.

By their very nature, implementation-based test criteria are primarily structural. Structural test criteria use the internal structure of software to define test data with the intention of exercising, or covering, certain aspects of the system. Several unit-level, structural test criteria have been defined based on control flow, such as statement and branch coverage. In addition, numerous criteria have been defined based on relationships between control structures, such as data flow criteria (e.g., all-def and all-use coverage). Clarke et al. [5] compare several families of data flow criteria; each family forms a hierarchy reflecting the relative comprehensive coverage of the criteria. Podgurski and Clarke [13] consider more general dependence relationships between components. This idea of developing a family of structural criteria based on relative coverage has also been extended to define test criteria for concurrency [16]. Another structural approach is fault-based test criteria, which prescribe test data geared to detecting particular kinds of faults, the goal being to cover the likely fault types and demonstrate the absence of those faults [10, 14]. Relationships between fault-based criteria have been analyzed as well [15].

More recently, structural criteria have been developed based on specifications (see, for example, [4]). These approaches blend functional and structural techniques in that the specification of a component's "function" is used to define the test data, yet the criteria define test data in terms of the syntax and semantics, or "structure", of the specification. One might think of a specification as consisting of assertions that define the software's expected behavior. Approaches to specification-based testing, therefore, have defined control and data flow criteria as well as fault-based criteria with regard to a specification's assertions.

It is these specification-based approaches that leads us to architecture-based test criteria. Just as an individual unit's implementation and specification consist of data, statements or assertions, and control, an architecture is similarly composed of data elements, processing elements, and connecting elements [12]. These elements, as well as the complex of relationships among the elements, should be exercised to adequately test the architecture.

The CHAM for a software architecture defines molecules (elements), solutions (combinations of elements), and transformations between solutions, all of which should be exercised during testing at the architectural level. We can, therefore, define architecture-based test criteria that require covering these structures. Basically, applying any one of our CHAM-based criterion proceeds as follows: (1) it determines the set of structures to be covered by the test plan; (2) it specifies a set of "paths" through the architecture that cover these structures, where each "path" is the set of solutions generated; and (3) it defines test data in terms of the architecture's interaction with the outside world that would cause this set of solutions to be generated.

We suggest the following family of architecture-based test criteria based on the CHAM model:

- *all-data-elements:* requires that all data defined in the architecture are communicated—for each data element d, at least one solution contains a molecule involving d;[2]
- *all-processing-elements:* requires that all processing elements are executed—for each processing element p, at least one solution contains a molecule involving p;

[2] Only one path in the test plan need contain such a solution.

- *all-connecting-elements:* requires that all communication channels and connections on them are exercised—for each connecting element c, at least one solution contains a molecule involving c;
- *all-transformations:* requires that all transformations are tested at least once—for each transformation rule $T : S_1 \longrightarrow S_2$, at least one path contains the reaction $S_1 \longrightarrow S_2$;
- *all-transformation-system:* requires that all distinct "paths", or non-repeating sequence of transformations from the initial solution to a stable solution, be tested—the test plan contains every non-repeating sequence of reactions; and
- *all-data-dependences:* requires that every sequence of interactions where a data element is output on a communication channel and is used either directly or indirectly as input to another communication channel be covered—for each such data dependence (d, d') at least one "path" contains a sequence of reactions covering (d, d').

We have begun to formally define these criteria based on the CHAM model, but the definitions are beyond the scope of this position paper. These are control and data flow criteria based on a CHAM architecture; as we continue this work, we may define others. We intend also to investigate how typical architectural defects might be reflected in the CHAM model and define fault-based test criteria accordingly. Moreover, the CHAM model also supports constructs (the membrane and airlock) to define modularity in complex architectures [8], for which we may also define test criteria.

These criteria vary in their comprehensive coverage of the architecture. Further analysis and experimentation is required to determine the relative comprehensiveness of these criteria. For a particular architecture and project, a testing strategy would select among the criteria by weighing not only the relative effectiveness of criteria, but also such factors as the resources available, time to delivery, and system criticality. Once the appropriate criteria have been chosen, applying the criteria to the CHAM architecture model defines the test cases of an architectural test plan.[3] This architecture-based, integration test plan can be used in several ways.

Most obviously, the test plan can be used to assess the architecture. Simulation capabilities have been developed for software architectures. A CHAM architecture simulation[4] would run the test cases over the architecture,[5] resulting in the set of solutions generated together with the causal history and timing. Causality between solutions results from the execution of transformation rules. These causal dependencies demonstrate the architectural behavior and may thereby reveal dynamic problems not easily revealed by static analyses. With this testing effort, different test criteria will target different architectural qualities; for instance, testing might assess the architecture's performance, load, or communication or identify missing functionality.

[3] The test criteria define test case inputs, or what is to be covered, but a test plan must also define expected behavior for the test cases as well as the environment in which testing is to occur.

[4] Without a simulator, a stubbed architecture could be implemented and tested prior to a full implementation effort.

[5] Actually, if the "path" to be covered in each test case is saved (see step 2 above), this may be used for simulation rather than the test data definition.

Another application of the architecture-based test plan is to evaluate the implementation for conformance to the architecture description. In this case, it is important to not only identify the aspects of the architecture that should be covered, but also to specify the expected behavior of the implementation. This requires that a test oracle be developed, which is a mechanism for checking execution results and comparing them to expected results [11]. For a CHAM-defined architecture, such an oracle would be a trace of the expected solutions and their transformations. Furthermore, using such an oracle requires a mapping between elements in the architecture model and those in the implemented system, since, for instance, they may use different names or be at different levels of abstraction.

Another testing activity that should be done while defining and analyzing the architecture is an assessment of, and improvement to, testability. Certain architectures are more difficult to test than others; thus, it may be better to re-architect before moving on in development. Moreover, testability can be enhanced if architecture test drivers are built into (or at least built concurrent with) the architecture to facilitate testing down the road.

4 Conclusion

In this paper, we argue that it is important to leverage current work in software architecture definition by developing test strategies based on architecture specifications. We have proposed several test criteria based on the Chemical Abstract Machine model. A testing strategy would specify which criteria to use to develop an architectural level integration test plan. The resulting plan can be used for testing the implementation for conformance to the architecture, with test oracles also derived from the CHAM. It may also be applied to evaluate the architecture itself, via either simulation or execution, to detect architectural problems revealed only by dynamic behavior. In another paper, Compare et al. [6] have also advocated a hybrid analysis strategy, combining algebraic and transition analysis, capable of detecting some architectural mismatch in dynamic behavior.

Related work in architecture-based conformance testing has been done in conjunction with the RAPIDE architecture definition language [9]. Our work has also been influenced by and is related to previous research in specification-based testing, Petri net testing, and protocol testing. We are formally defining the test criteria proposed here (as well as others) based on the CHAM model. We also plan to investigate testing based on other architecture description models and architecture definition languages and believe that our approach is generally applicable.

Testing at the architectural level has several benefits. For one, it enables focusing on architectural defects rather than relying on other testing strategies to detect these defects. Additionally, it facilitates detecting architectural defects earlier in the software lifecycle than after implementation and during system integration, as is typically done. Furthermore, since an architecture is often reused to develop multiple systems, the cost of any architecture level testing effort is amortized across the multiple systems. This leverages testing costs, which are extremely high relative to the rest of development.

References

[1] G.D. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, May 1994.

[3] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[4] J. Chang, D.J. Richardson, and S. Sankar. Structural Specification-based Testing with ADL. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 62–70. ACM SIGSOFT, January 1996.

[5] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[6] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. Available from the authors, October 1996.

[7] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific, New Jersey, 1993.

[8] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[9] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[10] L.J. Morell. A Theory of Fault-based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[11] O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient Specification-based Oracles for Critical Systems. In *Proceedings of the California Software Symposium*. Irvine Research Unit in Software, April 1996.

[12] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[13] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[14] D.J. Richardson and M.C. Thompson. The RELAY Model of Error Detection and its Application. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification (TAV2)*, pages 223–230. ACM SIGSOFT, July 1988.

[15] D.J. Richardson and M.C. Thompson. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[16] R.N. Taylor, C.D. Kelly, and D.L. Levine. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.