

Software Architectures and Integration Testing

Antonia Bertolino

Istituto di Elaborazione della Informazione, C.N.R.
via Santa Maria 46, I-56126 Pisa, Italy

Paola Inverardi & Henry Muccini

Dipartimento di Matematica Pura ed Applicata
Universita' di L'Aquila, via Vetoio, I-67010 L'Aquila, Italy

1 Introduction

In recent years the focus of software engineering is continuously moving towards systems of larger dimensions and complexity. Software production is becoming more and more involved with distributed applications running on heterogeneous networks, while emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality [20].

As a result, applications are increasingly being designed as sets of autonomous, decoupled components, promoting faster and cheaper system development based on COTS integration, and facilitating architectural changes required to cope with the dynamics of the underlying environment.

In this context Software Architecture (SA) can play a significant role. SAs have in the last years been considered, both by academia and software industries, as a means to improve the dependability of large complex software products and own concepts, formalisms, methods and tools have been introduced [13, 21, 12, 3, 6].

The software architecture of a large complex system supplies information about how the software is structured in parts and how those parts interact. Besides the static characteristics of a system, Software Architectures require the ability to describe dynamics at a *relevant* level of abstraction as well as the ability of expressing extra-functional requirements. As the complexity of software systems increases, the importance of Software Architectures becomes evident for the analysis, design and construction of the overall system structure. They represent the most promising approach to tackle the problem of scaling up in software engineering.

Testing is one of the most expensive activities in the development of complex software systems.

Traditionally, the testing of large software systems proceeds in subsequent steps following the well-known *V* model: unit, integration and system test, which correspond in bottom-up order to the code, design and specification stages of the development part of the life-cycle, (proceeding conversely in top-down fashion).

Nowadays, the emergence of new paradigms, centered on component-based assembly of systems, let suppose a software process in which unit testing plays a minor role, and testers have to focus more and more on how components work when plugged together.

However, while new models and methods have been proposed with respect to requirements analysis and design (notably UML [19]) and analysis of the consistency and

correctness of the SA as advocated in some works [6], how to approach the testing of these kinds of systems remains a neglected aspect and the following problems have to be solved:

- it is difficult to formalise and automate the integration testing stage, because the selection of test cases for the subcomponents and the order in which they are tested is entirely dependent on the adopted system functionality decomposition, which is hardly formalised. Integration test is thus normally devised ad hoc for each new system.
- The conventional approach to integration testing stresses structure over behavior: in fact, it is often explicitly stated that the goal of integration test is to check the compatibility between component interfaces. However, software components naturally embody assumptions about both the structure and the behavior of the contexts in which they are integrated. Besides conflicts between the information of static type present in the interfaces, when components are put to work together, conflicts can also arise in their dynamic interaction (*mismatch* [10, 11]). Conventional integration testing, by stressing the structural aspects of the decomposition, is not well suited to uncovering the latter.
- When existing components are re-used in the integration of the system, it is difficult to exploit previous test results.

Our research activity is directed to address these problems by using the software architecture of the system as the primary base. We intend to investigate how testing (in particular, integration testing) can be devised and controlled based on the software architecture. Indeed, since software architectures can provide a formal and explicit description of both static and dynamic properties of a system, as composed by components and connectors, the advantages can be manifold:

- the same design of an architecture could be used to steer the testing of the system during integration test. In general, the components and the connectors for a specific architecture are selected from a restricted collection (constrained by the adopted architectural style) and therefore the complexity of the testing task would be reduced because restricted to the style;
- the use of architectures should facilitate re-use of components. Consequently, regression test and re-use of test results of a component previously tested within other architectures should be facilitated as well;
- architectures typically specify information also about extra-functional properties of components and connectors. Therefore it should be possible to exploit the architecture specification to test for extra-functional requirements.

We will rely on descriptions of software architectures based on the Chemical Abstract Machine (Cham) formalism as introduced in [5, 14].

2 Cham Descriptions of Software Architectures

The peculiarity of Software Architecture descriptions in term of levels of abstraction and information content does not permit to straightforwardly export concepts, methods

and languages widely used in the specification of software systems to the description of Software Architectures [1, 14, 2].

Our attempt in using the Cham formalism for software architectural specifications is to propose an appropriate level of description by discriminating among details that have to be included in the description and those that have to be excluded. Moreover we are interested in expressing extra-functional constraints on the architecture to be refined. By extra-functional requirements we mean all those requirements that are meaningful for the customer but have no impact on the functional behaviour of the system. For example, requirements on the topology of the system architecture due to hardware or physical constraints or fault-tolerance requirements which may result in the need of replicating resources. From our perspective, the key issue here is in the ability of expressing these properties in the same formalism used for the specification of the functional/behavioural ones [21]. To this respect, we have experimented the use of the Cham formalism to describe the structural distribution of the components, i.e. the topological schema which has a *static* nature, and the issue of describing the (possibly different) interaction modalities among the various components, i.e. the *dynamic* structure [15, 9, 10].

In the following we briefly summarize the most relevant concepts the Cham model is based on.

A Chemical Abstract Machine specification contains *Molecules* m_1, m_2, \dots the basic elements of a Cham, and *solutions* S_0, S_1, \dots multisets of molecules interpreted as defining the *states* of a Cham; *transformation rules* T_1, T_2, \dots define a *transformation relation* $S_i \rightarrow S_j$ dictating the way solutions can evolve (i.e., states can change) in the Cham. Following the chemical metaphor, the term *reaction rule* is used interchangeably with the term *transformation rule*.

The transformation rules can be of two kinds: (four) general *laws* that are valid for all Chams and specific *rules* that depend on the particular Cham being specified.

The initial solution corresponds to the initial, static configuration of the system. We require the initial solution to contain molecules modeling the initial state of each component. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration.

A Cham description of a software architecture consists of a syntactic description of the static components of the architecture (the *molecule*) and of a set of reaction rules, beside the general ones, which describe how the system dynamically evolves through reaction steps.

The Cham formalism allows for two, quite different analysis techniques which permit to analyze and verify various properties of the system under specification including the usual *safety* and *liveness* properties. On the one hand, we can exploit the algebraic and equational nature of Cham. On the other hand, one can take advantage of this operational flavor to derive an LTS out of a Cham description whose node and arc labels represent respectively states and transitions relevant in the context of the SA dynamics.

3 The approach

In the following we describe the research directions we are currently working on to improve the software testing process. Our goal is to use the SA specification as a reference model to test the implemented system.

Although our approach builds on the Cham description of a SA it is worthwhile stressing that it is not committed to it. Our choice of the Cham formalism is dictated by our background and by its use in previous case studies. We are perfectly aware that other choices could be made and want to make clear that the use of a specific formalism is not central to our approach.

We identify interesting test classes for SA-based testing as sequences of interactions between SA components. More precisely, starting from an architectural description, carrying on both static and dynamic information, we first derive a *Labelled Transition System* (LTS), that graphically describes the SA dynamics. The problem is that the LTS provides a global, monolithic description of the set of all possible behaviors of the system. It is a tremendous amount of information flattened into a graph. It is quite hard for the software architect to single out from this global model relevant observations of system behavior that would be useful during validation.

We provide the software architect with a key to decipher the LTS dynamic model: the key is to use *abstract views* of the LTS, called ALTSs, on which he/she can easily visualize relevant behavioral patterns and identify those ones that are more meaningful for validation purposes. Test classes in our approach correspond to ALTS paths. However, once test class selection has been made, it is necessary to return to the LTS and retrieve the information that was hidden in the abstraction step, in order to identify LTS paths that are appropriate refinements of the selected ALTS paths.

Each selected LTS paths in our approach represents a test class for integration testing.

Lot of work has been devoted to testing concurrent and real-time systems, both specification driven and implementation based [7, 8, 16, 17, 18, 4] but all focus on unit testing.

Thus although the technical tools some of these approaches use are obviously the same of ours (e.g., LTS, abstractions, event sequences), their use in our context is different. This goal difference emerges from the very beginning of our approach: we work on an architectural description that drives our selection of the abstraction, i.e., the testing criterion, and of the paths, i.e., the actual test classes.

We definitely believe that the success of the approach described heavily depends on the availability of simple and appealing supporting tools. Our long term aim is to achieve a usable set of tools that would provide the necessary support to architecture-based testing. Our effort goes in two directions, on one side we are investing on automating our approach and we would also like to take advantage of other existing environments and possibly integrate with them, on the other we are involved in more experimentation. So far, we have developed the LTS generator starting from the Cham description, which also allows for keeping track of the state and arc labels. Work is ongoing to generalize it to ALTS generation and to implement a graphical front-end for Cham descriptions. Experimenting our approach is not an easy task, it requires the existence of a correct architectural description and a running implementation. This is obviously not often the case. The results we got so far for a real world case study are quite satisfactory and there are other real world case studies we are working on at the moment. We are confident these will provide other interesting insights to validate our approach.

References

- [1] R. Allen and D. Garlan. A Formal Approach to Software Architectures. In *Proceedings of the IFIP Congress*. Elsevier, September 1992.
- [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] S. Balsamo, P. Inverardi, C. Mangano, F. Russo. Performance Evaluation of a Software Architecture: A Case Study. *IEEE Proc. IWSSD-9*, April 1998, Ise-Shima, Japan.
- [4] A. Bechini, K.-C. Tai. Design of a Toolset for Dynamic Analysis of Concurrent Java Programs. *Proc. of 6-th Int. Workshop on Program Comprehension*, June 24-26, 1998, Ischia, Italy.
- [5] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.
- [6] A. Bertolino, F. Corradini, P. Inverardi, H. Muccini. Architectural Abstractions to Support Analysis and Testing. Internal Report 50/1999, University of L’Aquila, November 1999. Available at <http://univaq.it/inverard>.
- [7] R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs *IEEE Transactions on Software Engineering*, Vol. 24, Number 6, June 1998.
- [8] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4 (3), May 1978.
- [9] D. Compare and P. Inverardi. Modelling Interoperability by Cham: A case study. In *Proceeding of the First International Conference on Coordination Models, Languages and Applications*, April 1996, Lecture Notes in Computer Science, Springer-Verlag.
- [10] D. Compare, P. Inverardi and A. L. Wolf. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (33)2 (1999) pp. 101-131.
- [11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), November 1995.
- [12] D. Garlan and D.E. Perry. Special Issue on Software Architectures. *IEEE Transactions on Software Engineering*, 21(4), April 1995.
- [13] D. Garlan and M. Shaw. An Introduction on Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific, 1993.
- [14] P. Inverardi and A.L. Wolf. Formal Specifications and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):100–114, April 1995.

- [15] P. Inverardi and D. Yankelevich. Relating Cham descriptions of Software Architectures. *IEEE Proc. IWSSD-8*, March 1996, Padeborn.
- [16] G. Luo, G. Bochmann, A. Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-method. *IEEE Transactions on Software Engineering*, 20 (2): 149-162, 1994.
- [17] D. Mandrioli, S. Morasca, A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Trans. on Computer Systems* 13 (4), 1995.
- [18] Rachel Cardell-Oliver and Tim Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. *Springer Verlag LNCS 1486*, FTRTFTS98, Sep 98. Available at "<http://cswww.essex.ac.uk/>".
- [19] Rational Corporation. Uml Resource Center. UML documentation, version 1.3. Available from "<http://www.rational.com/uml/index.jtimpl>".
- [20] Clemens Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.
- [21] W. Tichy D. Garlan and F. Paulisch. Summary of the Dagstuhl Workshop on Software Architecture. *SIGSOFT Software Engineering Notes*, July 1995.