

Formal Specification and Validation of Dynamic Software Architectures*

Antonio Bucchiarone^{1,2}

¹ ISTI-CNR, Pisa, Italy.

`antonio.bucchiarone@isti.cnr.it`

² IMT Alti Studi Lucca, Italy.

Abstract. The principal characteristic of a Dynamic Software Architecture (DSA) is the ability to change its structure at run-time by adding or deleting components or connectors. I present here my research results and open issues on formal modeling and verifying dynamic software architectures.

1 Introduction

Modern software systems have changed from isolated static devices to highly interconnected machines that execute their tasks in a cooperative and coordinated manner. Therefore, the structure and the behavior of these systems are dynamic with continuous changes. These systems are known as *Global Computing Systems* (GCSs) and they use services as fundamental elements for developing them.

Software architectural models are intended to describe the structure and behavior of a system in terms of computational entities, their interactions and its composition patterns [29], so to reason about systems at more abstract level, disregarding implementation details. Since a GCS may change at run-time, Software Architecture (SA) models for them should be able to describe the changes of each system and to enact modifications during system execution. Such models are generally referred to as *Dynamic Software Architectures* (DSAs), to emphasize that the SA evolves during run-time. I present here some research results and open issues on formal development of dynamic software architectures. I present them by means of the *traffic light* process that will be described section 4 and that is depicted in Figure 1.

2 Dynamic Software Architectures

Since Global Computing systems and in general Service-oriented systems may change structure and behaviour at run-time, software architecture models for them should be able to describe the changes and to enact the modifications during the system execution. Such models are named *Dynamic Software Architecture (DSA)* [2, 3, 5, 20, 24] to emphasize that the system architecture evolves

* This work is supervised by Stefania Gnesi, `stefania.gnesi@isti.cnr.it`.

during runtime. In the last years several research papers and projects [8, 10, 17, 21, 28, 30] have has as main topics the dynamic system modeling and adaptation as well as providing new paradigms that extend the classic Software Architecture notations. For example Morrison et al in [25] define an *Active Architecture* as: *A software architecture that can evolve during execution by creating, deleting, reconfiguring and moving components, connectors and their configurations at runtime*. Chatlet et al. in [9] define *Plugin architectures* where components (i.e., Plugins) can be added (or deleted) to an existing system at runtime to extend (or reduce) its functionality, checking the preservation of properties. Bradbury et al. in [6] define a *Self-managing architecture* as: "an architecture that not only implements the change internally but also initiates, selects, and assesses the change itself without the assistance of an external user". Finally, Hirsch et al. in [17] propose the notion of *Mode* as a new element of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. Moreover, a variety of definitions of dynamicity for SAs have been proposed in literature. Below we list some of the most prominent definitions to show the variability of connotations that the word *dynamic* acquires.

- **Programmed** [14]: all admissible changes are defined prior to runtime and are triggered by the system itself;
- **Self-Repairing** [15]: changes are initiated and assessed internally, i.e., the runtime behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed;
- **Self-adaptive** [26]: systems can adapt to their environments by enacting runtime changes;
- **Ad-hoc** [14]: changes are initiated by the user as part of a software maintenance task, they are defined at run-time and are not known at design-time;
- **Constructible** [3]: it is a kind of ad-hoc mechanism but all architectural changes must be described in a given modification language, whose primitives constrain the admissible changes.

3 Research Contribution

When I started to work on this research topic there were many questions that arising. How can we represent these architectures? How can we formalise architectural styles? How can we construct style conformant architectures? How can we model software architecture reconfigurations? How can we ensure style consistency? How can we express and verify structural and behavioral architectural properties? The principal aspects that I have considered , trying to answer the previous questions are:

Uniform formal presentation of Dynamic SA Since that the different proposals for DSA are bound to particular language and models, my research is

aimed at understanding the main notions relying behind such proposals by abstracting away from particular languages and notations. I give a uniform formal representation that is abstract enough to cover most of these features. In this sense, this work is in the line of other previous research efforts [6, 32] and my representation of DSA as graph grammars is borrowed from the Le Métayer approach [22]. In particular I select Typed Graph Grammars (TGG) as a formal framework for mapping the different notions of dynamicity because (i) they provide both a formal basis and a graphical representation that is in line with the usual way architectures are represented, (ii) they provide a natural way of describing styles and configurations, (iii) they have been largely used for specifying architectures. The use of graph grammars is instrumental in comparing different mechanisms and better understanding the kinds of properties that can be naturally associated to such specifications.

Mechanisms to express and verify properties of a DSA In my research I am considering mechanisms to express and verify the properties that we expect to be satisfied by software architectures. In particular I consider

- **Structural properties:** that regard the topology of the architecture, i.e., cardinality of architectural elements and the way components are interconnected.
- **Behavioural properties:** that regard the behavior of each SA configuration, i.e., deadlock, liveness, responsiveness, reliability, etc.

For the above class of properties I have considered three analysis techniques:

- **Model Finding.** I consider the problem of analysing the state space of all possible architectures. Such analysis can serve as a computer-aided design process or as a debugging method to find out inconsistencies in the model or in its specification
- **Model Checking.** I consider the problem of verifying that a given architecture satisfies some structural or behavioural property expressed in a suitable logic.
- **Style Matching or Invariant Analysis.** I consider the problem of determining whether an architecture is conformant to a certain style or whether a reconfiguration is style preserving.

Definition of a Tool-supported process from DSA specification to code generation By combining different technologies and tools, I propose a SA-based approach aiming at combining exhaustive analysis techniques (Model-Finding and Model-Checking) and SA-based code generation to produce highly-dependable systems in a model-based development process. It is composed of three principal activities: (i) Formal Specification of DSAs, (ii) validation of each SA specification with respect to functional properties through Model-Checking, and finally (iii) architecture-based code generation.

The tool that I have used to implement the graph-based formal specification of a DSA is Alloy [18, 19]. Additionally, by using it, I show how to ensure style-consistency, perform model-finding and validate architectural structural properties after each SA reconfiguration. The tool that supports the second activity is UMC [31], an on-the-fly model checker for UCTL. It allows the efficient verification of UCTL formulae over a set of communicating UML state machines. Finally when each SA is validated with respect to the desired properties, Java code is automatically generated from the SA specification using ARCHJAVA[27].

My research objective was to propose an approach to design, verify and generate code of DSAs. In the next Section I present an overview of the "*Traffic Light*" process (shown in Figure 1) able to:

- (i) design formally structure, behaviour and evolution aspects of a DSA;
- (ii) formally verify structural and behavioral properties of each DSA;
- (iii) generate code automatically from the architectural models.

4 The Traffic Light Process

We have chosen the name *traffic light* for our process since that it is composed of three principal phases, each one represented by one color : **Red** for the DSA *structural design and analysis*, **Yellow** for the DSA *behavioral design and analysis* and **Green** for the automatic *code generation*. In the following we describe shortly the objectives of each of them.

4.1 DSA Structural Design and Analysis

The different proposals to design DSA are bound to particular language and models. In the **Red** phase we select graph grammars as a formal framework for mapping the different notions of dynamicity because (i) they provide both a formal basis and a graphical representation that is in line with the usual way architectures are represented, (ii) they allows for a natural way of describing styles, configurations and reconfigurations, (iii) they have been largely used for specifying architectures. In particular, we represent architectural styles by means of a *type hypergraph* and a set of constraints. The type hypergraph describes the types of components, connectors, ports and rules and their allowed connections. A configuration (i.e., a SA) compliant to a type hypergraph **T** is described by the notion of a *T-typed* hypergraph. Each SA is represented by a hypergraph where components (or connectors) are modeled using hyperedges and their ports (or roles) by the outgoing tentacles. Moreover, components and connectors are attached together connecting their respective tentacles to the same node. We represent reconfiguration of a DSA using rewriting rules among hypergraphs that state the possible ways in which a new configuration can be generated.

Tool Support The tool that supports the formal specification of a DSA is Alloy [19, 18]. Alloy provides a description language to represent software models, based on signatures and relations, that we found very suitable to model hypergraphs associated to DSAs. We have used it to implement formal aspects of Typed Graph Grammars (i.e., HyperGraphs, Partial and Total Morphisms, Matchings and SPO-based Rewriting) [12, 4]. Alloy also provides a logic, based on an extension of first-order logic with relational operators, to represent properties or constraints of the models. The *Alloy Analyzer* translates the model and the logical predicates into a (usually large) Boolean formula, uses efficient SAT solvers to decide satisfiability and provides a counterexample in the negative case. We have used it to show how to ensure style-consistency, perform model-finding and validate architectural structural properties.

4.2 DSA Behavioral Design and Analysis

Starting from the DSA designed in the previous phase, in the **Yellow** phase we validate the DSAs conformance to certain functional properties using Model Checking techniques. For each SA configuration, we associate to each component that compose it (i.e., each HyperEdge) a communicating UML state machine that describes the behavior of each of them. The complete behavioral specification is composed of a set of these UML state machines. After that we use the action- and state-based temporal logic UCTL [23] to describe behavioral properties that we want to check on our model. UCTL is composed of the action-based logic ACTL [13] and the state-based logic CTL [11]. This logic allows to specify the basic properties (i.e., deadlock, liveness and safety) that a run-time SA configuration should satisfy. Whenever the SA design is not properly specified (*not valid* arrows in Figure 1), the DSA itself needs to be revised. Thanks to the model checker we may correct the DSA specification. Whenever the DSA is validated (*valid* arrow in Figure 1) we can proceed to the **green** phase.

Tool Support The tool that supports this phase is UMC [31], an on-the-fly model checker for UCTL. It allows the efficient verification of UCTL formulae over a set of communicating UML state machines.

4.3 Code Generation

When the DSA is validated with respect to the desired properties (structural and behavioral), Java code is automatically generated. This activity is performed through two main steps: starting from a validated DSA design, ARCHJAVA code [1] is automatically obtained by means of a *JET-based Code Generator* [7]. Then, by exploiting the existing `ArchJava Compiler`, executable `Java Code` is generated. At this point we can stop our process.

Tool Support The language used in this phase is ARCHJAVA an Architectural Programming Language (APL) which extends the Java language with components classes (which describe objects that are part of the architecture), connections (which enable components' communication), and ports (which are the endpoints of connections). *Communication Integrity* is the key property enforced by ARCHJAVA ensuring that components can only communicate using connections and ownership domains that are explicitly declared in the architecture. ARCHJAVA guarantees communication integrity between an architecture and its implementation, even in the presence of advanced architectural features like runtime component creation and connection. A prototype compiler for ARCHJAVA is publicly available for download at the ARCHJAVA website³.

5 Conclusions and Future Work

At this point I have four principal open issues within my research and they may be described as follows:

- As kind of architectural dynamism, we have only considered the programmed dynamicity in which all admissible changes (e.g., adding and removing of components, connectors and connections) are defined prior to run-time and are triggered by the system itself. The immediate future research is to study how to extend the TGG approach for repairing and ad-hoc dynamism, proving properties associated to each of them. Related to this there is the necessity to use graph grammars with negative application conditions [16] in order to model productions that are equipped with a constraint about the context in which they can be applied. For instance, such conditions can state that the production is applicable only when certain nodes, edges, or subgraphs are not present in the graph.
- Hierarchical components are components that can be assembled together creating a more complex component. My objective will be to extend my approach to model and analyze Hierarchical DSAs. An idea can be to use Hierarchical Hypergraphs where each hyperedge can represent relations among components.
- Since I know in detail each phase of the *traffic light* process, my objective is to integrate them in a unique automated framework. For this we are developing an Eclipse-based framework named ARMADA (**A**utomated **R**e**M**orphing **A**mbient for **D**ynamic **A**rchitectures with the objective of facilitating DSA modeling and analysis within the software development life cycle.
- To validate the process, I have in mind the use of a more complex case study (e.g., from the Automotive field) described in the Sensoria project [28].

³ <http://archjava.org>

References

1. J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *International Conference on Software Engineering 2002*, 2002.
2. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.
3. J. Andersson. Issues in dynamic software architectures. In *Proceedings of ISAW4*, pages 111–114, June 2000.
4. P. Baldan, A. Corradini, and U. Montanari. Relating spo and dpo graph rewriting with petri nets having read, inhibitor and reset arcs. *Electr. Notes Theor. Comput. Sci.*, 127(2):5–28, 2005.
5. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proceedings of WICSA'04, 4th Working IEEE / IFIP Conference on Software Architecture*, pages 155–166, 2004.
6. J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of WOSS'04, 1st ACM SIGSOFT Workshop on Self-Managed Systems*, pages 28–33, 2004.
7. F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
8. A. T. S. Chan and S. N. Chuang. Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Trans. Software Eng.*, 29(12):1072–1085, 2003.
9. R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable dynamic plugin systems. In *FASE*, pages 129–143, 2004.
10. S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *ICAC*, pages 276–277, 2004.
11. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language Systems*, 8(2):244–263, 1986.
12. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.
13. R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419, 1990.
14. M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of BSCN'94*, pages 175–187, 1994.
15. D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of WOSS'02, First Workshop on Self-Healing Systems*, pages 27–32, 2002.
16. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.
17. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In *EWSA*, pages 113–126, 2006.
18. D. Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology*, 2002.
19. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
20. M. H. Kacem, M. J., A. H. Kacem, and K. Drira. Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In *ICEIS (3)*, pages 189–195, 2005.

21. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.
22. D. Le Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.
23. M. ter Beek and A. Fantechi and S. Gnesi and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'07), Berlin, Germany*, number 4916 in Lecture Notes in Computer Science, Berlin, 2008. Springer-Verlag.
24. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.
25. R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R. M. Greenwood. An active architecture approach to dynamic systems co-evolution. In *ECSA*, pages 2–10, 2007.
26. P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. In *Intelligent Systems, IEEE*, pages 54–62, 1999.
27. A. Project. Public web site, 2005. <http://archjava.org>.
28. Sensoria Project. Public web site. <http://sensoria.fast.de/>.
29. M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. In *Prentice Hall, NJ. USA*, 1996.
30. J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA*, pages 29–43, 2002.
31. UMC model checker. <http://fmt.isti.cnr.it/umc/>.
32. M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145(5):130–136, 1998.

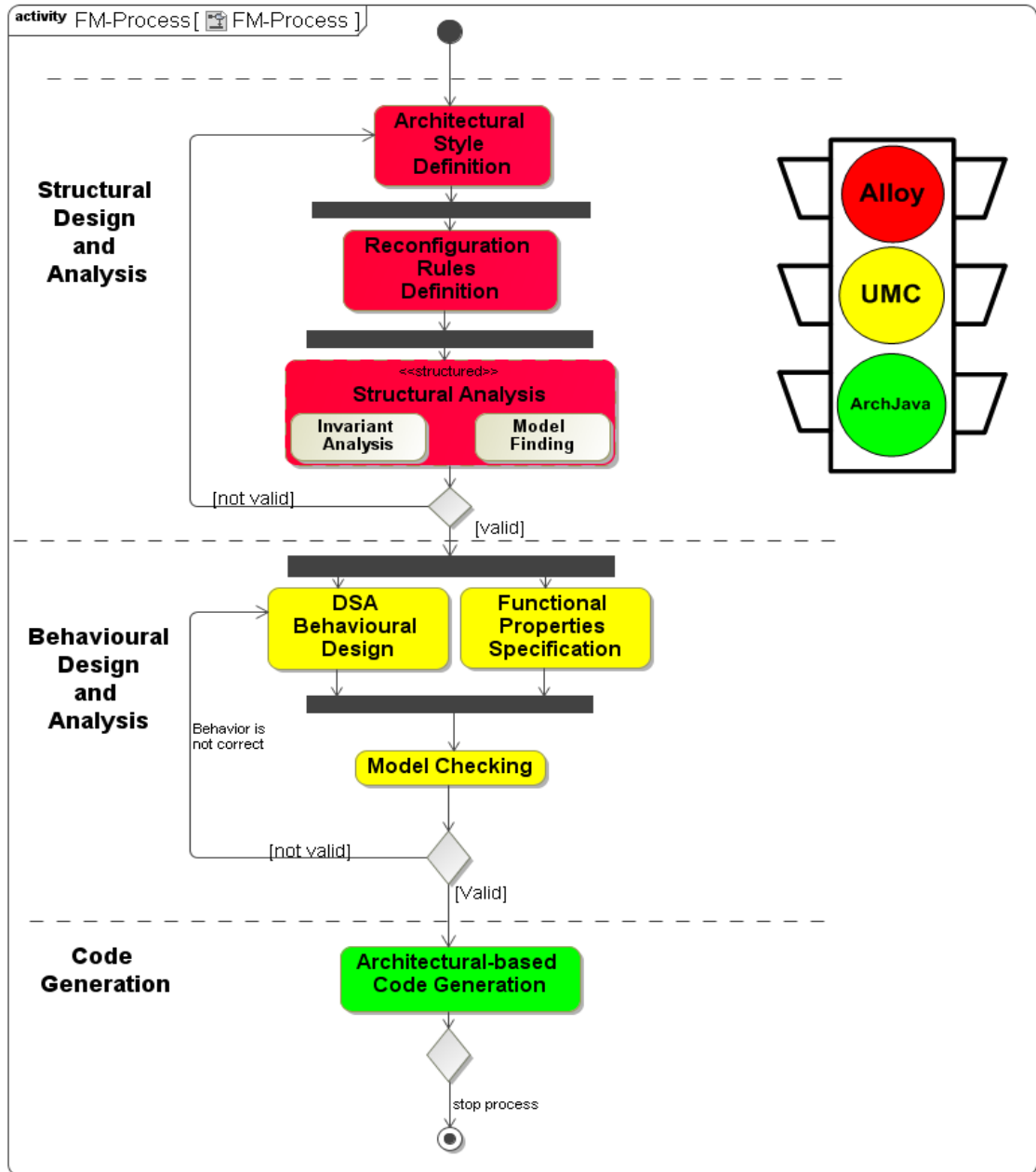


Fig. 1. The Traffic Light Process.