IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Dynamic Software Architectures for Global Computing Systems

PhD Program in Computer Science and Engineering

XX Cycle

Antonio Bucchiarone

2008

The dissertation of Antonio Bucchiarone is approved.

Program Coordinator:
Prof. Ugo Montanari ,
Computer Science Department, University of Pisa

Supervisor:
Dr. Stefania Gnesi,
ISTI-CNR, Pisa
stefania.gnesi@isti.cnr.it

Tutor:
Dr. Stefania Gnesi,
ISTI-CNR, Pisa

The dissertation of Antonio Bucchiarone has been reviewed by:

Dr. Reiko Heckel,
Department of Computer Science, University of Leicester
reiko@mcs.le.ac.uk

Dr. Sebastian Uchitel,
Computing Department, Imperial College London
s.uchitel@imperial.ac.uk

IMT Institute for Advanced Studies, Lucca

2008

To Francesca

Contents

List of Figures

List of Tables

Acknowledgements

My dears, these are the moments in which all of us stop for a moment and analyze our situation! As for my graduation, I'm at the end of a new experience of my life! I have to say that, making an analysis, the years of my PhD have been years in which life challenged me and at the same time it gave me a chance for making unique experiences. Everything started in September 2004, with my suitcase and the will to make new experiences. I found myself in Pisa and more precisely at the CNR! What can I say, I immediately found a friendly group able to make the days fun even when they weren't! This all thanks especially to people that with their own experience and will of making research built this all! They were the ones whom from the very first day believed in my potential and in my will to achieve! Thank you *Stefania*, thank you *Antonella*, because of you I found a family, and a work environment that was very stimulating and gratifying. The days at the ISTI were always amusing; we all always worked with the will of sharing this all during the break times! Thanks to all the components of the SE and FM&&T groups, and a particular thank you goes to *Eda*, *Andrea* and *Maurice* with whom I had the chance to relate more deeply and from which I have received so much, both professionally and more importantly, emotionally. Thanks also goes to my colleagues in the Pisa Department, thanks *Roberto*, thanks *Alberto* and thanks *Marzia*. I can't forget my colleagues in L'Aquila with whom I have been collaborating over these years. *Paola*, *Henry*, *Patrizio*, and *Massimo*. It was always a pleasure to collaborate with you all; you have always involved me in interesting work and always gave me useful advice to go ahead the best way. After my first period in Pisa, obtaining the PhD in Lucca led me to start a new experience! A new city, a new house, but especially new friends! It gave me the chance to meet great people with whom I shared the opportunity of acquiring my PhD and doing research! I thank all my friends in IMT, but a particular hug goes to *Luigi*, *Hernan*, *Manuela* and *Silvia*, the people with whom I shared unforgettable moments of my life! Our daily routine helped me comprehend the real meaning of the world friendship. I spent part of my last year in Lisbon, working on a research project at Nokia Siemens! It was a unique experience that I'd do again! Here again I found a group of very nice and friendly people who made the time fly! Thanks *Joao*, thanks *Madalena*, and thank you *Lui*, I'll always keep you in my heart! Finally, a special thank you to *Damien* whom has been a chat-friend in the hardest period of these last years. Thanks to our e-mails via Portugal-Germany, I had the chance to meet a very competent person whom was prepared for any challenge! Are we done yet? Did I forget someone? Here we go to the critical circle of the people whom made me live day by day and whom gave me the security and the comfort when I needed it! I'm talking about my family, a point of reference and an example! In those years I truly understood what it means to live without them, it's been a life test especially during the first period but it was especially a proof that if you love someone you love them even by the other side of the world! Thank you *mom*, thank you *dad*; with your attitude I always felt at home even if I was not there and thanks to my great siblings: *Giuseppe*, *Rita* and *Liliana* (and Kimba), you all are always in my thoughts! I want to dedicate my last sentences of this letter to a very special person. She is special because in the hardships of a long distance relationship, she's always been able to see the most important side of my journey. She's been by my side even if she was far away, and she's always encouraged my choices! *Francesca*, I dedicate this accomplish-

ment to you and I hope that the next one will be the most important one of our lives, the one which will relieve us of this distance that kept as apart during these years! You have been great as usual!

Salutations to you all and thanks from the bottom of my heart!

Antonio

Vita

November 3, 1977	Born, L'Aquila, Italy
April 2003	Master Degree in Computer Science University of L'Aquila, Italy
During 2004	Research Scholarship at Siemens C.N.X of L'Aquila, Italy.
From 2005	Collaborator at the FM&&T Labs of ISTI-CNR, Pisa involved in SENSORIA, MODCONTROL and TOCAI Projects.
October 2005	Laura Specialistica Degree in Computer Technologies University of Pisa, Italy
March 2005	PhD Admission at IMT of Lucca, Italy
February 2006	Professional Abilitation in Engineering , Section A (Information Field), University of Pisa, Italy
During 2007	Research Scholarship at Nokia Siemens Networks of Lisbon, involved in the SWARCES project.

Papers Included In the Thesis

1. **Formal Specification and Validation of Dynamic Software Architectures.**
A. Bucchiarone. Doctoral Symposium at 15th International Symposium on Formal Methods (FM'08), May 27 , 2008. Turku, Finland.
2. **Graph-Based Design and Analysis of Dynamic Software Architecture.**
R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch Lafuente. Concurrency, Graphs and Models, LNCS, Vol. 5065, pages 37-56, 2008.
3. **Dynamic Software Architectures Verification using DynAlloy.**
A. Bucchiarone and J. P. Galeotti. In Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08), To Appear.
4. **From Requirements to Java code: an Architecture-centric Approach for producing quality systems.**
A. Bucchiarone, D. di Ruscio, H. Muccini and P. Pelliccione. Chapter of the book Model-Driven Software Development: Integrating Quality Assurance edited by Joerg Rech (Fraunhofer IESE, Kaiserslautern, Germany) and Christian Bunse (International University, Bruchsal, Germany). IRM Press, CyberTech Publishing and Idea Group Reference imprints. 2008
5. **Modelling dynamic software architectures using typed graph grammars.**
R. Bruni, A. Bucchiarone, S. Gnesi, and H. Melgratti. In *Graph Transformation for Verification and Concurrency Workshop (GT-VC'07)*, Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 213, n. 1, pages 39-53, 2007.
6. **An architectural approach to the correct and automatic assembly of evolving component-based systems.**
P. Pelliccione, M. Tivoli, A. Bucchiarone, and A. Polini. *Journal of Systems and Software*, accepted on May 2008. To appear.
7. **Towards an architectural approach for the dynamic and automatic composition of software components.**
A. Bucchiarone, P. Pelliccione, A. Polini and M. Tivoli. In Proceedings of the 2nd International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA 2), held in conjunction with ISSTA2006 Portland, Maine, July 17th, 2006. Pages: 12-21. ISBN: 1-59593-459-6. Publisher: ACM Press, New York, NY, USA.

Papers Not Included In the Thesis

1. **Formal Methods for Service Composition.**
   M. H. ter Beek, A. Bucchiarone, and S. Gnesi. Annals of Mathematics, Computing and Teleinformatics 1, 5 (2007), 1-10.
2. **Testing Service Composition.**
   A. Bucchiarone, H. Melgratti and F. Severoni. In Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07) Mar del Plata, Argentina August 29-31, 2007.
3. **Web Service Composition Approaches: From Industrial Standards to Formal Methods.**
   A. Bucchiarone, M. ter Beek and S. Gnesi. In Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW'07), IEEE Computer Society , 2007.
4. **Architecting Fault-tolerant Component-based Systems: from requirements to testing.**
   A. Bucchiarone, P. Pelliccione and H. Muccini. In Proceedingsof the 2nd VODCA workshop on Views On Designing Complex Architectures, September 2006, Bertinoro, Italy. In ENTCS, Volume 168(2007), Pages 77-90 .
5. **A Practical Architecture-centric Analysis Process.**
   A. Bucchiarone, H. Muccini and P. Pelliccione. In the Second International Conference on Quality of Software Architectures, QoSA 2006, Vsters, Sweden, June 27-29, 2006. In Springer Verlag Lecture Notes in Computer Science (LNCS) Series, Volume 4214/2006.
6. **A New Quality Model for Natural Language Requirements Specification.**
   A. Bucchiarone, S. Gnesi, G. Lami, D.M. Berry and G. Trentanni. In Proceedings of the 12th International Working Conference on Requirements Engineering(REFSQ06), June 2006, Luxembourg, Essener Informatik Beitrage, ISBN 3-922602-26-6.
7. **Quality Analysis of NL Requirements: An Industrial Case Study.**
   A. Bucchiarone, S. Gnesi and P. Pierini. In Proceedings of the 13th IEEE International Requirements Engineering Conference, Paris, France. IEEE 0-7695-2425-7: 390-398.
8. **An Architecture-centric Approach for producing Quality Systems.**
   A. Bertolino, A. Bucchiarone, S. Gnesi, and H. Muccini. In Proceedings of the 1st International Conference on Quality of Software Architectures (QoSA 2005). LNCS 3712: 21-37.

Abstract

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, 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. Several recent research efforts have focused on the dynamic aspects of software architectures providing suitable models and techniques for handling the run-time modification of the structure of a system. A large number of heterogeneous proposals for addressing dynamic architectures at many different levels of abstractions have been provided, such as programmable, ad-hoc, self-healing and self-repairing among others. It is then important to have a clear picture of the relations among these proposals by formulating them into a uniform framework. When this work started there were many questions that arise. How can we represent 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? This thesis tries to answer them. In particular it presents a formal-based process that will be used to model and verify Software Architectures that are dynamic. The principal aspects that we have considered in this work are:

- formalisms used in the design of SA that are dynamic;
- mechanisms to express and verify structural and behavioral properties that we expect to be satisfied by each SA configuration;
- a complete tool-supported process able to integrate previous aspects.

These aspects are firstly illustrated over an explanatory example and then applied and validated over a real-world case study.

Chapter 1

Introduction

1.1 Global Computing Systems

Modern software systems have changed from isolated static devices to highly interconnected machines that execute their tasks in a cooperative and coordinate manner. Therefore, the structure and the behavior of these systems are dynamic with continuous changes. These systems are known as *Global Computing Systems (GCS)*, and have to deal with frequent changes of the network environment. Computing is not limited to a single ″computer″ but there are different types of devices (i.e., personal digital assistants (PDAs), mobile phones, laptops, etc..). The principal characteristics that these systems are summarized in the following:

- **Autonomy:** each GCS is composed of autonomous computational entities where activities are not centrally controlled, either because global control is impossible or impractical, or because the entities are created or controlled by different owners (i.e., Global Services).

- **Heterogeneity:** GCSs are composed of heterogeneous devices (i.e., PDAs, laptops, mobile phones, etc..). that provide different configurations and functionalities.

- **Mobility:** some computational entities are mobile, due to the movement of the physical platforms or by movement of entities from one platform to another.

- **User-Dependent:** the end-user of a GCS can be the source of some change and a GCS must be able to adapt itself to make the user's task easier.

- **Fault-Tolerance:** GCSs provide mechanisms to guarantee that faults in the system do not interrupt a service delivery. Usually these mechanisms are composed of two principal actions: ″error detection″ and subsequent ″system recovery or adaptation″. The run-time behavior of the system is monitored to determine whether a change is needed. In such a case, a reconfiguration is automatically performed without compromising current system execution.

- **Scalability:** GCSs are able to start small and then expand over time in terms of size (i.e. more number of users, devices and connections) and functionalities (i.e., new service request) insuring system availability.

The development of Global Computing Systems opens a great challenge: the range of devices, different infrastructures, context changes, user requests, etc. all introduce great complexity that demand a complete methodology to design and implement systems that are dynamic. When I started working in this research my first goal was to develop a methodology for these kind of systems. The result presented in this thesis is a SA-based process, which abstracts from any platform and has as an objective to furnish to all software architects an instrument for designing architectures that evolve by reacting to changes in requirements or constraints during run-time.

1.2 Architecture-Centric Development of GCSs

Based now on more than a decade of research in the field of Software Engineering, Software Architecture (SA) (Gar01) has today become an important part of software development processes (BCK03; HNS98; TMD08).

One of the most used definitions for software architectures is the following: *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, and relationships among them"* (Pre29). Researchers in industry and academia have integrated the Software Architecture description in their software development process. In current trends SA description is used for multiple purposes: while some companies use the SA description just as a documentation artifact, others make also use of SA specifications for enhanced analysis purposes, and finally many others use the architectural artifacts to (formally or informally) guide the design and coding process (BKP05; MWN$^+$04). However, putting SA in practice, software architects have learned that the SA production and management is, in general, an expensive task. Therefore the effort is justified if the SA artifacts are extensively used for multiple purposes helping on assuring the desired levels of dependability. Typical use of SA is as a high level design blueprint of the system to be used during the system development and later on for maintenance and reuse. At the same time, SA can be used by itself in order to analyze and validate architectural choices, both behavioural and quantitative. Thus, the problem of assuring as early as possible the correctness of a software system, occupies an ever increasing portion of the development cost and time budgets. Analysis techniques have been introduced to understand if the SA satisfies certain expected properties, therefore tools and architectural languages have been proposed in order to make specification and analysis rigorous and to help software architects in their work (e.g., (BI03)).

1.3 Dynamic Software Architectures

As previously stated, software architectural models are intented to describe the structure of a system in terms of computational components, their interactions, and its composition patterns (SG96), so to reason about systems at a more abstract level, disregarding implementation details. Since GCSs may change at run-time, software architecture models for GCSs should be able to describe the changes of the system (structure and behavior) and to enact the modifications during the system execution. Such models are referred as *Dynamic Software Architecture (*DSA*)* (ADG98; And00; L. 04; KJKD05; MT00), to emphasize that the system architecture evolves during runtime. In the last years several research papers and projects (CC03; CHG$^+$04; HKMU06; KM07; EU ; DHP02; SG02b), have had as main topics the dynamic system modelling and adaptation as well as providing new paradigma that extend the classic Software Architecture notations. For example Morrison et al in (MBO$^+$07) define an **Active Architecture** as: *"A software architecture that can evolve during execution by creating, deleting, reconfiguring and moving components, connectors and their configurations at runtime"*. Chatley et al. in (CEK$^+$04) 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 (BCDW04) 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 (HKMU06) 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 software architecture have been proposed in literature. Below I list some of the most prominent definitions to show the variability of connotations that the word *dynamic* acquires.

- **Programmed (End94):** all admissible changes are defined prior to runtime and are triggered by the system itself;

- **Self-Reparing (GS02):** 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 (OGT$^+$99):** systems can adapt to their environments by enacting runtime changes;

- **Ad-hoc (End94):** 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 (And00):** 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.

1.4 Contribution

During my PhD I mainly carried out researches to find effective solutions to specify and analyze DSAs. 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? This thesis tries to answer them. In particular it presents a formal-based process that will be used to model and verify Software Architectures that are dynamic.

The principal aspects that I have considered in this work are:

Uniform formal presentation of Dynamic SA Since that the different proposals for DSA are bound to particular language and models, this thesis 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 (BCDW04; Wer98) and my representation of DSA as graph grammars is borrowed from the Le Métayer approach (Le 98). In particular I 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 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 structural and behavioral properties of DSA In this thesis I consider 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 design to code generation In this thesis, 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 (Jac02; Jac06). 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 (UMC), 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 (Arc).

This process is firstly illustrated over an explanatory example and then it is applied and validated over a real-world case study.

1.5 Structure of the Thesis

This thesis is composed of four principal Chapters. Chapter 2 presents the State of the Art that covers aspects as Software Architecture Design, Analysis of SAs and SA-based code generation. It gives an overview of the solution proposed by other authors for problems similar or related to those that I studied during my PhD. In Chapter 3, I present my principal research results showing in detail the *traffic light* process. I have chosen this name because it is composed of three principal phases, each one represented by one color : `Red` for the *DSA formal design*, `Yellow` for the *DSA formal analysis* and `Green` for the *Architectural-based code generation*. It covers each aspect presented in Chapter 2 and gives details on each phase of the SA-based systems development: from DSA specification and validation to automatic code generation of each SA configuration. To have a clear idea of the process I use a "toy" running example and I present each technical aspect of my research presenting also tools used to support each step. In Chapter 4, in order to validate my results, I apply the process introduced before, to an Automotive Case study borrowed from the Sensoria Project (EU). The thesis ends with Chapter 5 in which I summarize the solutions proposed. A brief overview of possible directions for further researches is also discussed.

Chapter 2

State of the Art

2.1 Overview

Software applications for Global Computing Systems are composed of a number of software entities (i.e., components or connectors), distributed in the network, which cooperate in order to provide services to their users. This complexity is related to the great geographical distribution of the entities constituing the application, to the software heterogeneity and the evolutionary interaction between them. Quality of these systems has become a big issue, since failure can have economics consequences and can also endanger human life, we can think about software applications for aerospace domain, transportation and healt-care. Model-based specifications of component-based systems permit to explicitly model the structure and behaviour of components and their integration. In particular Software Architectures (SA) has been advocated as an effective means to produce quality systems. The architecture of a software system basically consists of the structure of components and of the way they are interconnected (SG96). Components are high-level computational and data entities that can range from a distributed application to a single thread, from databases to a simple data container (Szy02). Typical use of SA is as a high level design blueprint of the system to be used during the system development and later on for maintenance and reuse. At the same time,

SA can be used in order to analyze and validate architectural choices, both structural that behavioural. More recently, architectural artifacts have been used to implicitly or explicitly guide the design and coding process (ACN02). In summary, SA specifications are nowadays used for many purposes, like documenting, analysing, or to guide the design and coding process. In this thesis, as we have introduced in Chapter 1, we deal with software architectures that can change their structure during system execution. Typical changes, which are called *reconfigurations*, include components joining and leaving the system or changing their connections and are usually required for load balancing, fault-recovery and ridimensioning software systems (BCDW04; KJKD05; MK96; MT00). In this chapter and in the rest of the thesis we want to consider three main aspects in the development of systems that are dynamic. The first aspect to consider is the model used to design them. The second aspect is the set of mechanisms to express and verify structural and behavioural properties that we expect to be satisfied by software architectures after each reconfiguration. The third and last aspect is the code generation techniques from architectural specifications. Over the past 10 years, various alternative have been introduced to specify and analyze Dynamic Software Architectures, ranging from the more theoretical graph-based approaches (BHTV06; Le 98; WF02; HIM00) to implementation-oriented programming languages (APLs) such as ARCHJAVA(ACN02) or JAVA/A (BHH^{+}06; Hac04), passing through architectural description languages (ADLs) (MT00; KJKD05) and UML[1].
In this chapter we provide background informations on the state of the art on each of the following aspects: Dynamic Software Architecture design (Section 2.2), architectural-level analysis of DSAs (Section 2.3), and code generation from architectural design (Section 2.4) with the objective to analyze them respect to a set of characteristics that each DSA should provide. At the end of this chapter we analyze the state of the art presented here, in order to introduce better the main research results of the thesis.

[1]www.uml.org/

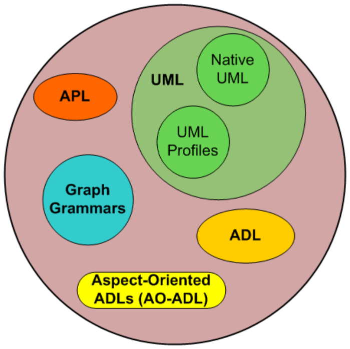


Figure 1: DSA Design Approaches

2.2 Dynamic Software Architecture Design

In this section, we report related works that address the design of DSAs. Recently many contributions have tried to consider the dynamic aspect of architectures by providing mechanisms for describing architectural reconfigurations. These can be categorized in a set as presented in Figure 1. Our objective is to understand if each element in the set provide the following peculiarity that each modeling approach should have to design DSAs. These characteristics are described in the following list:

- **Architectural Structure**: explicit specification mechanisms for architectural elements (components, connectors, ports, roles, interfaces, etc.) and architectural styles that constraint the construction of architectural configurations (pipeline, client-server, layered, multitier, peer-to-peer, etc.) (SG96).

- **Architectural Element Behavior**: ability to describe the behavior of each topology elements and of each Software Architecture configuration;

- **Architectural Reconfigurations:** ability to model Software Architecture evolutions (add/remove components and connections);

2.2.1 Architectural Description Languages (ADLs)

An ADL (MT00) is defined as a textual or graphical notation. It allows to specify Software Architectures(SAs) and it is generally accompanied with specific tools. Each ADL permits to define three essential concepts: *Component*, *Connector* and *Configuration*. *A component is a binary unit of independent production, acquisition, and deployment that interact to form a functioning system* (Szy02). The interaction among the components is represented by the notion of software "connector". Beyond the concepts of component and connector there is also another basic element that characterizes SAs, which is the system configuration. In other words, component and connectors can be composed together to make up different system configurations. Many ADLs have been proposed in the last fifteen years, with different requirements and notations, and with the objective to support components' and connectors' specification and their overall interconnection, composition, abstraction, reusability, configuration, heterogeneity, and analysis mechanisms (MT00). In this section we describe a set of ADLs that are able to design DSAs. Each ADLs is shortly introduced and at the end Table 1 summarizes each aspect presented and evaluated respect characteristics presented in the section above.
The Architecture Analysis and Design Language (**AADL**) (SAE) is based on MetaH language (LCV00). It allows separate component type and component implementation declarations. A component type declaration lists interface features while a component implementation declaration lists subcomponents, connections between subcomponents and behaviours of the components. The specification of the functional behaviour of components and subcomponents is made possible thanks to a new language that describes textually Mealy automata (FBF$^+$07). Another important feature is the possibility to model multiple run-time configurations of the system. Using *Modes* AADL is able to represent alternative operational states of a system or component. Each distinct configuration of a system is identified as a mode (state) within the modal state machine abstraction. The configuration that defines each mode and the events that cause the transitions in the behavior of the system must be specified. Finally AADL

offers a variety of capabilities to support architectural styles.
ACME (GMW97) is a general purpose architecture description language. Using Acme we can specify systems as graph of components and connectors in which the ports of a component fill the roles of a set of connectors to determine the interconnections topology. Each elements of this topology has a certain type and with the definition of structural properties we can specify as elements of those types may be legally composed. ACME does not support directly the behavioral specification of architectural elements, but since that is a general purpose ADL it can be used in synchronization with a set of behavioral specification approaches (i.e., process algebras, state machines, timed automata, etc.). In (BJC05), ACME has been extended with four new constructs more important to permit modeling of DSAs. The first is a *conditional* construct that allows the ADL programmer to express runtime conditions under which programmed reconfigurations should take place, together with a specification of what should change. The second extension is a pair of constructs that specify the destruction of existing ACME elements. The third and final extension is intended to express runtime dependencies between architectural elements.

ArchWare[2] (MBO$^+$07) is an ADL designed to model and verify *active software architectures* that are dynamic in the structure (e.g., cardinality of architectural elements, interconnection topology) and that can evolve at run-time. It is based on the high-order typed π-calculus (Mil99) and allows the verification of SAs using an analytical toolset based on theorem proving and model checking techniques.

With ArchWare we can specify SAs based on the concepts of components and connectors. Moreover, we can define architectural styles that are families of architectures with common structure and satisfy the same properties. With this language we can define three different kind of change in the architecture. *Dynamic change* allows the topology of components to be changed dynamically and the creation of new components and their interactions during execution. *Update change* allows component to be replaces. Finally, *Evolutionary change* allows the specification of the

[2]www.arch-ware.org

components and interactions to be changed during execution.

Darwin (MDEK95; MK96) allows the specification of distributed systems as a hierarchic construction of components. It views components in terms of both services they provide to allow other components to interact with them and the services they require to interact with other components. In general each component may provide and required many services and may be specified, implemented and tested independently of the rest of the system. Composite components and systems are specified in Darwin by declaring instances of components and binding the service required by one component to the services provided in another. Using Darwin we are able to specify architectures which change at runtime using *lazy* and *direct dynamic* instantiation. In the first each component, providing a service, is not instantiated until a user of that service attempts to access the service. In this way the SA structure can evolve according to a fixed pattern. Direct dynamic instantiation, instead, permits the definition of structures which can evolve in arbitrary way. Both static and dynamic aspects of Darwin have a precise operational specification in the π-calculus (MPW92).

Wright (ADG98) represents architectural structure as graph of components and connectors. Components have interfaces, which in Wright are called *ports*. Connectors also have interfaces, which are called *roles*. Each port defines a logically point of interaction with its environment and each role defines participant in the interaction also specifying the expected behavior of each participant in the interaction. Wright allows the user to formally specify behavior using CSP formalism (End94). To permit SA reconfigurations specification, Wright has been extended with two elements: (a) what events in the computation trigger a re-configuration, and (b) how the system should be reconfigured in response to a trigger. First, special "control" events have been introduced into a component's alphabet, and allowed to occur in port descriptions. In this way, the interface of a component can also describes when reconfigurations are permitted in each protocol in which it participates. Second, this control events are used in a separate view of the architecture, the configuration program, which describes how these events triggers reconfigurations. The formal semantics based on CSP allows to analyze SAs described in Wright respect to consistency and completeness properties.

LEDA (CPT99) is an ADL for the specification and validation of DSAs. The language is structured in two levels: *components*, representing system modules, and *roles*, which describe the observable behaviour of components. Roles are written in an extension of the π-calculus, thus allowing the specification of dynamic architectures. With LEDA composite components can be described and the relation among its subcomponents is expressed by a set of *attachments* or connection among the roles of these components. An other important aspect of LEDA is that it does not distinguish between components and connectors, nor between ports and roles in fact connectors are described in LEDA as specific classes of components, their behaviour being described by roles.

Olan (BBB$^+$98) is an ADL designed for applications that involve multiple users in a distributed environment. In Olan the definition of an architectural topology is basically made of the specification of the instantiation of components and their interconnection, i.e. the dependencies between required and provided services of their interfaces. The specification of components interconnection is realized using the implementation of a particular kind of components, the *composite*. A composite implementation contains the specification of a group of components in addition the communication between sub-components is also specified. Interconnections between components or sub-components is specified using the data flows, the execution flows, the communication protocol and the run-time mechanism used to perform the communication. The connector element is used to interconnect a set of components and Olan offers to the application architect a set of predefined connectors each of which corresponding to a communication pattern and an implementation on top of a middleware platform. Components and connectors do not allow to express their dynamic behavior but the language has been augmented with OCL (OMG03) enabling to describe operation changing the architecture. The obtained language is based on a set of reconfiguration rules of the form *Event*, *Condition* and *Action*. Moreover Olan does not offer the basis for analysing component behaviours.

xADL[3] (DvdHT01) is a highly extensible XML-based ADL that supports run-time and design-time modeling, architecture configuration management, and model-based system instantiation. It is based on xArch[4], a core representation for basic architectural elements. In xADL, two schemas accomplish the separation of run-time and design-time models.
The `INSTANCES` schema defines the core set of architectural constructs common to most ADLs (e.g., components, connectors, interfaces, links and sub-architectures). Constructs modeling design-time aspects of a system are defined in the `STRUCTURE and TYPES` schema. In addition the last schema provides also a type system for the architectural elements.

Aspects of elements like bahaviors and constraints on how elements may be arranged are not specified directly in xADL but an extension has been presented in (GMS05). Authors of this paper extend xADL with a Schema able to describe the abstract behavior of components and connectors. With this they are able to support performance and reliability modeling and analysis of SAs. The three most important aspects of modeling the dynamicity of architectures, in xADL, are defined in three different schemas: `Versions`, `Options` and `Variants`. Versions record information about the evoltion of architectures and elements (i.e., components, connectors, and interfaces). Options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of an element or group of elements. Finally, variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements.

Some ADLs support the dynamic reconfiguration of SA taking advantage of Aspect-Oriented Software Development (AOSD) techniques (FECA05). **AspectLEDA** (MPM07) is an extension of the ADL LEDA (CPT99) with primitives for describing Aspect-Oriented (AO) concepts. These aspect are usually used to describe SA reconfigurations. When we want to describe a new SA in AspectLEDA the first step is describing, using LEDA, the base SA that includes neither reference to aspect nor special primitives supporting aspects. Next, aspects are described in LEDA

[3]http://www.isr.uci.edu/projects/xarchuci/
[4]http://www.isr.uci.edu/projects/xarch/

as regular components. They are defined as architectural components and the interaction between the system element and the aspect is described in the *attachments* section. **PRISMA** (PACR06) is another Aspect-Oriented ADL. A PRISMA system is a component that includes a set of connectors, components and other systems that are correctly attached. PRISMA provides the evolution of aspect-oriented software architectures and their dynamic reconfiguration. PRISMA Architectures are defined at two different levels of abstraction: the type definition level and the configuration level. An architecture configuration is evolved by invoking an *evolution service* that update the number of architectural elements, the communication among them, and the structure of the architecture. Moreover, the original version of this language has been extended in (SAP$^+$07) with a *configuration* aspect that encapsulates every property and behavior related to SA dynamic reconfiguration. This aspect has a set of attributes that contain the current configuration of the system and a set of services that maintain and evolve this configuration. Any system that needs reconfiguration capabilities to evolve its SA imports this aspect. Regarding behavioral modeling, PRISMA use π-calculus process algebra.

Table 1: ADLs for DSAs

Name	**Structure**	**Behavior**	**Reconfiguration**
AADL	Components type Component implementation Style Definition	Mealy Automata	Modal State Machine
ACME	Components Connectors Ports, Roles Style Definition	No Directly	*destruction* constructs *conditional* construct
Archware	Components Connectors Style Definition	π-calculus	*Active* SA
Darwin	Components Services	π-calculus	Lazy Instantiation Direct Dynamic Instantiation
Wright	Components Connectors Ports, Roles	CSP	Event-based
LEDA/ASPECT-LEDA	Components Roles	π-calculus	Configuration Programs Aspect-Oriented Primitives
OLAN	Components Connectors Interfaces	NO	OCL-based
PRISMA	Components Connectors	π-calculus	Evolution Service
xADL/xARCH	XML Schemas	No Directly	Versions Options Variants

2.2.2 UML-based Approaches

The Unified Modelling Language (UML)[5] gives a set of elements to model component based software architecture. With the innovation introduced by UML 2.0 we are able to model composite structure diagrams with new component notation, port, required interface and provided interface notation. All these innovations focus on the modelling of static software aspects and for these reason UML remains inappropriate in modelling dynamic architectures. In fact, aspects like reconfiguration and architectural evolution, are not specifically deal in UML. In order to cover these weaknesses a set of researchers are working on. Researches to represent DSAs in UML (RKJ04; PMSA04; MRRR02) can be approached in two ways. The first consists in using the existing UML notations (MRT99). The second consists in the extension mechanisms in UML2.0 with profiles (MRRR02). Below we try to give an idea on principal way to use UML to design DSAs.

Kacem et al. in (MHKD06) propose a new UML2.0 profile for specifying dynamic software architectures. It integrates graph transformation and UML2.0 notations. The structural aspect of a SA is described using UML2.0 according to an architectural style while behavioral aspects are expressed using a new notation, based on UML2.0 and architectural rewriting rules (Le 98). The structural aspects of SA are graphically modeled using the component diagram. The dynamic aspects are defined by graph rewriting rules and finally a coordination protocol is defined as a partial order among reconfiguration operations. The coordination aspect models the dependency among the reconfiguration operations specified in the dynamic part. It represents how these operations must be managed in order to ensure the application evolution. This aspect is based on activity diagram. All the architectural properties are expressed in the OCL language (OMG03). In order to use this profile to model Patient Monitoring Systems, authors have developed a FUJABA [6] plugin that implements the proposed UML profile.

In (AB06) authors propose a UML profile to model context-aware ap-

[5]www.uml.org/

[6]http://wwwcs.uni-paderborn.de/cs/fujaba/

plications independently from the platform. This profile allows designers to specify the contexts that impact an application and the variability of an application architecture structure as well as its behavior according to this context. To define a new UML profile, authors distinguish three different types of adaptation: structural, behavioral and architectural. The first consists in extending the object's structure by for example adding or deleting methods or attributes to the objects; the second adapts the behavior of the applications' objects, the third consists in adding and deleting objects to an application according to the context. In order to model the three types of adaptation, authors have defined elements of the UML profile. UML class diagrams have been extended to support structural and architectural adaptation while sequence diagrams to support behavioral adaptation. To illustrate the approach an online shopping application is introduced but no design tool is used. Another limit of this proposal is that it does not introduce mechanisms to specify and verify architectural properties before and after adaptations. Other UML profile have been proposed for the modeling of a specific kind of dynamic architectures, where the dynamism is caused by the *mobility* of physical devices.

Grassi et al. in (GMS04) use Sequence Diagrams to model the interaction logic among components, and a Collaboration Diagram to model the interaction structure only. To model mobility, the collaboration diagram has been extended with the stereotype *moveTo*. Moreover, they model also physical mobility by Deployment Diagram. *Baumeister* et al. in (BKK$^+$03) present an extension to UML class, sequence and activity diagrams to model mobile systems. Class Diagrams are used to model the structure of the system and in order to model concepts as locations and mobile objects, they have been extended with a set of stereotypes. Sequence Diagrams models mobile, nested and dynamically changing structure by generalizing the concept of object lifeline. An alternative way to model mobility has been presented in the same paper using a variant of the activity diagrams. *Merseguer et al.* in (MCM00) propose to use State Diagram to model the internal behavior od each component of a software application, and Sequence Diagrams to model interaction scenarios among components. The modeling of component mobility simply consists in the addition at suitable points of its sequence diagram of a state whose dispatched action (*goto* action) moves that component to a different location. *Balsamo and Marzolla* in (BM03) propose to use Use Case and Activity Diagrams. A Use Case is used to express the possible preference of different mobility behaviors. The Activity Diagrams are used to represent both the effect of mobility on the system configuration and the internal activities of each system component. Each node of the activity diagram corresponds to a particular configuration and models mobility activities that leads to a configuration change. Service-Oriented Architecture (SOA) is one kind of dynamic software architecture. It is a component model that inter-relates different functional units of an application, called *services*, through well-defined interfaces and contracts between them (PG03). Baresi et al. in (BHTV06) propose an UML profile for SOAs where structural and dynamic aspects are defined by an extended component diagram. They use graph transformation rules to capture the dynamic aspects of a SOA. Each rule is defined as a pair of two instance graphs (i.e., using class diagrams) with the left-hand-side defining the pre-conditions and the right-hand-side defining the postconditions of the transformation. Both graphs represent a part of the configuration as an instance of the architectural style.

Table 2: UML-based approaches for DSA Design

References	**Structure**	**Behavior**	**Reconfiguration**
(MHKD06)	Component Diagram	NO	Graph Rewriting Rules OCL
(AB06)	Class Diagram	Sequence Diagram	Sequence Diagram
(GMS04)	Collaborative Diagram	Sequence Diagram	Collaborative Diagram Deployment Diagram
(BKK$^+$03)	Class Diagram	NO	Sequence Diagram Activity Diagram
(MCM00)	Class Diagram	State Diagram	Sequence Diagram
(BM03)	Use Case Diagram	Activity Diagram	Activity Diagram
(BHTV06)	Class Diagram	NO	Class Diagram Rewriting Rules

2.2.3 Graph-based Approaches

Several works have proposed graph grammars to design DSAs (CMR96; CMR$^+$97). In the following we present them showing in which way represent the same aspects used in the previous sections.

Baresi et al. (BHTV06) formalize architectural style as a *typed graph transformation system* (CMR96). It consists of a *type graph* to define architectural elements and their relationships, a set of *constraints* to further restrict the valid models, and a set of *graph transformation rules*. Nodes of the type graph define the architectural elements (i.e., components, connectors, ports, interfaces, etc..). Edges define the possible relationships among these elements. Moreover, a concrete architecture is an instance graph of the type graph. Reconfiguration mechanisms are modeled using transformation rules that can be applied to change the SA configuration. To do this authors apply a transformation rule to a host graph using the *Double-Pushout* semantics (CMR$^+$97). Authors address also how to ensure the consistency between architecture instances and the architectural style. They reformulate this problem as a *reachability problem* which is automatically solved by graph transformation or model checking tools. A future work of Baresi et al is to develop a integrated CASE environment for the analysis and stepwise refinement of SA. They are conducting experiments with existing graph transformation tools (i.e., AGG[7], PROGRES (SWZ99), Fujaba, GTXL[8], etc..) and model checkers (i.e., CheckVML (SV03), GROOVE (Ren03), etc..) with the final objective of a tool chain that seamlessly integrates the different components. The approach of Baresi et al. has been extended in (Thö05) to include behaviour modelling by encoding an activity diagram-like specification of local behaviour into the graph structure and providing rules interpreting them.

Le Métayer (Le 98) describes architectures by graphs and the architectural style by a context-free graph grammar. This notion of graphs is inspired by works on the chemical reaction model (BFM00; BM93) and set-theoretic graph rewriting (RV93). Nodes are used to represent computational en-

[7]tfs.cs.tu-berlin.de/agg/

[8]tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html

tities (i.e., client, server, etc..) while egdes correspond to the communication links between entities. The evolution of the architecture is defined by a *coordinator* that is expressed by conditional graph rewriting rules. Finally he uses static type checking to prove that the rewriting rules are consistent with the respective style but without tool support.
In the same paper he presents a small language to describe the behaviour of each entity. Its commands are very much in spirit of CSP (Hoa78) and its semantic is presented as a labelled transition system.
Wermelinger and Fiadeiro (WF02) describe SAs by diagrams in a category of programs using CommUnity (FM97), a parallel program design language. Each component of the SA is written in CommUnity with the usual notion of state and interacts with other components through synchronization and memory sharing. Architectural reconfiguration are represented as a rewriting process over graphs with nodes that represent program instances and edges that represent instance morphisms. A reconfiguration rule is a graph production, and a reconfiguration step is a direct derivation using a double-pushout transformation approach. This approach enforces that component state is only changed by computations and not by reconfiguration steps and proves that the graph obtained through direct derivation is well-typed respect to the style. Authors do not present some tool support.
Hirsch et al. in (HIM00) presents an approach for the specification of SAs styles using hyperedges replacements systems and for their dynamic reconfigurations using constraint solving. They represent SAs as graphs and Architectural Styles as graph grammars. Edges of each graph are components while nodes are ports of communication. The construction and dynamic evolution of the style are obtained as graph rewriting over the productions. To model evolution they have used graph rewriting combined with constraint solving in order to specify how components will evolve and communicate. Constraint productions are used to coordinate the dynamic evolution of the SA. Regarding verification aspects, they do not introduce particularly verification aspects and as previous related work is not tool supported. Intention of the authors is to implement the simulation of software architecture derivations and reconfigurations.

Table 3: Graph-based approaches for DSA Design

Reference	Structure	Behavior	Reconfiguration
(BHTV06) (Thö05)	Typed Graph	Activity Diagram	DPO
(Le 98)	Graph	CPS-like	*Coordinator*
(WF02)	CommUnity Program Graph Productions	NO	DPO
(HIM00)	Graph Graph Grammar	NO	Graph Rewriting Constraint Solving

2.3 Analysis of Dynamic Software Architectures

While how to model SAs has been for a long time the main issue in the SA community, how to select the right architecture has become one of the most relevant challenges in recent days. Model Checking, deadlock detection, testing, performance analysis, and security are, among others, the most investigated analysis techniques at the architectural level. Among the techniques that allow designers to perform exhaustive verification of the systems (such as theorem provers, term rewriting systems and proof checkers) model checking (E. 00) has as main advantage that it is completely automatic. The user provides a model of the system and a specification of the property to be checked on the system and the model checker provides either true, if the property is verified, or a counter example is always generated, if the property is not valid. The counter example is particularly important since is show a trace that leads the system to the error condition. While presenting a comprehensive analysis of the state of the art in architectural analysis is out of the scope of this section, it will focus on architecture-level Model-Checking techniques. For further reading on the topic, interested readers may refer to (BI03; DN02; Ros06). Initial approaches for Model Checking at the architectural level have been provided by the Wright architectural language (AG97) and the Tracta approach (MKG99). More recently, many other approaches have been proposed, as listed and classified in Figure 2. By focusing on the model-based approaches, Bose (Bos99) presents a method which automatically translates UML models of SA for verification and simulation using SPIN (Hol03). A component is specified in terms of port behaviours and performs the computation of provided services. A mediator component is specified in terms of roles and coordination policies. Safety properties are checked. Lfp (JB05) is a formal language dedicated to the description of distributed embedded systems control structure. It has characteristics of both ADL and coordination language. Its model checker engine is Maude (CDE$^+$07) based on rewriting logic semantics. Fujaba[9] is an approach, tool supported, for real-time Model Checking of component-based diagrams, the real-time behaviour is modelled by means of real-time statecharts (an extension to UML state diagrams), properties are specified in TCTL (Timed Computation Tree Logic) (ACD90) and the UPPAAL (UPPsala and AALborg University) (BLL$^+$95) model checker is used as real-time model checker engine. Arcade (BGH01) (Architecture Analysis Dynamic Environment) applies model checking to a DRA (Domain Reference Architecture) to provide analysts and developers with early feedback from safety and liveness evaluations during requirements management. The properties are represented as LTL formulae and the model checker engine is SPIN. AutoFOCUS (Aut) is a model-based tool for the development of reliable embedded systems. In AutoFOCUS, static and dynamic aspects of the system are modeled in four different views: structural view, interaction view, behavioral view, and data view. AutoFOCUS provides an integrated tool for modeling, simulation, and validation. CHARMY (Pel05; CHA; IMP05) is a proposal to model-check SA compliance to desired functional temporal properties. In CHARMY SA topology and behavior are described via UML based specifications and automatically translated into a formal prototype. In particular, components and state diagrams, are automatically interpreted to synthesize a formal Promela prototype, which is the SPIN model checker modeling language. Moreover CHARMY provides support for simulating the SA: it uses the SPIN simulation engine and offers simulation features which

[9]www.cs.uni-paderborn.de/cs/fujaba/index.html

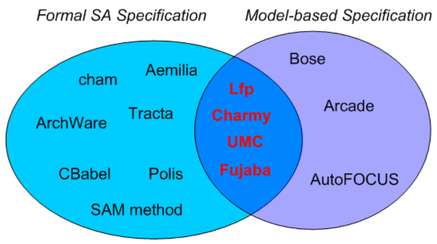


Figure 2: DSA Verification Techniques

interpret SPIN results in terms of CHARMY state machines. Properties whose validity need to be checked on the architectural model are modeled through scenarios, by expressing desired and undesired behaviors. Such scenarios are automatically translated into Büchi automata (B6̈0), an operational representation for LTL formulae. SPIN is then used to check the conformance of Promela prototype with respect to such behavioral properties. CHARMY has been used to model and verify fault-tolerant and Telecommunication systems (BMP06; BMP07). UMC (UML on the fly Model Checker) (UMC) is the model-checker used in this thesis to verify the conformance of a SA design respect to desired properties. UMC takes in input a set of statechart diagram descriptions (describing the dynamic behavior of the components of the SA) and verifies a set of correctness properties formalized in the action- and state-based temporal logic UCTL (tFGM08). The algorithm implemented in UMC is able to check the validity of a formula without generating the global model of the system bypassing the state explosion problem that makes verification tools inapplicable. More details will be provided in Section 3.3 when we will present the complete proposed process.

2.4 Architectural-based Code Generation

In this section we present an overview on the various techniques used to generate code from a SA specification. We focus the attention on languages that can be used to generate code from a high-level description

Table 4: Code Generation from ADLs

ADL	Data Born	Tool Support	Output Code	Reference
Darwin	1991	LTSA+SAA	C++	(MDEK95; MK96)
Fujaba	1997	Fujaba	Java	(Fuj)
AADL	2001	Osate	Ada, C, C++	(SAE)
Prisma	2002	PrismaCase	C♯	(PACR06)

of the SA. They can be distinguished in Architecture Description Languages (ADLs), such as languages for describing SAs, and Architectural Programming Languages (APLs), such as languages that integrate SA concepts into programming languages. We conclude with a comparison among APLs. It is important to note that code generated from ADLs not necessarily contains architecture concepts. This can have impact on the readability of the code and can reduce its modifiability and maintainability. Furthermore, modifications on the generated code made by developers can invalidate architectural constraints. APLs have been introduced to solve this problem. All these aspects will be detailed in the following.

2.4.1 ADLs-based code generation

Some ADLs support code generation from an architectural description of the system. Table 4 lists the ADLs that support code generation: it shows the ADL name, the tool support and the type of code that they produce as output. The table presents ADLs that are currently used in an industrial context and that are continuously updated showing the last release and the references.

However, the implementation step is, at the best, only supported by code generation facilities not capable of explicitly representing architectural notions at the code level. Thus, the notion of SA components, connectors and configurations is kept implicit and the implementation inevitably tends to loose its connection to the intended architectural structure during the maintenance steps. The result is *"architectural erosion"*

(PW92).

2.4.2 Architectural Programming Languages (APLs)

APLs overcome the problem of architectural erosion in implementation by integrating SA concepts into programming languages. With APLs, there is an inclusion of architectural notions, like components, ports with provided and required interfaces as well as protocols and connectors, into a programming language (typically Java). The basic idea of architectural programming is to preserve the SA structure and properties throughout the software development process so to guarantee that each component in the implementation may only communicate directly with the components to which it is connected in the architecture. This aspect is called *Communication Integrity* between code and SA. In this section ArchJava and Java / A will be presented, which are the most advanced APLs (BHH$^+$06), in order to understand their main characteristics and to compare them with respect to aspects that are important for an APL.

ArchJava

ArchJava(ACN02) is an APL which extends the Java language with component classes (which describe objects that are part of the architecture), connections (which enable components communication), and ports (which are the endpoints of connections). Components are organized into a hierarchy using ownership domains, which can be shared along connections, permitting the connected components to communicate through shared data. A component in ArchJava is a special kind of object whose communication patterns are explicitly declared using architectural declarations. Component code is defined in ArchJava using *component classes*. Components communicate through explicitly declared ports. A *port* is a communication endpoint declared by a component. Each port declares a set of required and provided methods. A provided method is implemented by the component and is available to be called by other components connected to this port. Conversely, each required method is provided by some other component connected to this port. Each provided method must be implemented inside the component. ArchJava requires developers to declare connection patterns that are permitted at run-time. Once connect patterns have been declared, concrete connections can be made between components. All connected components must be part of an ownership domain declared by the component making the connection. *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 run-time component creation and connection. A prototype compiler for ArchJava is publicly available for download at the ArchJava web site[10].

Example We illustrate ArchJava through a simple example. Figure 3 shows a UML composite component diagram of a toy example[11].

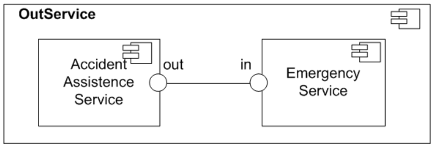


Figure 3: OutService Composite Component in ArchJava.

```
public component class OutService {
protected owned AccidentAssistanceService aas = ...;
protected owned EmergencyService es = ...;

connect pattern AccidentAssistanceService.out, EmergencyService.in;

public OutService () {
     connect (aas.out, es.in);
       }
}

public component class EmergencyService {
```

[10]http://archjava.org

[11]Borrowed from Sensoria, Research supported by the EU within the FET-GC2 IST-2005-16004 Integrated Project Sensoria (Software Engineering for Service-Oriented Overlay Computers).

```
public port in {
     provides void AlertEmergencyService (int loc);
     provides void EmergencyLevel (int level);
     provides void AlertAccepted ();
}
public void AlertEmergencyService (int loc) {
     ...
}
public void EmergencyLevel (int level) {
     ...
}
public void AlertAccepted () {
     ...
}
   }

public component class AccidentAssistanceService {
public port out {
      requires void AlerEmergencyService (int loc);
      requires void EmergencyLevel (int level);
      requires void AlertAccepted ();
   }
}
```

The *OutService* component is made up of two subcomponents: the *AccidentAssistanceService* (AAS) and the *EmergencyService* (ES). The first has one *out* port and the second one *in* port through which the two components are connected. A port is a communication endpoint declared by a component. For each port the language provides constructs to define *requires* and *provides* methods. ARCHJAVA requires developers to declare in the architecture the connection patterns that are permitted at run-time. Taking a look to the code for the specification in Figure 3, the declaration *"connect pattern"* in the code permits the *OutService* component to make connections between the *out* port of its AAS subcomponents instance to the *in* port of the ES component instance. This connection binds the required methods (*AlertAccepted*, *AlertEmergencyService*, etc.) in the *out* port of the AAS to a provided method with the same name and signature in the *in* port of the ES component. Thus when AAS invokes *AlertAccepted* on its *out* port, the corresponding implementation in ES will be invoked.

Java/A

The basic idea of JAVA/A (BHH$^+$06; Hac04) (as in ARCHJAVA) is to integrate architectural concepts, such as components, ports and connectors, as fundamental parts into Java. The underlying component model is compatible with the UML component model (Hac04). This compatibility and the one-to-one mapping of these concepts allows software designers to easily implement UML 2.0 component diagrams. They can express the notions present in these diagrams using built-in language concepts constructs of Java. Furthermore, the visibility of architectural elements in the JAVA/A source code prevents architectural erosion. The basic concepts of the JAVA/A component model are *components*, *ports*, *connectors* and *configurations*. Any communication between JAVA/A components is performed by sending messages to ports. This message must be an element of the required interface of the perspective port. The port will then pass on the message to the attached connector, which itself will delegate the message to the port at its other end. Each port may contain a *protocol*. These protocols describe the order of messages that are allowed to be sent from and to the respective port. Any incoming and outgoing communication must conform to the protocol. Protocols are realised by UML state machines and ensure the soundness of a configuration at compile-time. A *Connector* in JAVA/A links two components by connecting ports they own. The JAVA/A compiler is not yet complete and available but authors claim that it will transform JAVA/A components into pure Java code which can be compiled to byte code using the Java compiler. It will be possible to compile and deploy each component on its own, since the component's dependencies on the environment are encapsulated in ports. The correctness of an assembly (i.e., deadlock-freedom) can be ensured using the UML state machine model checker HUGO (HUG05). Another important aspect that JAVA/A has is the *dynamic reconfiguration*. It summarises changes to a component-based systems at run-time, concerning creation and removing connections between ports. JAVA/Asupports each of these reconfiguration variants. JAVA/Ahas a semantic model that uses a states as algebras approach (BHH^{+}06) for representing the internals of components and assemblies, and the I/O-transition systems for describing the observable behavior.

Example Figure 4 shows a composite component diagram of the same system already introduced for ArchJava (in Figure 3).

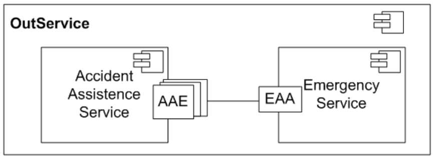


Figure 4: OutService Composite Component in Java/A.

The composite component contains an assembly of two components *Accident Assistance Service (AAS)* and *Emergency Service (ES)* whose ports are wired by a connector. The AAE port of the AAS component is depicted as stacked boxes since it is a *dynamic port* which can have an arbitrary number of port instances. In contrast, the *static port* EAA must have a single instance at any time. Port protocols are specified with UML state machines. A protocol describes the order and dependencies of messages which are sent and received by a port. The code corresponding to this specification is described below.

```
simple component AccidentAssistanceService {
dynamic port AAE {
  provided {
   void AlertAccepted();
   void AlertNoAccepted();
   void EmergencyAccepted();
   void EmergencyNotAccepted();
     }
  required {
   signal AlertEmergencyService (Location Loc);
   signal EmergencyLevel(int Level);
   void AlertAccepted();
     }

  try {
   Component aas = ComponentLookUp (this, "AccidentAssistanceService");
   Port aae = aas.getPort ("AAE");
   ConnectionRequest cr = (this, this, EAA, aas, aae, new Connector());
   reconfigurationRequest(cr);
    }
  Catch (ReconfigurationException e) {...}
 }

simple component EmergencyService {
port EAA {
  provided {
    signal AlertEmergencyService(Location Loc);
```

```
        signal EmergencyLevel(int Level);
        void AlertAccepted();
      }
16.   required {
        void AlertAccepted();
        void AlertNoAccepted();
        void EmergencyAccepted();
        void EmergencyNotAccepted();
17.   }
18.   <! // protocol of EAA
         states {
                  initial Initial;
                  simple Q1,Q2,Q3,Q4;
                }
         transitions {
          Initial -> Q1;
          Q1->Q2 {trigger AlertEmergencyService();}
          Q2->Q1 {effect AlertNoAccepted();}
          Q2->Q3 {effect AlertAccepted();}
          Q3->Q4 {trigger EmergencyLevel();}
          Q4->Q3 {effect EmergencyNotAccepted();}
          Q4->Q1 {effect EmergencyAccepted();}
          }
   !>
19. }

20.  composite component OutService
21.  {
22.    assembly {
          component types {AccidentAssistenceService,EmergencyService}
23.          connector types {
                          AccidentAssistenceService.AAE;
                          EmergencyService.EAA;
                          }
24.             initial configuration {
                          AccidentAssistenceService AS =
                              new AccidentAssistenceService();
                          EmergencyService ambulance =
                              new EmergencyService();
                          EmergencyService police =
                              new EmergencyService();
                          Connector cn0 = new Connector();
                          cn0.connect(ambulance.EAA, AS.AAE);
                          Connector cn1 = new Connector();
                          cn1.connect = (police.EAA, AS.AAE);
                          }
                  }
      }
25.  }
```

In lines 1-12 and 13-19 the two simple components (Accident Assistance Service (AAS) and Emergency Service (ES)) are declared while in lines 20-25 a composite component "OutService" is declared as an assembly of the two previous one. In lines 2 and 14, the ports AAE and EAA are defined. Each port declaration contains a set of provided operations (i.e., lines 3 and 15) and a set of required operations (i.e., lines 4 and 16). Port protocols are specified by UML state machines which are textually represented using the notation UTE (HUG05). For instance, lines 18-19 show the UTE representation of the UML state machine for the port EAA. In line 24 a possible configuration of the *OutService* composite component is declared. It presents two instances of the ES component (ambulance and police) that are attached at the AAS by the EAA and AAE ports. The last interesting aspect the JAVA/A can models is the *dynamic reconfiguration* that describes changes to a component-based system at run-time, concerning creation and destruction of components and building up and removing connections between ports. This is made with a code like to lines 6-11 where a possible reconfiguration in the OutService composite component (i.e., the connection and disconnection of ES) is presented. An idle ES disconnects from the AAS and reconnects whenever there is an accident and the AAS alerts the ES. When AAS alerts the ES executes the code in the 6-11 lines which realized the (re)connection of an ES to the AAS.

A Comparison

ARCHJAVAand JAVA/A employ similar approaches. Both augment Java with the concepts of component and connector. ARCHJAVAcomponents have ports with required and provided interfaces. However, ports in ARCHJAVA do not have associated protocols. As a result the dynamic behavior of ports is not capturated in ARCHJAVA. ARCHJAVAas well as JAVA/A allows hierarchical component composition. In JAVA/A there is no possibility of communicating with components other then sending messages to their ports, whereas in ARCHJAVA outer components can invoke methods of inner components directly, which breaks the encapsulation. While ARCHJAVA lacks a semantic model, JAVA/A provides a complete one based on algebras and I/O-transitions systems. As far as concern tool support, in (SG04) the authors have developed additional Eclipse plug-ins that integrates AcmeStudio (Acm) and ARCHJAVA. With this framework an architect can model an architecture using AcmeStudio, and have access to AcmeStudio's verification engines to check desired ar-

Table 5: APLs Comparison 1

APL	Components	Ports	Configurations	Encapsulation
ARCHJAVA	Yes	Yes	Implicit	Partial
JAVA/A	Yes	Yes	Explicit	Yes

Table 6: APLs Comparison 2

APL	Behavioral Modeling	Tool Support
ARCHJAVA	No	Total
JAVA/A	Yes	Not yet

chitectural properties. The architect can then generate ARCHJAVA code using the refinement plug-in. As developers complete the implementation to provide the functionality of the system, ARCHJAVA 's checks help ensuring that the implementation conforms to the architect's design. Unfortunately the existing ARCHJAVA environment supports only the verification of architectural properties and it does not force the developers to respect the component behavior described into the SA. For JAVA/A the tool support is not yet complete and it is one of the future work. So far, a JAVA/A compiler should transform JAVA/A components into pure Java code which can be compiled to byte code using the Java compiler. However, this compiler is not yet publicly available. Tables 5 and 6 synthesize the above discussion and way of understanding the key features and differences of ARCHJAVA and JAVA/A.

2.5 Conclusions and Research Proposal

In this Chapter we have presented related works with the objective to evaluate the ability of current approaches to design, verify and realize dynamic software architectures. For sure these not cover all works in the literature but gives and idea on the different research directions and proposals and has been our starting point of this work. For each design approach we have analyzed the ability to define aspects as structural elements and constraints (components, connectors, ports, architectural styles, etc.), behavior of each structural element and the possibility to design systems in which its structure can evolve by inserting/deleting topology elements. This chapter shows that area of dynamic software architecture specification is well researched but exist a lot of different sometimes conflicting notations, concepts, and definitions. To model DSAs we have presented different approaches, ADLs, UML and Graphs. ADLs are informal or formal notations that do not present friendly mechanisms to be used by software developers. Most of them address either structural or behavioural properties, but not both and this makes difficult rigorous analysis and verification of architectural properties. UML is a more user-friendly notation, and it is today more used from both the industrial and academic perspective. However, it does not specifically address the modeling of DSAs and does not have native formal method supports. Graphs expresses a natural way to specify dynamic software architectures using graph rewriting rules but it lacks in the behavioral modeling and analysis. Another aspect that we have considered is the ability to generate code directly from the Software Architecture design. We have presented two principal way realize it, one based on ADL design and one based on APL design. APLs respect to ADLs are more useful since that they overcome the problem of architectural erosion in implementation by integrating SA concepts into programming languages. Using them we do not loose architectural structural properties during each SA reconfiguration. After the previous evaluation we can summarize the ability of current approaches to represent and validate dynamic software architectures. For each of them we have considered the ability to design, verify and generate code. The outcome is summarized in Table 7.

Each approach in Table 7 not cover totally all aspects. Our research objective is to propose an approach to design, verify and realize DSAs. We want to propose a complete process able to:

(i) design structure, behaviour and reconfiguration aspect in a formal way;

Approach	Design		Formal Verification	Code Generation
	Structure	Behaviour		
ADLs	+	−	±	±
UML	+	+	−	±
APLs	+	±	±	±
Graph grammars	+	±	+	−

Table 7: Summary of approaches to DSAs

(ii) formally verify structural and behavioral properties of each DSA;

(iii) generate code automatically from the architectural models.

Other two features of this process must be:

(iv) the generated code must respect both structural (i.e., each component can only communicate using connectors and ownership domains that are explicitly declared in the DSA) and behavioral constraints (i.e., methods provided by components can be invoked only consistently to the behaviours defined for the components);

(v) the approach must be supported by automated tools, which allow formal design and analysis and permits code generation from the validated architecture.

In the next section we introduce an overview of our process while in the next chapter we present the main results of the thesis in detail, relating them to each step of the following process.

2.6 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.

2.6.1 Red Phase: 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 (Jac06; Jac02). 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.

Why Alloy? AGG[12], PROGRES[13], Fujaba(Fuj), CheckVML(SV03) and GROOVE[14] are existing tools for graph transformation-based modeling. All of them allow to design typed and attributed graphs. Full support of cardinality constraints, including automatic constraint checking, is only

[12]http://tfs.cs.tu-berlin.de/agg/

[13]www-i3.informatik.rwth-aachen.de/research/projects/progres/

[14]groove.sourceforge.net

provided by AGG. Except for CheckVML, which is intended to translate graph transformation models into the input language of model checkers, all the tools allow the execution of graph transformation rules, even with negative application conditions. Their limits are in the possibility of designing Typed HyperGraphs. The unique one that support Typed HyperGraph design is *Graph eXchange Language* (GXL) but it lacks in verification aspects. We use Alloy to implement formal aspects of Typed Graph Grammars (i.e., HyperGraphs, Partial and Total Morphisms, Matchings and SPO-based Rewriting) (CMR$^+$97; BCM05). 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. Positive and negative characteristics of Alloy are:

+ It is based on a simple notation with a simple relational semantics

+ It is easier to learn and use for developers without a strong formal background

+ It offers a completely automated SAT based analysis mechanism (no manual manipulations are necessary)

- It searches for a counterexample up to certain bound k in the number of elements of the model.

2.6.2 Yellow Phase: Behavioural 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 (tFGM08) to describe behavioral properties that we want to check on our model. UCTL is composed of the action-based logic ACTL (DV90) and the state-based logic CTL (CES86). 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 5), 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 5) we can proceed to the `green` phase.

Tool Support

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

Why UMC? I have chosen to use UMC since that it is our in-house model checker and it allows the efficient verification of functional correctness properties formalized in the action-based and state-based branching-time temporal logic UCTL (tFGM08) over a set of communicating UML state machines (describing the SA components' dynamic behaviour). UMC uses an on-the-fly model-checking algorithm with a linear complexity, which has as advantage that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result.

2.6.3 Green Phase: Code Generation

After the DSA has been validated w.r.t. the desired properties (both structural and behavioural), Java code is automatically generated in the `green` phase. This activity is performed through two main steps: starting from a validated DSA design, ArchJava code (ACN02) is automatically obtained by means of a *JET-based Code Generator* (BBM03). Then, by exploiting the existing `ArchJava Compiler`, executable `Java Code` is generated.

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 run-time component creation and connection. A prototype compiler for ArchJava is publicly available for download at the ArchJava website[15].

Why ArchJava? The most famous and advanced APLs are ArchJava and Java/A as described in Section 2.4. Both augment Java with the concepts of component and connector and allow hierarchical component composition. I use ArchJava since that the tool support for Java/A is not yet complete. So far, a Java/A compiler should transform Java/A components into pure Java code which can be compiled to byte code using the Java compiler. However, this compiler is not yet publicly available.

[15]http://archjava.org

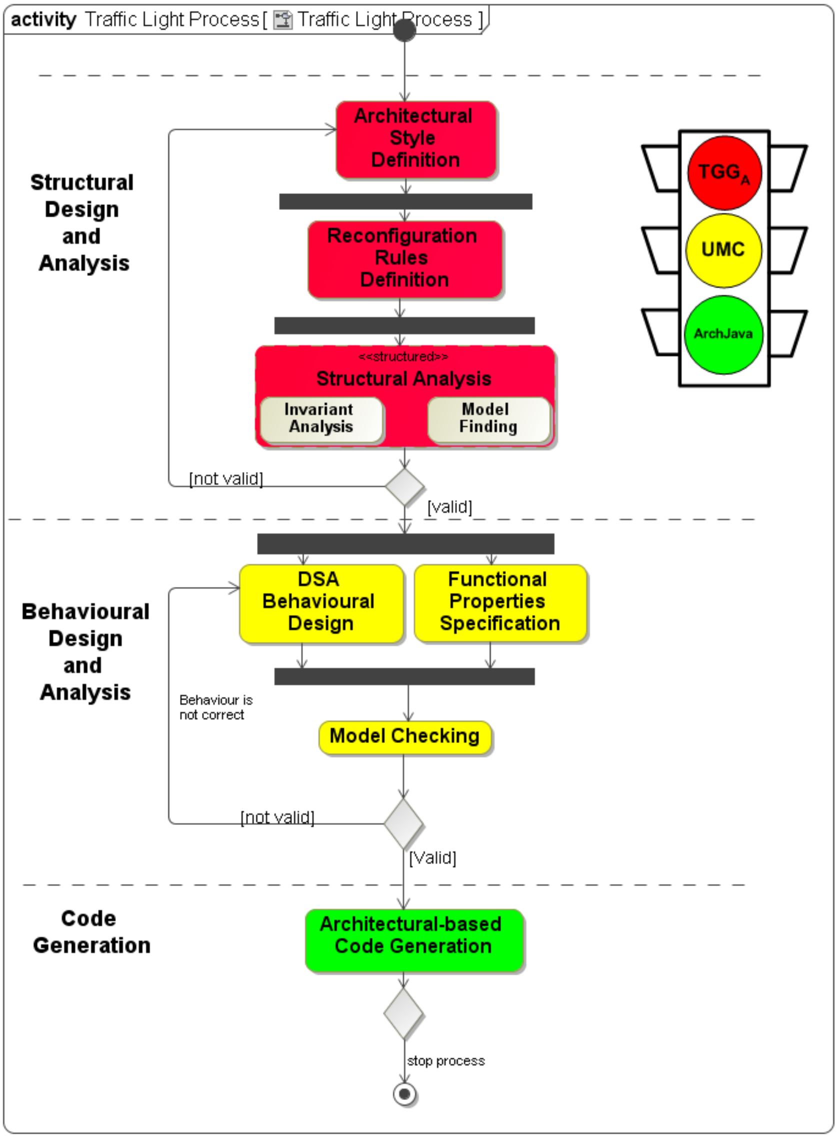

activity Traffic Light Process [Traffic Light Process]
Structural Design and Analysis
Architectural Style Definition
Reconfiguration Rules Definition
<<structured>>
Structural Analysis
Invariant Analysis
Model Finding
[not valid]
[valid]
TGG_A
UMC
ArchJava
Behavioural Design and Analysis
DSA Behavioural Design
Functional Properties Specification
Behaviour is not correct
Model Checking
[not valid]
[Valid]
Code Generation
Architectural-based Code Generation
stop process

Figure 5: The *Traffic Light* Process.

Chapter 3

Formal Development of DSAs

3.1 Running Example: Road Assistance Scenario

We use as running example a simple bike scenario (see (BLMT07)), an ecological variant of the automotive case study of the Sensoria Project (EU).

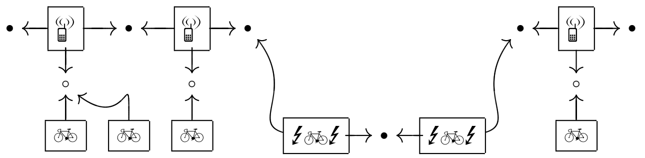


Figure 6: The road assistance scenario.

A road assistance service platform is supported by a wireless network of ad hoc stations that are situated along a road. Bikes equipped with electronic devices can access the service as they move along the road, e.g. to request a taxi in case of breakdowns. The graph in Figure 6 depicts a simple configuration of such a system. Each bike (🚲) is connected to the service access point (○) of a station (📱) which is possibly shared with other bikes. A station and its accessing bikes form a *cell*. Stations, in addition to the service access point, use two other communication points that we call chaining point (•). Such points are used to link cells in larger cell-chains. Bikes can move away from the range of the station of their current cell and enter the range of another cell. A handover protocol supports the migration of bikes to adjacent cells as in standard cellular networks. Stations can shut down, in which case their *orphan* bikes call for a repairing reconfiguration. We shall consider two shutting down situations: one in which the adjacent stations are able to bypass the connection and adopt all orphan bikes and another in which the bypassing is not possible and orphan bikes switch from their normal mode of operation to a cell mode (↯🚲↯), in which they become standalone stations.

3.2 DSA Structural Design and Analysis

The approach described in this section follows what discussed in (BBGM08; BG08) and it is based on modelling of dynamic software architectures using typed graph grammars (`TGG`). Before to introduce in detail our idea we introduce some definitions that we will use in the following. These definitions come from Graph Grammars theory (CMR$^+$97; BCM05; Roz97).

3.2.1 Typed Graph Grammars

This section introduces some basics of the algebraic approaches to graph rewriting considered in the paper. We concentrate on *typed hypergraphs rewriting systems* and in the *single-pushout* (SPO) approach (EHK$^+$97).
Typed rewriting is a variant of the classical approach where rewriting takes place on so-called typed graphs, i.e., graphs labelled over a structure which is itself a graph (i.e., type graph). We present a set of definition that we will use in our formalization.

Definition 1 (Hypergraph) *A* (hyper)graph *is a triple* $H = (N_H, E_H, \phi_H)$, *where* N_H *is the set of nodes,* E_H *is the set of (hyper)edges, and* $\phi_H : E_H \to N_H^+$ *describes the connections of the graph, where* N_H^+ *stands for the set of non-empty* *strings of elements of N_H. We call $|\phi_H(e)|$ the* rank *of e, with $|\phi_H(e)| > 0$ for any $e \in E_H$.*

The connection function ϕ_H associates each hyperedge e to the ordered, non empty sequence of nodes n is attached to.

Definition 2 (Graphs Morphism) *Let G and H be two graphs. A pair of functions $< f_N, f_E >$ where $f_N : N_G \rightarrow N_H$ and $f_E : E_G \rightarrow E_H$ is a graph morphism from G to H if f_N and f_E preserve the tentacle functions, i.e. $f_N^* \circ t_G = t_H \circ f_E$*

Definition 3 (Typed Hypergraph) *Let T be a graph. A typed graph G over T is a graph $|G|$, together with a graph morphism $\tau : |G| \rightarrow T$. A morphism between T-typed graphs $f : G_1 \rightarrow G_2$ is a graph morphism $f : |G_1| \rightarrow |G_2|$ consistent with the typing, i.e. such that $\tau_{G_1} = \tau_{G_2} \circ f$.*

The graph transformations can be done in several way (Roz97), of which the most important are the single-pushout (EHK+97) and double-pushout (CMR+97) approaches. The basic idea of graph rewriting is to consider a set of *graph rewriting rules* of of form $p : L \rightarrow R$, where L is the left-hand and R is the right-hand side of the rule, as schematic descriptions of a possibly infinite set of *direct derivations*. $G \rightarrow^p H$ denotes the direct derivation, where the *match* $m : L \rightarrow G$ fixes an occurrence of L in a graph G. Application of rule p yields a *derived graph H* from G by replacing the occurrence of L in G by R. Each graph rewrite rule defines a partial relation between the elements on its left- and right-hand sides, determining which elements are preserved, deleted, or created by an application of a rule. In this work, taken a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation approach (EHK+97).

For each node or edge x in L there exists a corresponding node or edge in G, namely $m(x)$. We have another morphism named r that maps all items from L to R, which are to remain in G during the rewriting application. Elements that are considered in the match m and that have no image under r are to be deleted. The other are preserved. Elements in R which have no pre-image under r are added to G'. r' is a partial morphism, since that elements from G may be deleted and introduced to get H. New nodes are not in the image of r' but in the image of m'.

Definition 4 (SPO direct derivation) *Given a typed graph* G*, a production* p*, and a match (i.e., a total graph morphism) $g : L \rightarrow G$, we say that there is a direct derivation r' from G to H using p, written r': $G \rightarrow_p H$, if, for suitable morphisms r' and m', the following is a pushout square.*

$$\begin{tikzcd}
\boxed{\mathsf{L}} \arrow[r, "r"] \arrow[d, "m"] & \boxed{\mathsf{R}} \arrow[d, "m'"] \\
\boxed{\mathsf{G}} \arrow[r, "r'"] & \boxed{\mathsf{G'}}
\end{tikzcd}$$

Figure 7: SPO-based graph rewriting.

Finally, a T-typed graph grammar.

Definition 5 ((T-typed) graph grammar) *A* (T-typed) graph grammar $\mathcal{G}$ *is a tuple $\langle T, G_{in}, P\rangle$, where G_{in} is the* initial (T-typed) graph *and P is a set of* productions.

3.2.2 Formalization of DSA

In this sections we describe our idea representing each aspect in two different ways: *Informal* and *Formal*. We start with an informal definition of each architectural aspect and proceed with the Graph-based formalization.

Software Architecture Configurations

Informal Definition Software architectures (SAs) (PW92; SG96) basically consist of the structure of components and the way they are interconnected. Components are high-level computational and data entities that can range from a distributed application to a single thread, from databases to a simple data container. Basic SA elements are (GS06):

- **Components**:*"is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can*

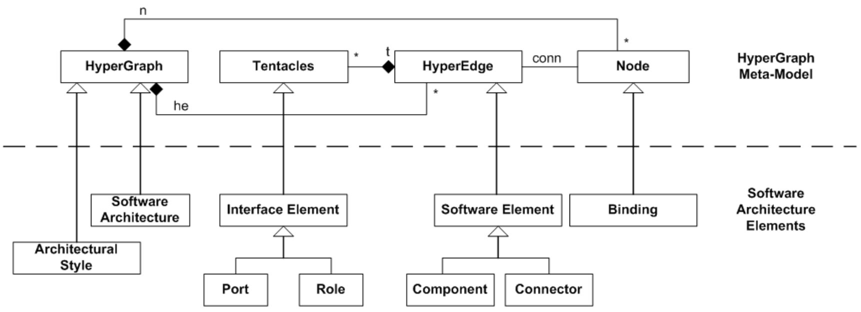


Figure 8: HyperGraph MetaModel and SA elements.

be deployed independently and is subject to composition by third parties"; Each component has a set of **ports**, which model the run-time interfaces of that component, through which it interacts with other components.

- **Connectors**: model the communication between components. We can have define different ways to do it, for example like client-server and pipes communication links. Each connector has a set of **roles**, which model the behavior required of the components that use a given connector.

- **SA Configuration**: it is a graph of components and connectors that are interconnected using ports and rules.

Formal Definition Each software architecture is represented by a hypergraph where components (resp. connectors) are modelled using hyperedges and their ports (resp. roles) by the outgoing tentacles. Components and connectors are attached together connecting their respective tentacles to the same node. In Figure 8 we present the HyperGraph Meta-Model and the respective Software Architecture elements, while in Table 8 graphical symbols that we use to represent them within a graph are depicted.

Through the thesis, we shall omit the prefix 'hyper' for simplicity. Ordinary directed graphs are a particular instance of hypergraph where

$HyperGraphElements$	$ArchitecturalElements$	$GraphicalSymbol$
HyperEdge	Component, Connector	□, ■
Tentacles	Port, Role	→
Node	Binding	○, ●, etc..

Table 8: Graphical Symbols

each edge has two tentacles.

Architectural Style Definition

Informal Definition When designing an architecture, it is desirable to consider the concept of an *architectural style* (SG96), i.e. some set of rules or patterns indicating which components and connectors can be part of the architecture an how they can be legally interconnected. An architectural style can also be seen as a (possible infinite) set of *valid* architectures. Typical architectural styles include client-server, pipelines, layered, multitier, peer-to-peer, etc.

Formal Definition An Architectural style is just a type graph `T` that describes only types of ports, components, connectors plus a set of invariant constraints indicating how these elements can be legally connected. A configuration compliant to such style is then described by the notion of a `T-typed graph`. Typed Graphs are defined as graphs equipped with a typing morphism. Figure 9 depicts the type graph `T` of our running example. It describes the types of components, ports and their allowed connections. The typing morphism is defined using τ_G that maps each element of the configuration in only one element of the type graph T.

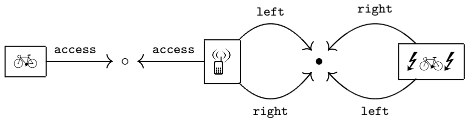


Figure 9: Type Graph T of the running example

Software Architecture Reconfigurations

Informal Definition A Software Architecture is *dynamic* if it change during run-time. Typical changes, which are called *reconfigurations*, include component joining and leaving the system or changing their connections and are usually required for load balancing, fault-recovery, and redimensioning software systems. For instance, in our running example, the system must deal with bikes and stations leaving and joining the system or with bikes that migrate from one station to another. We also need reconfigurations to deal with a station shutting down, by migrating the collection of bikes in adiacent stations (left- or right-side). An additional issue that one would like to have in a reconfigurations mechanism is the capacity to give guarantees about the architectural style. We would like to preserve it after each reconfiguration.

Formal Definition Since we represent architectures by graphs, its reconfigurations are described by a set of rewriting productions that state the possible ways in which a SA configuration may change. The graph transformations can be done in several way (Roz97), of which the most important are the single-pushout (EHK^{+}97) and double-pushout (CMR^{+}97) approaches. In this work we use the former and we leave the use of the latter as our future work.

Dynamic Software Architetures: DSAs

Informal Definition Dynamic Software Architectures are SA which may evolve during system execution. The evolution is generally expressed in terms of reconfiguration operations which correspond to the addition/removal of a component or a connection between components. In addition, a reconfiguration can be induced by an internal (component) event or by an external (user) event. Accordingly, many reconfigurations can not be known in advance and many others may depend on the behaviour of some components.

Formal Definition A DSA is described by a T-Typed graph grammar and in this section we characterize different forms of dynamisms that we can have (And00; End94; GMK02; Ore96; OGT^{+}99; SG02a). It is done using typed graph grammars. Additionally, we show that for verification aspects it make sense to focus only on two forms of dynamicity: Programmed and Repairing.

Given a grammar $\mathcal{G} = \langle T, G_{in}, P \rangle$, we will use the following notions:

- The set $\mathcal{R}(\mathcal{G})$ of *reachable configurations*, i.e., all configurations to which the initial configuration G_{in} can evolve. Formally, $\mathcal{R}(\mathcal{G}) = \{G | G_{in} \Rightarrow^* G\}$.
- The set $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$ of *acceptable configurations* of an architecture are defined as the graphs that have type T and satisfies a auitable property P. Formally, $\mathcal{D}_{\mathsf{P}}(\mathcal{G}) = \{G \mid G\ is\ a\ T{-}typed\ graph \wedge \mathsf{P}\ holds\ in\ G\}$.

Programmed dynamism

Programmed dynamism assumes that all architectural changes are identified at design time and triggered by the program itself (End94). Many proposals in the literature (L. 04; HIM00; Le 98) that use graph grammars for specifying DSA present this kind of dynamism. A programmed DSA $\mathcal{A}$ is associated with a grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P \rangle$, where T stands for the style of the architecture, G_{in} is the initial configuration, and the set of productions P gives the evolution of the architecture. The grammar fixes the types of all elements in the architecture, and their possible connections, where the productions state the possible ways in which a configuration may change.

Programmed dynamism enables for the formulation of several verification questions. Consider the set of desirable configurations $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$, then it should be possible (at least) to know whether:

- the specification is correct, in the sense that any reachable configuration is desirable. This reduces to prove that $\mathcal{R}(\mathcal{G}) \subseteq \mathcal{D}_{\mathsf{P}}(\mathcal{G})$, or equivalently that $\forall G \in \mathcal{R}(\mathcal{G}) : \mathsf{P}\ holds\ in\ G$.

- the specification is complete, in the sense that any desirable configuration can be reached. This corresponds to prove $\mathcal{D}_{\mathsf{P}}(\mathcal{G}) \subseteq \mathcal{R}(\mathcal{G})$, or equivalently that *if* P *holds in* G *then* $G \in \mathcal{R}(\mathcal{G})$.

Hence, programmed dynamism provides an implicit definition of desirable configurations. That is, the sets of desirable and reachable configurations should coincide, i.e., $\mathcal{D}_p(\mathcal{G}) = \mathcal{R}(\mathcal{G})$.

Repairing (or healing) dynamism

Self repairing systems are equipped with a mechanism that monitors the system behaviour to determine whether it behaves within prefixed parameters. If a deviation exists, then the system itself is in charge of adapting the configuration (GS02).

We can think about a repairing architecture as an ordinary graph grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P\rangle$ in which the set of productions is partitioned into three different sets, i.e., $P = P_{pgm} \cup P_{env} \cup P_{rpr}$. Rules in P_{pgm} describe the normal, ideal behaviour of the architecture, i.e., $\mathcal{G}'_{\mathcal{A}} = \langle T, G_{in}, P_{pgm}\rangle$ is a programmed DSA. Rules in P_{env} model the *environment* or, in other words, the ways in which the behaviour of the architecture may deviate from the expected one. Rules in P_{env} may state that the communication among components may be lost or that a non authorised connector become attached to a particular component. Rules P_{rpr} indicate the way in which an undesirable configuration can be repaired in order to become a valid one. That is, the left-hand side of any rule in P_{rpr} identifies a composition pattern in the system that is undesirable. In this way a repairing architecture implicitly defines the desirable configurations of the system as those reachable configurations G that do not exhibit an undesirable composition pattern (i.e., a left-hand-side match for a repairing rule). Formally, the designer would expect that

$$\begin{aligned} G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}}) \quad \textit{iff} \quad & G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \wedge \\ & \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G') \end{aligned}$$

As for the case of programmable dynamism, repairing dynamism allows for the formulation of the following two questions:

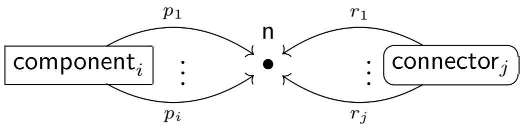


Figure 10: A general graph of types of a software architecture

- the specification is complete. This reduces to prove that $G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$ implies $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \ \wedge \ \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$.
- the specification is correct. This corresponds to prove $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \ \wedge \ \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$ implies $G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$.

In addition, this kind of dynamism naturally poses the question of whether reparing rules are adequate, i.e., whether the set of reparing rules assures that for any configuration that is reachable but not desirable there exists a sequence of repairing rules that moves the configuration to a desirable one. Formally,

- If $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \ \wedge \ (\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$ then $G \Rightarrow_{q_0} G_1 \Rightarrow_{q_1} \ldots \Rightarrow_{q_n} G_n$ with $G_n \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$ and $\{q_0, \ldots, q_n\} \in P_{rpr}$.

Ad-hoc dynamism

Roughly ad-hoc dynamism allows the architecture to evolve freely by adding and removing components and connectors without any restriction. The typed grammar corresponding to ad-hoc DSA should therefore exploit a fully general type graph that contains an infinite number of hyperarcs component$_i$ and connector$_j$ (Figure 10), one for every natural $i, j \in \mathbb{N}$. Any hyperarc component$_i$ (connector$_j$) stands for the type of all connectors that expose exactly i ports (respectively, j roles). For simplicity, we define all nodes as having the same type (otherwise the type graph should be extended, by adding an infinite number of nodes, to represent every possible types). Similarly, the set of production is infinite as it must allow for adding/ removing any kind of components and connectors. This leaves little space for verification issues, as the only guarantees given by ad-hoc dynamicity is that reached graphs are software configurations.

Constructible dynamism

Constructible DSAs are similar to ad-hoc DSAs but here rewriting productions are not the free combination of basic primitives: they are full-fledged programs written in some specific language. The main difference w.r.t. ad-hoc DSA is that a constructible dynamic architecture is mostly characterised by the specific programming language allowed for defining the reconfiguration programs that can manage the evolution. Generally speaking, constructible dynamism provides a very weak notion of desirable configurations, and hence verification aspects are almost meaningless when assuming autonomous reconfiguration (likewise ad-hoc dynamism). However, the situation is slightly different when considering reconfigurations controlled externally (see discussion in the next Section).

Unconstrained vs Constrained dynamism

Basically, constrained dynamism refers to the fact that a change may occur only after pre-defined constraints are satisfied. Such constraints may be (i) the configuration topology, e.g., when components are not connected in a specific, or (ii) the state of a component, e.g., when a component enters into the quiescent state. Topological constraints are naturally modelled by both positive and negative application conditions of graph productions. Hence, topological constrained dynamism may be characterised by a graph grammar whose productions have some contexts (either positive or negative). Differently, constraints related to particular states of components have not an immediate counterpart in our proposal (since our framework does not describe component states). Nevertheless, they can be encoded by thinking about different states of components as different types of hyperedges. In this way, the change of a component state s into s' is represented as the rewrite that removes the hyperarc denoting the component in state s and adds a new hyperarc of type s' with attachments analogous to those of the removed arc. In this case, the fact that the grammar describes a dynamism constrained on the state of some components is hidden by the encoding. Another possibility is to use of attributed graph grammars (LKW93) for equipping components with attributes describing their states.

Unconstrained dynamism refers to the fact that transformations can be applied at any moment. The graph grammar counterpart is the fact that productions have no associated constraints or application conditions, being, in some sense, context free, because they either produce or consume arcs but they do not read them.

Self dynamism

Usually, some kind of dynamisms (like programmed and repairing) are also qualified as "self", meaning that the changes are initiated by the system itself and not by an external agent. We map the notion of self and external dynamism to particular features of the rewrite system. As a starting point we discuss some alternative ways for choosing a particular reconfiguration in a DSA, as proposed in (BCDW04).

- *External*: The reconfiguration rule is selected by an external source. This option resembles the external choice of process calculi, in which the branch of computation to be selected is indicated by the context of process. In this sense, we can interpret a reduction of the form $G \Rightarrow_p G'$ as the fact that the environment selects the application of the production p.

- *Autonomous*: The system selects one of all the applicable transformations in a non-deterministic way. This corresponds to the notion of internal choices in process calculi. Accordingly, we may represent such reductions by hiding the actual name of the applied rule. That is, a rewriting step $G \Rightarrow_p G'$ in which p is autonomous can be represented as $G \Rightarrow_\tau G'$, where τ stands for a hidden change.

- *Pre-defined*: Pre-defined selection is a special case of autonomous choice, in which the system selects in a pre-defined way the appropriate transformation to apply from the set of available ones. In this case, the choice is completely deterministic (like a conditional choice if - then - else - of process calculi). This can be mapped into graph grammars as the definition of priorities in the selection of productions to be applied. As shown in (HHT96), application conditions can be used as priorities for restricting the order in which rules are applied.

Let $\mathcal{G} = \langle T, G_{in}, P_{ext} \cup P_{self} \rangle$ be a grammar, where P_{ext} stands for the set of all reconfigurations that are controlled by the environment, while P_{self} contains all the autonomous productions. We say $\mathcal{G}_{\mathcal{A}}$ has (i) self dynamism if $P_{ext} = \emptyset$, (ii) external dynamism if $P_{self} = \emptyset$, or (iii) mixed dynamism otherwise. Assuming that all rewriting steps $G \Rightarrow_p G'$ are written $G \Rightarrow_\tau G'$ when $p \in P_{self}$, we define the following sets associated to the grammar $\mathcal{G} = \langle T, G_{in}, P_{ext} \cup P_{self} \rangle$:

- The set $\mathcal{S}(\mathcal{G})$ of autonomous or self reconfigurations, i.e., the set of all configurations reachable by applying autonomous changes is: $\mathcal{S}(\mathcal{G}) = \{G \mid G_{in} \Rightarrow_{\tau^*} G\}$.
- The set $\mathcal{E}_c(\mathcal{G})$ of reconfigurations associated to an external sequence $c = p_1 \dots p_n$ of commands:

  $$\mathcal{E}_c(\mathcal{G}) = \{G \mid G_{in} \Rightarrow_{c'} G \wedge c' = \tau^*, p_1, \tau^*, \dots, \tau^*, p_n, \tau^*\}.$$

  Note $\mathcal{E}_c(\mathcal{G})$ contains all the configurations reachable from the initial configuration by applying the sequence c of external chosen rules interleaved with the application of zero or more autonomous reconfigurations.

Clearly, $\mathcal{S}(\mathcal{G})$ and $\mathcal{E}_c(\mathcal{G})$ are subsets of $\mathcal{R}(\mathcal{G})$. Hence, we can proceed as in Section 3.2.2, and formulate some verification problems. In particular, we can specialise the problem $\mathcal{R}(\mathcal{G}) \subseteq \mathcal{D}_{\mathsf{P}}(\mathcal{G})$ to either $\mathcal{S}(\mathcal{G}) \subseteq \mathcal{D}_{\mathsf{P}}(\mathcal{G})$ or $\mathcal{E}_c(\mathcal{G}) \subseteq \mathcal{D}_{\mathsf{P}}(\mathcal{G})$. The last relation is particular interesting when considering ad-hoc or constructible dynamism. In this case, it is possible to check whether a particular reconfiguration program may produce acceptable configurations.

Table 9: Classification summary

Dynamicity	**References**	**Correctness**	**Completeness**
Programmed	(L. 04; End94) (HIM00; Le 98) (Wer98)	+	+
Repairing	(ADG98; GS02) (OGT$^+$99; SG02a) (GMK02)	+	+
Ad hoc	(BISZ98; End94) (YM92)	-	-
Constructible	(And00; Ore96)	$\pm$	$\pm$

Final Remarks

In this section we have characterised different aspects of dynamic reconfiguration as particular features of graph rewriting systems. By taking advantage of this framework, we have distilled whether such kinds of dynamisms allow for posing typical questions about the completeness and correctness of the architectural specification. Figure 9 summarises the conclusions for the different types of dynamisms.

As mentioned in Section 3.2.2, given a characterization of all desirable configurations of a programmable architecture, e.g., by defining a property P that should hold in every configuration, then it would be possible to prove whether the architectural specification is correct (by showing that P holds in every reachable configuration) and complete (by proving any configuration satisfying P is reachable). Correctness and completeness properties could also be associated to repairing dynamism. But, differently from programmed dynamism, some reachable configurations of a repairing architecture may be non desirable, and hence, these configurations should be transformed into a desirable one by using repairing rules. The main idea is that undesirable configurations are characterized as those reachable configurations in which some repairing rule is applicable to obtain only desirable configurations. Such questions are meaningless for ad-hoc dynamicity, where every configuration is potentially reachable. Analogously for constructible dynamism, even if some kind of weak analysis could be performed in this case. For instance, to prove that particular configurations are not reachable when the reconfiguration language forbid some kind of programs.

Actually, the above characterization corresponds to the case in which transformations are all autonomous, i.e., when we assume self dynamism. When external dynamism is considered, also correctness and completeness properties over ad hoc and constructible architectures can be formulated. For instance, given a particular (set of) desirable configuration(s) it can be proved whether a particular transformation or configuration program selected by a programmer produces a desirable configuration. Even more interesting is the case in which mixed dynamism is considered. Assume an ad hoc architecture where some productions are considered external and others autonomous or self. In this case, external transformations account for the reconfigurations activated by a user, while autonomous transformations model the actual program that performs the transformation (a kind of scripting). In this case, it would be possible to check whether a particular script produces a correct configuration when it is applied over a specific configuration. In this thesis we only consider programmed dynamism. Other dynamisms will be part of our future work.

3.2.3 TGG_A: An Alloy Implementation of Typed Graph Grammars

The implementation of each concepts introduced in the previous section has been done using Alloy (Jac06; Jac02), a light-weight approach to the modelling and analysis of software models. Since that we use Typed Graph Grammars to represent DSAs, after an overview of basic Alloy concepts we present TGG_A, our implementation of Typed Graph Grammars concepts that will be used to design and verify structural aspects of DSAs.

The main aspects on which we focus are concerned with:

- *Architectural Representation*, i.e. convenient ways to design a DSA, to build it, to browse it;
- *Architectural Styles*, i.e. convenient ways to constrain DSAs under consideration to satisfy certain requirements;
- *Structural Properties*, i.e. convenient logical formalisms to express relevant structural properties;
- *Architectural Analysis*, i.e. efficient techniques and tools for verification.

We show how to tackle these aspects with our approach. The outcome of our experience suggests that TGG_A is well suited for an early phase of the development, where the architectural constraints imposed by the style are defined in an iterative process of refinement of the model and style, assisted by model-finding techniques.

Alloy

Alloy (Jac02; Jac06) provides a description language to represent software models, based on signatures and relations, which is suited for a set-theoretic presentation of graphs. Alloy also provides a logic, based on an extension of first-order logic with relational operators, to represent properties or constraints of models. We have used this logic to implement concepts like architectural style, architectural configurations, architectural reconfigurations and architectural properties. The Alloy Analyzer translates the model and the logical predicates into a Boolean formula, uses efficient SAT solvers to decide satisfiability and provides a counterexample in negative case. We will show how to use these capabilities to ensure style-consistency, perform model-finding and validate architectural properties. Before to introduce these aspects we present some preliminary aspects of the Alloy languages summarized from (Jac06).

Signatures and Fields

A *signature* introduce a set of atoms. The declaration:

```
sig Edge{}
```

introduces a set named `Edge`. A signature is more than just a set, because it can include declarations of relations. A set can be introduced as a subset of another set, thus

```
sig E1 extends Edge {}
```

introduces a set named `E1` that is a subset of `Edge`. The signature `E1` is an extension or subsignature of `Edge`. A signature such as `Edge` that is declared independently of any other is a *top-level* signature. The extensions of a signature are mutually disjoint, as are top-level signatures. So if we declare:

```
sig Edge {}
sig Node {}
sig E1 extends Edge {}
sig E2 extends Edge {}
```

we can say that `Edge` and `Node` are disjoint, and `E1` and `E2` are disjoint (but not that Edge = E1 + E2). Moreover we can define an *abstract signature* that has no elements except those belonging to its extensions. To define an atom (i.e., basic element) of our system we define a signature marking it with the keyword *one*, represents singleton sets - sets that contain a single elements. In an instance, such a set will correspond to a single atom. A signature defines a local namespace for its declarations, so we can use the same field name in different signatures, and each occurrence will refer to a different field. The only restriction is that if two signatures share a field name, they must not overlap.

Relations

Relations are declared as fields of signatures. The signature *Edge*

defines a collection of edges, each of which shows some connections that map each label to nodes. The keyword `lone` in the declaration in-

```
sig Edge
{
  conn: Label->lone Node
}
```

dicates multiplicity, in this case that each label is mapped to at most one node.

Types and Type Checking

Alloy's type system has two functions. First, it allows the analyzer to catch errors before any serious analysis is performed. The essential idea it that an expression is erroneous if it can be shown to be redundant, using types alone. This notion of error, although unconventional, accepts and rejects expressions much as one would except. Second, the type system is used to resolve overloading. When different signatures have fields with the same name, the type of an expression is used to determine which field of a given name is meant. Types are associated implicitly with signatures. A *basic type* is introduced for each-top level signature and for each extension signature. When signature `E1` extends signature `Edge`, the type associated with `E1` is a *subtype* of the type associated with `Edge`. There are two kinds of type error. First, since the Alloy's logic assumes that all relations have a fixed arity, it is illegal to form expressions that would give relations of mixed arity. Second, an expression is illegal if it can be shown, from the declarations alone, to be redundant, or to contain a redundant subexpression. A common and simple case is when an expression is redundant because it is equal to the empty relation.

Facts, Predicates, Functions, and Assertions

The constraints of a model are organized into *paragraphs*. Assumptions are placed in `fact` paragraphs; implications to checked are placed in `assertions`; constraints to be used in different contexts are packaged as `predicates`; and reusable expressions are packaged as `functions`. Constraints that are assumed always to hold are recored as `facts`. A model can have any number of facts, each a paragraph of its own, labeled by the keyword fact, and consisting of a collection of constraints. The order in which facts appear, and the order of constraints within a fact, is not important. Many facts are constraints that apply to each element of a signature's set. These can be recorded more succinctly as *signatures facts*. A constraint immediately following a signature is implicitly quantified over its elements, and each field reference is implicitly deferences, just like fields mentioned in field declarations. The following code describes a fact in which two graphs must have different nodes.

```
fact GraphElements_Constraints
{
  all g1,g2: Graph |g1!=g2 => #(g1.n & g2.n)=0
}
```

A function is a named expression, with zero or more declarations for arguments, and a declaration expression for the result. When the function is used, an expression must be provided for each argument; its meaning is just the function's expression, with each argument replaced by its instantiating expression. The following code describes a function defining the arity of an edge (i.e., number of tentacles).

```
fun arity[e: Edge]: Int
{
      #(e.conn)
}
```

A predicate is a named constraint, with zero or more declarations for arguments. When the predicate is used, an expression must be provided for each argument; its meaning is just the predicate's constraint with each argument replaced by its instantiating expression. A predicate can be used to represent an operation, which describes a set of state transitions, by constraining the relationship between pre- and post-states. The following code presents the "rewriting step" predicate in which g and g' denote the before and after graph respectively.

```
pred rwStep[g,g': Graph, p: Production] {...}
```

An `assertion` is a constraint that is intended to follow from the facts of the model. The Alloy Analyzer checks assertions. If an assertion does not follow from the facts, then either a design flaw has been exposed, or a misformulation. Even assertions that do follow are useful to record, both because they express properties in a different way, and because they act like regression tests, so that if an error is introduced later, it may be detected by checking assertions.

Command and Scope

To analyze a model in Alloy, we must write a `command` and instruct the tool to execute it. A `run` command tells the tool to search for an instance of a predicate. A `check` command tells it to search for a counterexample of an assertion. In addition to naming the predicate or assertion, we may also give a `scope` that bounds the size of the instances or counterexamples that will be considered. If we omit the scope, the tool will use the default scope in which each top-level signature is limited to three elements. For example the command `check Edge for 5` places a bound of 5 on all top-level objects.

Modules and Polymorphism

Alloy has a single module system that allows you to split a model among several modules, and make use of predefined libraries. Modules correspond one-to-one with files. Every analysis is applied to a single module; any other modules containing relevant model fragments must be explicitly imported. Each module has a path name that must be match the path of its corresponding file in the file system. Paths are interpreted with respect to a collection of root directories, given as preferences in the tool. The first line of every module is a `module header` of the form:

```
module modulePathName
```

Every module that is used must have an explicit *import* immediately following the header as in the following code:

open `modulePathName`

Analysis

Checking an assertion and running a predicate reduce to the same analysis problem: finding some assignment of relations to variables that makes a constraint true. Alloy's relational logic is undecidable. This means that it is impossible to build an automatic that can tell whether an assertion is valid - that is, holds for every possible assignment. The analysis underlying Alloy, i.e. *model finding*, makes a different compromise. Rather than attempting to construct a proof that an assertion holds, it looks for a *refutation*, by checking the assertion against a huge set of test cases, each being a possible assignment of relations to variables. If the assertion is found not to hold for a particular case, that case is reported as a *counterexample*. If no counterexample is found, it's still possible that the assertion does not hold, and has a counterexample that is larger that any of the test cases considered. Instance finding is well suited to analyzing invalid assertion because it generates counterexamples, which can usually be easily traced back to the problem in the description. To make instance finding feasible, a `scope` is defined that limits the size of instances considered. The analysis effectively examines every instance within the scope, and an invalid assertion will only slip through unrefuted if its smallest counterexample is outside the scope. The scope thus defines a multidimensional space of test cases, each dimension corresponding to the bound on a particular signature.

The Alloy Analyzer

Every analysis involves solving a constraint: either finding an instance (for a `run` command) or finding a counterexample (for a *check*). The Alloy Analyzer is therefore a constraint solver for the Alloy logic. In its implementation, however, it is more if a compiler, because, rather than solving the constraint directly, it translates the constraint into a boolean formula and solves it using an off-the-shelf *SAT (Satisfiability) Solver*. The

Alloy Analyzer is bundled with several SAT solvers, the fastest of which are Chaff (MMZ+01) and (GN02), and a preference setting lets we choose which is used.

TGG_A in detail

We use Alloy to implement graphs, graphs morphisms, graphs matchings, and graph transformations. Starting from this core implementation we have extended it to represent concepts like architectural styles, architecture configuration and reconfigurations, etc. The relation between graph and architectural concepts have been already depicted in Figure 8. In this section, for each aspect introduced in Section 3.2.2 we show the respective Alloy implementation. At the end we show in which way, using TGG_A we can verify structural properties of a DSA.

Graphs The three basic concepts in the model of each graph are *nodes*, *tentacles* and *edges* that are represented as three Alloy signatures as follows:

```
1  sig Node{}
2  sig Tentacles{}
3  sig Edge
4  {
5    tentacles: set Tentacles,
6    conn: tentacles->lone Node
7  }
8  sig Graph
9  {
10   he: set Edge,
11   n: set Node
12 }
```

According to the above definition, nodes and tentacles are atomic concept, while edges has a field `tentacles` that describes the set of tentacles and `conn` that maps each tentacle to nodes. The keyword `lone` in the declaration indicates *multiplicity*, in this case that each label is mapped to at most one node. The signature `Graph` (lines 8-12) is used to define as a graph as structure composed of nodes and edges. In order to construct correct graphs we have defined some constraints. In Alloy the constraints of a model are organized into *paragraphs*. Assumptions are placed in *fact* paragraphs. Constraints that are assumed always to hold are recorded as *facts*. A model can have any number of facts, labeled by the keyword *fact*, and consisting of a collection of constraints. In the following code we present some facts that we have defined to ensure that our graphs satisfy some properties. Characters between // and the end of the line are comments that describe each constraint.

```
// facts on Graphs and Graph elements (Nodes , Edges and Tentacles)
fact GraphElements_Constraints
{
  // each element (Nodes, Edges and Tentacles) must be element
  // of a single Graph
  all edge: Edge | some g: Graph | edge in g.he
  all node: Node | some g: Graph | node in g.n
  all t1: Tentacles | some g: Graph | some e1:g.he | t1 in e1.tentacles

  // nodes at which each Edge is connected must be nodes of the same Graph
  all g:Graph| all e: g.he | univ.(e.conn) in g.n
}
```

In order to see an example of a graph, generated from the previous code, we have defined a predicate (i.e., `show`) and a *command* to find an instance of the predicate. Moreover a *scope* is defined that limits the size of instances considered. In the following code we want to generate a graph with two edges, one node and two tentacles.

```
one sig g1 extends Graph {}
one sig e1,e2 extends Edge {}
one sig n1 extends Node {}
one sig t1,t2 extends Tentacles {}
pred show[]
{
  g1.he= e1 + e2
  g1.n = n1
  //t1 is a tentacle of e1
  t1 in e1.tentacles
  //t2 is a tentacle of e2
  t2 in e2.tentacles
}
run show for 1 Graph, 2 Edge, 1 Node, 2 Tentacles
```

When we run the code above the Alloy Analyzer generates a unique instance of a graph that is depicted in Figure 11.

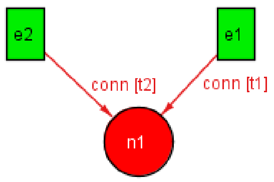


Figure 11: Graph Instance

Typed Graphs, Type Graph, Morphisms To generate Architectural configurations that are conform to a style, we have defined in Alloy the signature `TypedGraph` that is a graph with a typing morphism. Additionally we have defined a set of constraints that must be valid for each typed graph. Listing 3.1 presents the implementation of the total morphism among two graphs (lines 1-26) and the definition of Typed Graphs with relative constraints (lines 28-48). It is important to note that the target graph of the morphism in each Typed Graph is the Type Graph `T`.

An *architectural style* consists of a set of basic elements (components, connectors, ports and roles) that can constitute an architectural configuration plus a set of constraints indicating how these elements can be legally connected. We define in Alloy a module called `STYLE` that contains all these elements. It is subdivided in two parts, the first to define basic elements and the second to define constraints on them. Each basic element is defined using a singleton extension of node, tentacle or edge signatures. Each instance of the `Bike` component type must have only one connection to the `AccessPoint` interface by an `Access` port (lines 18-22) . Each instance of the `BikeStation` component type must have two connections, both to the `ChainPoint` interface but using two different ports, one of `Left` type and another of `Right` type (lines 23-27) . Finally each instance of the `Station` component type must have three connections type `Left`, `Right` and `Access`. The first two are connected to `ChainPoint` interfaces while the third to the `AccessPoint` interface (lines 13-17). In Listing 3.2 we can see the Alloy code that implements these aspects.

```
sig TotalMorphism
{
 source: Graph,
 target: Graph,
 fE: Edge->Edge,
 fN: Node->Node
}
{

 // Domain of the relation f
 fE.univ = source.he
 fN.univ = source.n

 // Range of the relation f
 univ.fE in target.he
 univ.fN in target.n

 //uniqueness
 all e1: source.he | one e2: target.he | fE[e1]=e2
 all n1: source.n | one n2: target.n | fN[n1] = n2

 // well-formedness
 all e1: source.he | e1.tentacles = fE[e1].tentacles
 all e1: source.he | all t1:e1.tentacles
                | (fE[e1]).conn[t1] = fN[e1.conn[t1]]
}

sig TypedGraph
{
 typingmorphism : TotalMorphism
}
{
 typingmorphism.target=TypeGraph
}

fact onTypedGraph
{
 all tg: TypedGraph |
    one morph: tg.typingmorphism|
    one g1: morph.source |
    all e1,e2: g1.he |e1!=e2 and morph.fE[e1] =Station and
           morph.fE[e2]= Station => #(e1.conn&e2.conn)=0
 all tg: TypedGraph |
    one morph: tg.typingmorphism|
    one g1: morph.source |
    all e1,e2: g1.he |e1!=e2 and morph.fE[e1] =BikeStation and
          morph.fE[e2]= Station => #(e1.conn&e2.conn)=0
}
```

Listing 3.1: TypedGraph and Total Morphism

```
module STYLE
open TGG
//-------------TYPEGRAPH DEFINITION--------------------
/* ---------------NOTATION---------------------------------
      AP = Access_Point
      CP = Chain_Point
-----------------------------------------------------------*/
//Architectural Bindings
one sig CP, AP extends Node {}
//Ports of Components
one sig Left, Right, Access extends Tentacles {}
//Architectural Components
one sig Station extends Edge{}
{
 tentacles = Left+Right+Access
 conn = Left->CP + Right->CP + Access->AP
}
one sig Bike extends Edge{}
{
 tentacles = Access
 conn = Access->AP
}
one sig BikeStation extends Edge{}
{
 tentacles = Left+Right
 conn = Left->CP + Right->CP
}
fact onTypeGraph
{
 TypeGraph.n = CP+AP
 TypeGraph.he = Station + Bike+BikeStation
}
```

Listing 3.2: Type Graph of the Running Example

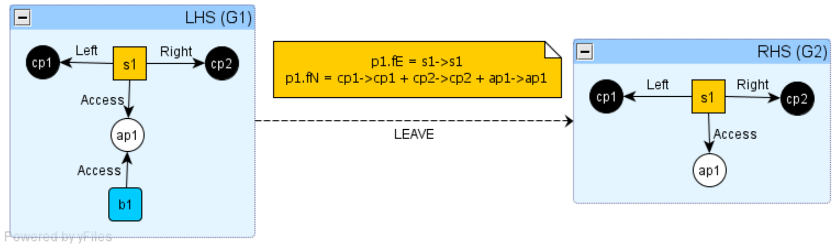


Figure 12: LEAVE1 Production

SPO Graph Rewriting We ha defined a set of productions (i.e., reconfiguration rules) that state the possible ways in which a SA configuration may change. Each rule is defined as partial morphisms $p : L \rightarrow R$ (shown in Listing 3.3), where L and R are typed graphs. Given an initial graph G and a production p, a rewriting of G using p is implemented in Alloy using the single-pushout graph transformation approach (EHK$^+$97). To implement it we have defined the predicate `rwStep` that executes one single rewriting step and produces the target graph G' (shown in Listing 3.5). Another important aspect that the SPO rewriting approach uses is the exist of a match among two Typed Graphs. It is the precondition of the rewriting step application. For this reason we have implemented the `Match` and `TypedMatch` signature as presented in the Listing 3.4.

For our running example we have defined six rewriting rules. Here we show only, `JOIN1`, `LEAVE1` and `MIGRATION`, the other are presented in the Appendix A. The rule `JOIN1` is used for a bike to join a station. Dually, the rule `LEAVE1` is used for a bike to leave the station. Additionally we consider the migration of bikes caused by their mobility. Clearly, the problem can be tackled by a sequence of leave and join reconfigurations. However, it would be better to perform it in a single step defining the rule `MIGRATION`. Listing 3.6 shows the `LEAVE1` production while Figures 12, 13 and 14 depict all of them. The complete code is presented in the Appendix A at the end of the thesis.

```
sig PartialMorphism
{
 source: Graph,
 target: Graph,
 fE: Edge -> lone Edge,
 fN : Node -> lone Node
}
{
 // mapping functions description
 fE.univ in source.he
 fN.univ in source.n
 univ.fE in target.he
 univ.fN in target.n

 // f maps a subgrapgh of the source graph
 all e1: fE.univ |univ.(e1.conn) in fN.univ

 // injectivity
 all n1,n2: source.n | fN[n1] = fN[n2] => n1=n2
 all e1,e2: source.he |fE[e1] = fE[e2] => e1=e2

 // well-formedness of the partialmorphism
 all e1: fE.univ | e1.tentacles = fE[e1].tentacles
 all e1: fE.univ | all t1: e1.tentacles | fE[e1].conn[t1] = fN[e1.conn[t1]]
}

sig TypedPartialMorphism
{
 // source and target TypedGraphs
 s: TypedGraph,
 t: TypedGraph,
 PMorphism: PartialMorphism
}
{
 PMorphism.source = s.typingmorphism.source
 PMorphism.target = t.typingmorphism.source
 all e1: PMorphism.fE.univ |
   s.typingmorphism.fE[e1] = t.typingmorphism.fE[PMorphism.fE[e1]]
 all n1: PMorphism.fN.univ |
   s.typingmorphism.fN[n1] = t.typingmorphism.fN[PMorphism.fN[n1]]
}
```

Listing 3.3: PartialMorphism and Typed Partial Morphism

```
sig Matching
{
  source: Graph,
  target: Graph ,
  fE: Edge -> lone Edge,
  fN : Node -> lone Node
}
{

  // mapping functions description
  fE.univ = source.he
  fN.univ = source.n
  univ.fE in target.he
  univ.fN in target.n

  // f maps a subgrapgh of the source graph
  all e1: fE.univ |univ.(e1.conn) in fN.univ

   // injectivity
  all n1,n2: source.n | fN[n1] = fN[n2] => n1=n2
  all e1,e2: source.he |fE[e1] = fE[e2] => e1=e2

  // well-formedness of the matching
  all e1: fE.univ | e1.tentacles = fE[e1].tentacles
  all e1: fE.univ | all t1: e1.tentacles |
                fE[e1].conn[t1] = fN[e1.conn[t1]]
}
// Matching among two TypedGraphs
sig TypedMatching
{
  s: TypedGraph,
  t: TypedGraph,
  match: Matching
}
{
  match.source = s.typingmorphism.source
  match.target = t.typingmorphism.source
}
```

Listing 3.4: Matching

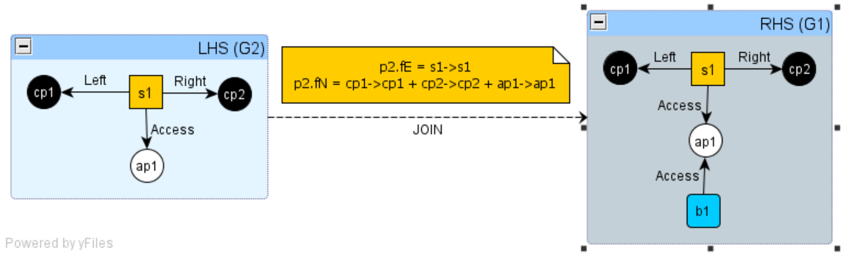


Figure 13: JOIN1 Production

```
pred rwStep [G,G':TypedGraph, P: TypedPartialMorphism,
             trace: TypedPartialMorphism]
{
  one g: TypedMatching | {
  g.s=P.s and g.t = G and

  G'.typingmorphism.source.he = (G.typingmorphism.source.he)-
                              g.match.fE[(P.s.typingmorphism.source.he)] +
                              (P.t.typingmorphism.source.he)

and

G'.typingmorphism.source.n = (G.typingmorphism.source.n)-
                             g.match.fN[(P.s.typingmorphism.source.n)] +
                             (P.t.typingmorphism.source.n)
}

and
trace.s = G and trace.t=G'
and
all n1: G'.typingmorphism.source.n |
        n1 in univ.(g.match.fN) and
        (g.match.fN.n1) in (P.PMorphism.fN).univ
        =>
        n1 in (trace.PMorphism.fN).univ
}
```

Listing 3.5: SPO Rewriting Step Predicate

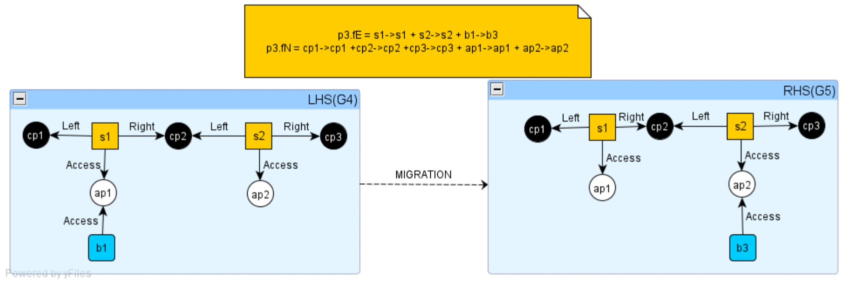

p3.fE = s1->s1 + s2->s2 + b1->b3
p3.fN = cp1->cp1 +cp2->cp2 +cp3->cp3 + ap1->ap1 + ap2->ap2
LHS(G4)
RHS(G5)
cp1
Left
s1
Right
cp2
Left
s2
Right
cp3
Access
ap1
Access
b1
ap2
MIGRATION
b3

Figure 14: MIGRATION Production

```
module PRODUCTIONS

open TGG
open STYLE
//-------------BASIC ELEMENTS-------------------
one sig s1,s2,b1,b2,b3 extends Edge{}
one sig cp1,cp2,cp3,ap1,ap2 extends Node{}
//------------GRAPHS----------------------------
one sig g1 extends Graph {}
{
 he =s1+b1
 n= cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 b1.conn = Access->ap1
}
one sig g2 extends Graph{}
{
 he = s1
 n = cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
}
//------TYPED GRAPHS------------------------------
one sig G1 extends TypedGraph{}
{
 typingmorphism.source = g1
 typingmorphism.fE = s1->Station + b1->Bike
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
one sig G2 extends TypedGraph{}
{
 typingmorphism.source = g2
 typingmorphism.fE = s1->Station
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
//-------------LEAVE1 PRODUCTION--------------------------
one sig p1 extends PartialMorphism{}
one sig LEAVE1 extends TypedPartialMorphism{}
{
 s = G1
 t= G2
 PMorphism = p1
 PMorphism.source = g1
 PMorphism.target = g2
 PMorphism.fE = s1->s1
 PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1
}
```

Listing 3.6: Running Example Productions

ModelFinding

Model finding is the main analysis capability offered by Alloy. The Alloy Analyzer basically explores (a bounded fragment) of the state space of all possible models. For instance, we can easily use the Alloy Analyzer to construct initial configurations: we need to ask for a graph instance satisfying the style facts and having a certain number of bikes, stations and bikestations. In order to test this Alloy potentiality we have created a module called `MODEL-FINDING` in which only defining elements of our initial configuration (i.e., edges, nodes and tentacles) we can generate the set of possible software architectures composed of a precise number of components, ports and attachments. To generate style-conformant SAs configurations, in the Listing 3.7, we open the modules `TGG` and `STYLE` (lines 3-4). When we execute the run command (line 22) the Alloy Analyzer firstly verifies each constraints defined in both modules and after generates all possible configurations with 3 Bikes and 2 Stations. Figure 15 presents only two of them.

```
1   module MODELFINDING
2
3   open TGG
4   open STYLE
5
6   one sig b1,b2,b3,s1,s2 extends Edge{}
7   one sig ap1,ap2, cp1,cp2,cp3 extends Node{}
8   one sig InitialGraph extends Graph{}
9   {
10    he= b1+b2+b3+s1+s2
11    n= ap1+ap2+cp1+cp2+cp3
12  }
13  one sig InitialConfiguration extends TypedGraph{}
14  {
15    typingmorphism.source = InitialGraph
16    typingmorphism.fE = b1->Bike + b2->Bike + b3->Bike +
17                    s1->Station + s2->Station
18    typingmorphism.fN = ap1->AP + ap2->AP + cp1->CP + cp2->CP +
19                    cp3->CP
20  }
21  pred showTypedGraph[]{}
22  run show for 8 but 1 TypedGraph
```

Listing 3.7: Model Finding Module

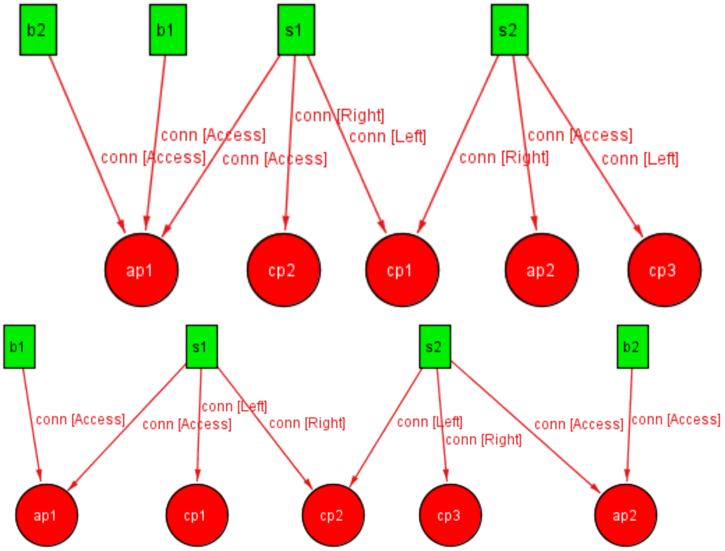


Figure 15: Two possible Style-Conformant SA configuration

Structural Analysis

In order to validate, from the structural point of view, each DSA, in this section we present a set of structural analysis that we can perform using Alloy. Before to introduce them, we show the signatures that we have defined to represent Programmed DSA. In line with the idea already proposed in (BS06), we have specified the `ProgrammedDSA` signature where each configuration is a TypedGraph and each `Transition` is a signature with three relations: `startingState` and `arrivalState` identify the source and target `TypedGraph`, while `trigger` defines the `Production` (i.e., rewriting rule) that triggers each reconfiguration. Each `ProgrammedDSA` provides a `next` relation that assigns exactly one configuration (i.e., TypedGraph) different from the final one. Since that we are specifying DSA that are programmed, the `trigger` field of the `Transition` set, will be one production of the predefined set of productions.

After these signatures, we have defined facts to constrain them in List-

```
sig Transition {
 startingState: TypedGraph,
 arrivalState: TypedGraph,
 trigger : TypedPartialMorphism
}
sig ProgrammedDSA
{
 configurations: set TypedGraph,
 transitions: set Transition,
 first, last: configurations,
 next: (configurations-last) one -> one (configurations-first)
}
{
  // the first TypedGraph can not be an arrivalState
  first not in transitions.arrivalState

  // the last TypedGraph can not be a startingState
  last not in transitions.startingState
  next.univ in transitions.startingState
}
```

ing 3.8. The first two describe the determinism and the correctness of each transition. The last is very important, it is the responsible for the Typed Graph evolution.

Programmed DSA Example In the Listing 3.9 we present an example of Programmed DSA (e.g.,`pdsa1`) where its initial Configuration is `G3` and the set of possible reconfigurations is composed of four productions.

In Figure 16 we can see the Alloy Analyzer output and the graphical representation of the example. The start configuration of the programmedDSA is `G3` while `G7` is the final. The latter is generated after four reconfiguration steps.

Reachability Alloy allow us to check the reachability of given configurations through a finite sequences of rewriting steps. To do this we have defined the predicate `Reachability`. When we check an assertion on `G7` using this predicate, the Alloy Analyzer outptus it "`No counterexample found`", it means that `G7` is a reachable configuration of our running example.

Figure 17 presents the set of *Reachable Configurations* of the Programmed DSA. This fact demonstrates that we can check easily if a par-

```
fact determinism
{
 all pdsa: ProgrammedDSA | no disj t1,t2: pdsa. transitions {
 t1.startingState = t2.startingState
 t1.trigger = t2.trigger}
}
fact CorrectTransition
{
 all pdsa: ProgrammedDSA |
 all t: pdsa.transitions |
  t.startingState in pdsa.configurations and
  t.arrivalState in pdsa.configurations
}
fact GenerationTransition
{
   all pdsa: ProgrammedDSA|
   all  trans: pdsa.transitions |one tr: TypedPartialMorphism |
   rwStep[trans.startingState, trans.arrivalState, trans.trigger, tr] and
   trans.arrivalState = trans.startingState.(univ.next)
}
```

Listing 3.8: Programmed DSA Constraints

```
one sig pdsa1 extends ProgrammedDSA{}
{
 first = G3
 transitions.trigger = LEAVE1 + JOIN1 + STATIONConn + LEAVE2
}

pred show[]{}
run show for 8 but 1 ProgrammedDSA
```

Listing 3.9: Programmed DSA Example

```
pred Reachability [conf: TypedGraph]
{
 all pdsa: ProgrammedDSA |
  conf in pdsa.configurations
}
```

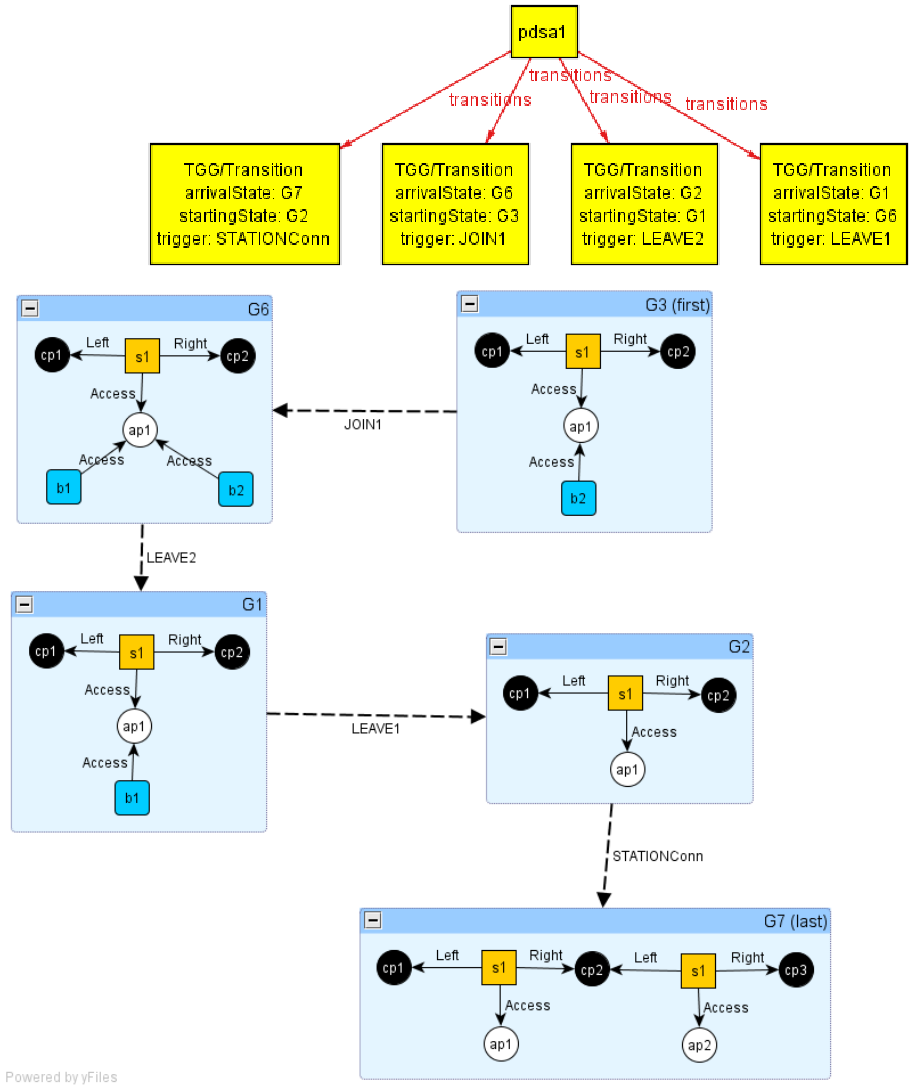

pdsa1
transitions
transitions
transitions
transitions
TGG/Transition arrivalState: G7 startingState: G2 trigger: STATIONConn
TGG/Transition arrivalState: G6 startingState: G3 trigger: JOIN1
TGG/Transition arrivalState: G2 startingState: G1 trigger: LEAVE2
TGG/Transition arrivalState: G1 startingState: G6 trigger: LEAVE1
G6
G3 (first)
JOIN1
LEAVE2
G1
G2
LEAVE1
STATIONConn
G7 (last)
cp1
cp2
cp3
s1
ap1
ap2
b1
b2
Left
Right
Access
Powered by yFiles

Figure 16: Programmed DSA Example

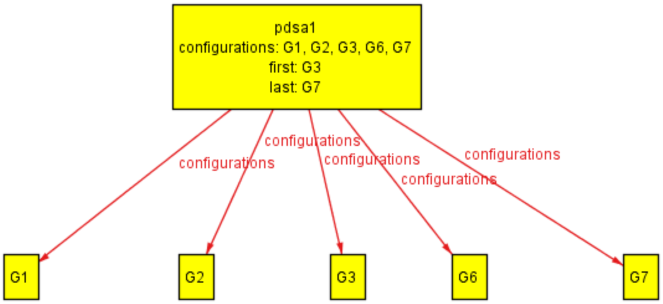


Figure 17: Reachable Configurations

ticular configuration is reachable. In order to have the system correct and complete we must yet verify that each reachable configuration is desirable. We demonstrate this using the *Invariant Analysis* of the running example.

Invariant Analysis The second kind of analysis that we perform is called *Invariant Analysis*; its objective is to check if a property `P` is invariant under sequences of applications of reconfigurations. The objective of this analysis is that: given a property P is invariant under sequences of applications of some operations. In our case this operation is the rewriting step that from an initial configuration G and a Production P generates a new configuration G′. A technique useful for stating the invariance of a property P consists of specifying that P holds in the initial configuration, and that for every non initial configuration and every rewriting operation, the following holds: P(G) and $rwStep(G, G') \rightarrow P(G')$. To do this we have defined the predicate `InvariantAnalysis` that will be used to verify the invariants of a set of properties on our system.

```
pred InvariantAnalysis[]
{
 all pdsa: ProgrammedDSA | all conf: pdsa.configurations | Property1[conf]
}
```

For this objective we have defined a set of properties that each DSA configuration, after a rewriting step must satisfy. In the following we describe each of them with the Alloy code related.

1. **Property 1**: Each bike can be connected to only one access point using one port of type `Access`
2. **Property 2**: The system can not have bikes disconnected and each bike has at most one connection.
3. **Property 3**: If one bike is connected to an access point then must exist a unique station that is connected to the same access point.

```

pred Property1[conf: TypedGraph]
{
  all e: conf.typingmorphism.source.he |
    conf.typingmorphism.fE[e] = Bike
    =>
    conf.typingmorphism.fN[univ.(e.conn)] =AP
    and
    e.tentacles=Access
}
pred Property2[conf: TypedGraph]
{
  all e: conf.typingmorphism.source.he |
    conf.typingmorphism.fE[e] = Bike
    =>
    #(e.conn)=1
}
pred Property3[conf: TypedGraph]
{
  all e1: conf.typingmorphism.source.he |
    conf.typingmorphism.fE[e1] = Bike
    =>
    one n: conf.typingmorphism.source.n |
    one e2: conf.typingmorphism.source.he |
    conf.typingmorphism.fN[n] = AP and
    conf.typingmorphism.fE[e2] = Station and
    n in univ.(e1.conn) and n in univ.(e2.conn)
}
```

Listing 3.10: Invariant Properties

If we check the Invariant Analysis predicate for each property in Listing 3.10 we have that each property is valid for each reachable configuration of our running example. Considering the definition presented in

Section 3.2.2 we can conclude that the specification of the running example is correct and complete.

3.3 DSA Behavioural Design and Analysis

Starting from the SA configurations generated in the previous phase, in this section we describe how we validate the SAs conformance to certain behavioral properties using Model Checking techniques. We associate to each component of each SA (i.e., each HyperEdge) a communicating UML state machine that describes the behavior of each of them. The complete SA behavioral specification is composed of a set of these UML state machines. After that we use the action- and state-based temporal logic UCTL (tFGM08) to describe behavioral properties that we want to check on our SA model. Whenever the SA specification is not properly specified (*not valid* arrows in Figure 5), the SA itself needs to be revised. Thanks to the model checker we may correct the SA specification. Whenever the SA is validated (*valid* arrow in Figure 5) we can proceed to the code generation phase. The tool that supports this phase is UMC (UMC), an on-the-fly model checker for UCTL. It allows the efficient verification of UCTL formulae over a set of communicating UML state machines. We proceed describing each concept that we use in this phase starting from the principal aspects of UMC.

3.3.1 UMC Models

A complete UMC model description is given by providing by a set of class definitions and a set of object instantiations. Class definitions represent a template for the set of active or non active objects of the system. In the case of active objects a statechart diagram associated to class is used to describe the dynamic behaviour of the corresponding objects. A state machine (with its event queue) is associated to each active object of the system. Non-active objects play the role of "interfaces" towards the outside of the system, and can only be the target of signals. A system is constituted by fixed static set of objects (no dynamic object creation), and a system is an ″input closed″ system, i.e. the input source is modelled as an active object interacting with the rest of the system. There is a predefined non-active `OUT` class, and a predefined `OUT` object, which can be used to model the sending of signals to the outside of the system, and there is a predefined non-active `ERR` class, and a predefined `ERR` object, which can be used to model the notification or error signals to the outside of the system; further non-active classes and objects can be defined by defining classes without statechart. At least one active object must be defined, and the declaration of an object must appear after the declaration of the class to which it belongs. In the following section we describe in more detail the two syntactic model components, namely classes and objects, in the subsequent one we describe and overview of their semantics.

Classes and Objects

The behavior of an object belonging to a class is defined by a statechart diagram associated with the class itself. In particular, the definition of a class statechart consists of the following elements:

- the class name
- the list of events which trigger the transitions of the object of the class (signals or call operations)
- the list of attributes (variables) local to the objects of the class
- the structure of the states of the class (nodes of the statechart diagram)
- the transitions of the objects of the class (edges of the statechart diagram)

In the case of non-active objects, the corresponding class declarations can only define the list of accepted signals and operations. The events handled by the class are distinguished between *asynchronous* signals and *synchronous* call operations; the seconds can also have a return type which can be ″void″ or basic ″int″, ″bool″ and ″obj″ types or a Class name.

The attributes (local variables) of a class, and the parameters of events can explicitly typed, and the allowed types are just the "int", "bool" and "obj" types, or a Class name. Parameters can only have an implicit "in" mode. A statechart consists in a sequence of state definitions which starts from the definition of the top level state. The definition of outer states must precede the definition of the top level state. The top level state of a statechart must be a composite sequential state.

A *Composite sequential* state is defined by a list of substates (which can be composite sequential states, parallel states or simple states). The first substate of a composite state is assumed to be its default initial substate. The name "initial" denotes the default initial pseudostate (and must appear as first substate), if no initial pseudostate is explicitly provided, the first substate of the sequence is implicitly assumed as default initial substate. For any simple state, in the definition of a composite sequential state, it is not necessary to give any further explicit definition. A *composite parallel* state is defined as a parallel composition of several composite sequential substates also called *regions* of a parallel state. A state definition can also define the set of events deferred while active. A `Defers` clause defines the list of events (matching those already declared as Signals or Operations) to be referred. The definition of a transition contains a set of source states, a trigger, an optional guard, an optional list of actions and a set of target states. A transition with more than one source is called a `join` transition. In the case of joins transitions, the first state in the source list is required to be "the most transitively nested source state". In this sense the first state univoquely determinates the priority of the transition. A transition with more than one target is called a `fork` transition. The trigger of a transition can be an event declaration or an hypen symbol ("-") which means the absence of any explicit trigger (i.e., a *completion transition*). If the trigger is an event declaration with formal parameters, the name of the parameters can be used inside the actions part of the transition. `source->target` is a shortcut for `source-(-)->target`. The guard (if present) is a simple form of boolean expression involving expression involving the object attributes. The actions part can be a sequence of simple actions. Each simple action can be an assignment of an expression to a local attribute, the sending of a signal to a target object, the calling of a synchronous operation on a target object, or an assignment of a synchronous function call to a local attribute. A signal is similar to an event declaration, but its arguments are constituted by value expressions instead that by formal parameters. A signal is preceded by a target specification; the meaning is that the signal is sent to the events queue of the specified destination object (the term ″self″ can be used to denote the same object; if no declaration is specified, then ″self″ is implicitly assumed). Once the needed classes are have their behavior defined by the appropriate statechart, we can define the actual evolving system as a set of object instances. Each object instance is declared by an object declaration which introduces the object name, the name of its class, and possibly any specific initial values for its attributes. This initial values can be literals or names of other objects.

3.3.2 The Temporal Logic UCTL

The action- and state-based temporal logic, UCTL (tFGM08), is composed of the branching-time action-based logic ACTL (DV90) and the branching-time state-based logic CTL (CES86). With its syntax we are able to specify properties that should be satisfied in a state and moreover we can combine these basic predicates with temporal operators dealing with the actions performed.

$$\begin{array}{lcl} \phi & ::= & \mathit{true} \mid p \mid \phi \wedge \phi' \mid \neg\phi \mid E\pi \mid A\pi \\ \pi & ::= & X_\chi\phi \mid \phi\,{}_\chi U\,\phi' \mid \phi\,{}_\chi U_{\chi'}\,\phi' \mid \phi\,{}_\chi W\,\phi' \mid \phi\,{}_\chi W_{\chi'}\,\phi' \end{array}$$

Predicates are ranged over by p, *state formulae* are ranged over by ϕ, *path formulae* are ranged over by π and *actions* are ranged over by χ. E and A are the *path quantifiers* "exists" and "for all", resp., while X, U and W are the indexed "next", "until" and "weak until" *temporal operators*, resp.

Starting from these basic UCTL operators, it is straightforward to derive the standard logical operators $\vee$ and $\Rightarrow$, the well-known temporal logical operators EF ("possibly"), AF ("eventually") and AG ("always") and the diamond and box modalities <> (“possibly”) and [] (“necessarily”), resp., of the Hennessy-Milner logic (HM85).

The semantic domain of UCTL is a doubly labelled transition system (L^2TS for short) (DV95). An L^2TS is a labelled transition system whose states are labelled by atomic propositions and whose transitions are labelled by sets of actions.

Definition 6 (Doubly Labelled Transition System) *A* Doubly Labelled Transition System *(L^2TS for short) is a quintuple (Q, q_0, Act, R, AP, L), where:*

- *(Q, q_0, Act, R) is an LTS;*
- *AP is a set of atomic propositions with p ranging over AP; p will typically have the form of an expression like VAR = value;*
- *$L : Q \longrightarrow 2^{AP}$ is a labelling function that associates a subset of AP to each state of the LTS.*

3.3.3 The Model Checker UMC

UMC is an on-the-fly model checker for UCTL developed at ISTI (UMC), which allows the efficient verification of UCTL formulas (i.e., specifying action-and/or state-based properties) over a set of communicating UML state machines. The possible system evolution are formally represented ad an L^2TS, whose transitions represent the possible evolutions of a system configurations. More concretely, the states of this L^2TS are labelled with the observed structural properties of the system configurations (like the active substates of objects, the values of object attributes, etc.), while its transitions are labelled with the observed properties of the system evolutions (like which is the evolving object, which are the executed actions, etc.).

The big advantage of an on-the-fly approach to model checking is that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result. The basic idea underlying UMC is that, given a state of an L^2TS, the validity of a UCTL formula on that state can be evaluated by analyzing the transitions allowed in that state and the validity of a certain subformula in only some of the next reachable states, all this in a recursive way. The current version of UMC uses an on-the-fly model-checking algorithm which has a linear complexity.

Another interesting feature offered by UMC is the possibility to select a desired subset of system events or object attributes, and to show the minimized graph of all the possible system evolutions (traces) in which only the relevant labels are shown. This allows one to obtain abstract slices of the system behaviour, for which only certain kinds of interactions are considered. These abstract slices are very useful for achieving confidence in the overall correctness of the design. Since abstracted full-trace minimization of an L^2TS requires a full traversal of this L^2TS, and moreover has a high complexity, this functionality is, unfortunately, only possible for finite and reasonably sized systems.

The current UMC prototype can be experimented via a web interface (UMC).

3.3.4 UMC model of the running example

In the running example, described in section 3.1, we have *bikes* which have certain attributes (e.g., startstation, etc.), and which move across the *stations* trying to reach a given place, and we have *stations* which have its accessing bikes. Each bike can interact with a station accordingly to their respective state. A bike can migrate from one station to another station or can be stop the trip when it has reached the destination place. Stations can shut down, in which case their orphan bikes call for a repairing reconfiguration. Several configurations of this kind of system have been produced, and the version we are presenting here is a carefully simplified model. The full code of the model classes is shown in Appendix B. In more detail, we have two classes, corresponding to the two kinds of active objects in the system: class `Bike` and class `Station`. A System is supposed to be constituted by a set of Bike and Stations instantiations. A `Station` object is characterized by a list of other stations to which it is connected (i.e., left- and right-side) and by the presence (or not) of bikes connected. At each moment, if some bike is present near to the station, the station activity starts with the communication to the bikes. If some stations shut down then the bikes connected can perform the exit action from the station and entering in the next station (i.e., migration). If a station does not have bike connected, it is inactive, waiting for a bike to connect. The complete dynamic behavior of the objects of classes Bike and Station is shown below with also the UMC textual form. The statecharts for class Bike and Station are shown in Figures 18 and 19.

```
Class Bike is
-- public interface
Signals:
stationFailure(theStation: Station)
connectionBikeOK(theStation: Station),
serviceOK

-- private part
Vars:
startStation: Station,
actStation: Station,

State Top= START, WAIT, STOP, WAIT4HELP

Transitions:
START -> WAIT
{- / actStation:=startStation; startStation.connectBike(self)}
WAIT -> WAIT4HELP
{connectionBikeOK(theStation) /actStation:=theStation;
actStation.serviceRequest(self)}
WAIT -> WAIT
{stationFailure(theStation) [theStation/=null ] /
actStation:= theStation; theStation.connectBike(self)}
WAIT ->STOP
{stationFailure(theStation)[theStation=null] / OUT.BIKESTOPPED}
WAIT4HELP -> STOP
{serviceOK / OUT.BIKEDISCONNECTED}
WAIT4HELP -> WAIT
{stationFailure(theStation)[theStation/=null] /
actStation:=theStation; theStation.connectBike(self)}
WAIT4HELP -> STOP
{stationFailure(theStation) [theStation=null] / OUT.PHONECALL}

end Bike;
```

Listing 3.11: UMC Model of Class ”Bike”

```
Class Station is

Signals:
connectBike(theBike: Bike)
stopBike (theBike: Bike)
serviceRequest (theBike: Bike)

Vars:

myBike: Bike,
nextStation: Station;

State Top = EMPTY, READY, STOP

Transitions:
EMPTY -> EMPTY
{connectBike(theBike) / theBike.stationFailure(nextStation)}
EMPTY -> READY
{connectBike(theBike) / myBike:= theBike;
myBike.connectionBikeOK(self)}
READY -> STOP
{serviceRequest(theBike) / myBike:=theBike; myBike.serviceOK}
READY -> EMPTY
{serviceRequest(theBike) / myBike:= theBike;
myBike.stationFailure(nextStation)}

end Station;

Objects:
// static object instantiation
bike1: Bike (startStation -> station1, actStation -> station1)
station1: Station (nextStation-> station2)
station2: Station (nextStation->station3)
station3: Station
```

Listing 3.12: UMC Model of the Class ”Station”

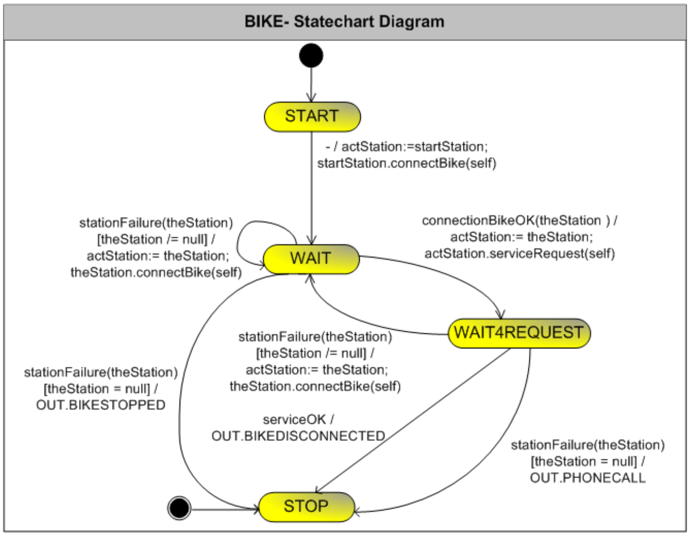

BIKE- Statechart Diagram
START
- / actStation:=startStation;
startStation.connectBike(self)
WAIT
stationFailure(theStation)
[theStation /= null] /
actStation:= theStation;
theStation.connectBike(self)
connectionBikeOK(theStation) /
actStation:= theStation;
actStation.serviceRequest(self)
WAIT4REQUEST
stationFailure(theStation)
[theStation /= null] /
actStation:= theStation;
theStation.connectBike(self)
stationFailure(theStation)
[theStation = null] /
OUT.BIKESTOPPED
serviceOK /
OUT.BIKEDISCONNECTED
stationFailure(theStation)
[theStation = null] /
OUT.PHONECALL
STOP

Figure 18: Statechart for "Bike"

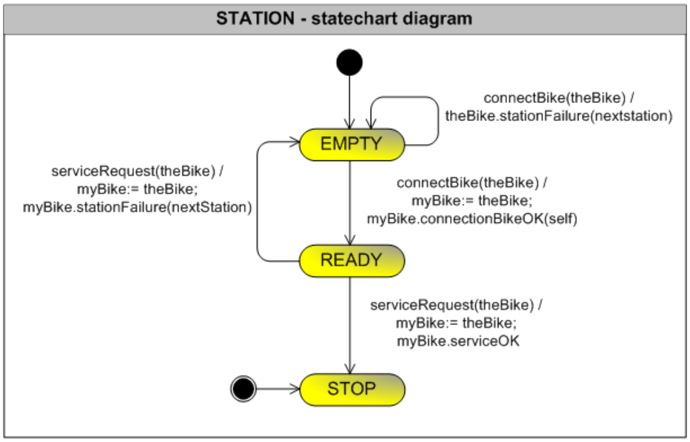

STATION - statechart diagram
connectBike(theBike) /
theBike.stationFailure(nextstation)
EMPTY
serviceRequest(theBike) /
myBike:= theBike;
myBike.stationFailure(nextStation)
connectBike(theBike) /
myBike:= theBike;
myBike.connectionBikeOK(self)
READY
serviceRequest(theBike) /
myBike:= theBike;
myBike.serviceOK
STOP

Figure 19: Statechart for "Station"

3.3.5 Validation with UMC

In this section we show how to validate our running example configurations using UMC. We have used UMC v3.5 to verify a set of behavioral properties specified using UCTL. In the following we firstly describe each of them both in an informal an formal way, after that we show the validation results.

`Station` Responsiveness A Station is *responsive* if it guarantees a response to each received request.
We want to prove that each time action `connectBike` takes place, eventually in the future the action `connectionBikeOK` or `stationFailure` takes place. In detail, if a `Bike` requests to be connected to a certain `Station`, then the `Station` will reply with a notification of either a successful or a failed connection.

```
 AG [bike1:connectionBike] AF
{bike1.connectionOK or bike1.stationFailure} true
```

We verified this formula with UMC and it is true.

`serviceRequest` Coordination It may never happen that action `serviceOK` takes place if action `serviceRequest` has not taken place before. A `Bike` cannot receive a notification that some service has been provided, if it was not previously requested to the `Station`.

```
A [true {not serviceOK} W {serviceRequest} true].
```

We verified this formula with UMC and it is true.

`connectionBike` Reliability Each time action `connectionBike` takes place, eventually in the future `connectionBikeOK` takes place. It means that connection requests from `Bike` to `Station` are always followed by a notification of success.

```
AG [bike1:connectionBike] AF {bike1.connectionOK } true
```

We verified this formula with UMC and it is false. This is not surprising: each `Station` of the system might be temporarily unable to provide the requested connection (they send `stationFailure` response).

`serviceRequest` Uniqueness Each time action `serviceRequest` takes place, then `stationFailure` or `serviceOK` takes place and afterward it may not happen that eventually one of the latter two actions takes place again. It means that after a service request from `Bike` to `Station`, it cannot happen that the `Bike` receives more than one notification.

```
AG[bike1:serviceRequest]
    not EF <bike1.serviceOK or bike1.stationFailure>
    EF <bike1.serviceOK or bike1.stationFailure> true
```

We verified this formula with UMC and it is true.

With UMC, after this verification we can also generate the abstract traces of the system obtained by observing the interactions among one Bike (bike1) and three Stations (s1,s2,s3) of the running example SA configuration. Such graph gives a precise and complete view of the system execution behavior. It is depicted in Figure 20

3.4 Architectural-based Code Generation

Whenever the SA is validated with respect to the desired properties, Java code can be automatically generated from the SA specification. According to Figure 21, this activity is performed through two main steps: starting from a validated *UMC specification*, ArchJava *code* is automatically obtained by means of a *JET-based Code Generator*. Then, by exploiting the existing ArchJava *Compiler*, executable *Java code* is generated. Here we focus on the first step of the translation in Figure 21 which is based on the following directives:

1. Each UMC class becomes an ArchJava component. For instance, the component Bike in Figure 22(a) induces the following ArchJava specification:

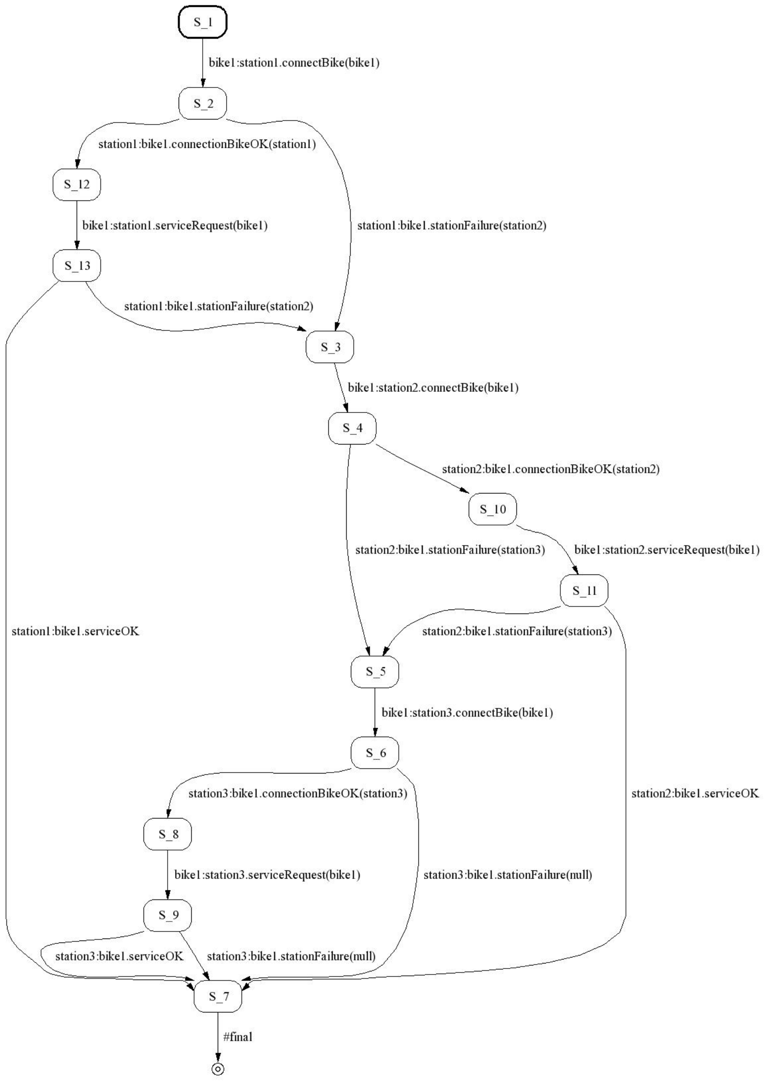

S_1
bike1:station1.connectBike(bike1)
S_2
station1:bike1.connectionBikeOK(station1)
S_12
bike1:station1.serviceRequest(bike1)
station1:bike1.stationFailure(station2)
S_13
station1:bike1.stationFailure(station2)
S_3
bike1:station2.connectBike(bike1)
S_4
station2:bike1.connectionBikeOK(station2)
S_10
station2:bike1.stationFailure(station3)
bike1:station2.serviceRequest(bike1)
S_11
station1:bike1.serviceOK
station2:bike1.stationFailure(station3)
S_5
bike1:station3.connectBike(bike1)
S_6
station3:bike1.connectionBikeOK(station3)
station2:bike1.serviceOK
S_8
bike1:station3.serviceRequest(bike1)
station3:bike1.stationFailure(null)
S_9
station3:bike1.serviceOK
station3:bike1.stationFailure(null)
S_7
#final

Figure 20: Abstract behavioural slice

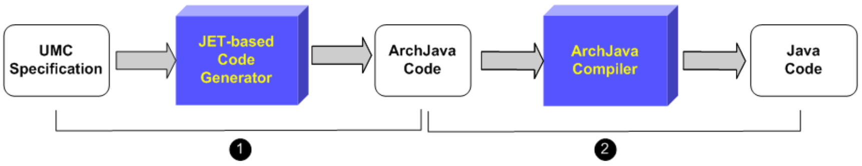


Figure 21: JET/ArchJava Code Generation

```
public component class Bike{
...
}
```

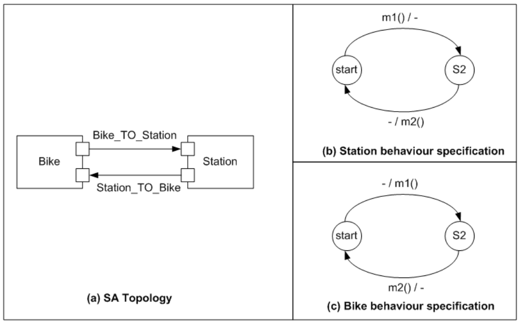


Figure 22: Sample UMC specification

2. Each UMC component's (i.e., classes) sent and received message is used to synthesize the component ports. We recall that ArchJava has both ports for provided services and ports for required services. An ArchJava port only connects a pair of components. This means that if a component needs to communicate with more than one component, it needs additional ports. Thus, the provided component services are partitioned into sets of services provided to different components. The same is done for required services. Accordingly, the suitable number of required and provided ports is declared into the ArchJava specification of the component (containing the declaration of required and provided services, respectively). For instance, the sample SA in Figure 22 gives place to the following ArchJava code fragments concerning the Bike component implementation:

```
public port Bike_TO_Station {
  requires void m1();
}
public port Station_TO_Bike {
  provides void m2();
}
```

3. For each UMC component an ArchJava specification is generated to encode the associated state diagram. ArchJava does not offer a direct support for that and we propose guidelines to extend the ArchJava specifications so that a state diagram associated to a software component is implemented as an adjacency list. In particular:

   - for each method invoked by a given component the corresponding state diagram changes state accordingly so having trace of what methods can be invoked or not. States and transitions of the considered state diagram are declared as Java constants and are used to univocally refer to these elements (see lines 3-9 in the code below).
   - each state machine contains a fixed definition of transitions as an internal Java class (see line 15-37 in the code below). The

state diagram is defined as a *LinkedList*. The constructor of the state diagram class contains the definition of the state machine adding to the *LinkedList* of the state diagram an element for each state containing all existing transitions (for each existing transition a new object of the internal class transition is added) (see lines 40-50).

- each state diagram class contains also a method that simulates the transition fire, i.e., this method gets as input the transition (according to the runtime behaviour of the system) and checks if it is possible, in the actual computation state, to perform the transition fire (see lines 52-63). If the behaviour is allowed then the actual state is updated to the transition target state, otherwise an exception is raised. In case a method cannot be invoked in a certain time, an exception is raised. The exception is defined as an additional ARCHJAVA specification, i.e., a java class extending the java.lang.Exception class.

4. A main ARCHJAVA specification is also generated to define the binding among components ports and the instantiation of the involved state machines.

These directives ensure the *communication integrity*, i.e., components can only communicate using connections and ownership domains that are explicitly declared in the SA. The rest of the section outlines the approach supporting the automatic generation of code that implements such directives. This automation is required since manual coding could diverge or not completely adhere to them. As previously said, the code generator implementing the four directives above has been developed in JET(BBM03). JSP-like templates define explicitly the target ARCHJAVA code structure and get the data they need from the UMC models. In particular, the code generator consists of four templates: *main.jet* is a default template that gets data as input and applies the other templates. Being more precise, it applies the *componentMain.jet* template, that implements the directive 4 previously described, producing the target *MAIN.archj* file (see line 2 in Figure 24). Then, for each component in the source UMC

```
public class SM_Bike {
  /** State encoding*/
  public final int S_start= 0;
  public final int S_WAIT= 1;
  public final int S_WAIT4REQUEST= 2;
  public final int S_STOP= 3;

  /** Transition encoding */
  public final int T_startStation.connectBike=0;
  public final int T_stationFailiure=1;
  public final int T_theStation.connectBike=2;
  public final int T_connectionBikeOK=3;
  public final int T_stationFailure=4;
  public final int T_actStation.serviceRequest=5;
  public final int T_serviceOK=6;
  public final int T_OUT.BIKEDISCONNECTED=7;
  public final int T_OUT.BIKESTOPPED=8;
  public final int T_OUT.PHONECALL=9;

  private int currentState=S_start;
  private LinkedList states = new LinkedList();

  private class transition{
    private int state;
    private int transition;
    private int send_receive;

    public transition(int transition, int state, int send_receive){
      this.transition=transition;
      this.state=state;
      this.send_receive=send_receive;
    }
    public int getTransition(){
      return transition;
    }
    public int getState(){
      return state;
    }
    public int getSendReceive(){
      return send_receive;
    }
  }
```

```
 /** State Machine constructor*/
 public SM_Bike(){
  System.out.println("SM_Bike.constr");

  LinkedList start = new LinkedList();
  start.add(new transition(T_startStation.connectBike, S_WAIT ,0));
  states.add(start);

  LinkedList WAIT = new LinkedList();
  WAIT.add(new transition(T_stationFailiure, S_WAIT ,1));
  WAIT.add(new transition(T_theStation.connectBike, S_WAIT ,0));
  WAIT.add(new transition(T_connectionBikeOK, S_WAIT4REQUEST ,1));
  WAIT.add(new transition(T_theStation.connectBike, S_WAIT ,0));
  WAIT.add(new transition(T_stationFailure, S_STOP ,1));
  states.add(WAIT);

  LinkedList WAIT4REQUEST = new LinkedList();
  WAIT4REQUEST.add(new transition(T_stationFailure, S_WAIT ,0));
  WAIT4REQUEST.add(new transition(T_actStation.serviceRequest,
             S_WAIT4REQUEST ,0));
  WAIT4REQUEST.add(new transition(T_serviceOK, S_STOP ,1));
  WAIT4REQUEST.add(new transition(T_stationFailure, S_STOP ,1));
  states.add(WAIT4REQUEST);

  LinkedList STOP = new LinkedList();
  STOP.add(new transition(T_OUT.BIKEDISCONNECTED, S_STOP ,0));
  STOP.add(new transition(T_OUT.BIKESTOPPED, S_STOP ,0));
  STOP.add(new transition(T_OUT.PHONECALL, S_STOP ,0));
  states.add(STOP);
}

 public void transFire(int trans) throws SMException {
    LinkedList transitions=(LinkedList)states.get(currentState);
    for(int i=0;i<transitions.size();i++){
     if(((transition)transitions.get(i)).getTransition()==trans){
     currentState=((transition)transitions.get(i)).getState();
     System.out.println("Bike.transallowed:␣" + trans
                 + ",␣state␣:␣" + currentState);
     return;
    }
   }
   System.out.println("trans␣not␣allowed:␣" + trans);
   throw new SMException();
 }
}
```

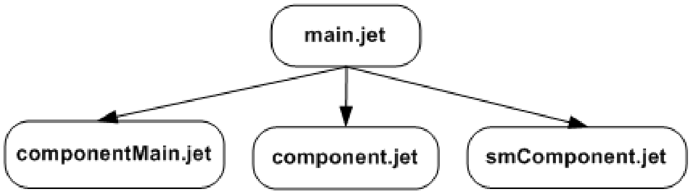
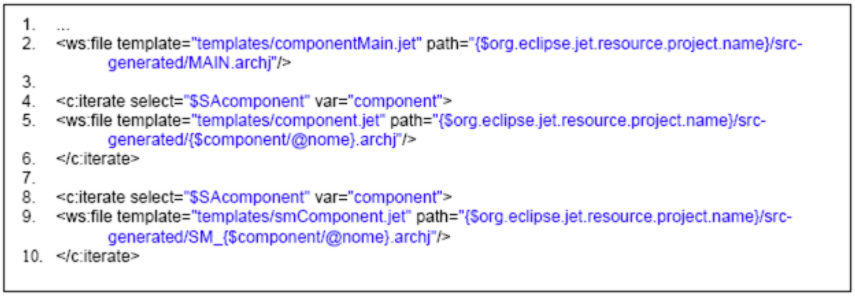


Figure 23: JET-based Code Generator templates



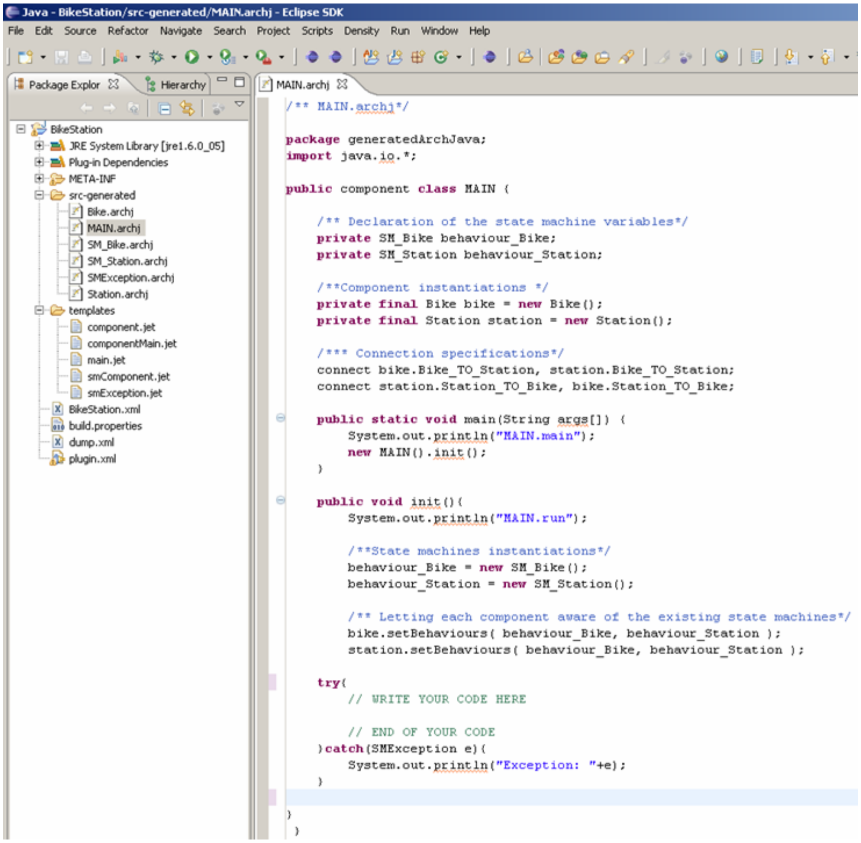
```
1.  ...
2.  <ws:file template="templates/componentMain.jet" path="{$org.eclipse.jet.resource.project.name}/src-
          generated/MAIN.archj"/>
3.
4.  <c:iterate select="$SAcomponent" var="component">
5.  <ws:file template="templates/component.jet" path="{$org.eclipse.jet.resource.project.name}/src-
          generated/{$component/@nome}.archj"/>
6.  </c:iterate>
7.
8.  <c:iterate select="$SAcomponent" var="component">
9.  <ws:file template="templates/smComponent.jet" path="{$org.eclipse.jet.resource.project.name}/src-
          generated/SM_{$component/@nome}.archj"/>
10. </c:iterate>
```

Figure 24: Fragment of the main.jet template

specification, the *component.jet* template is applied in order to generate the component implementation according to points 1 and 2 above (see line 4-6 in Figure 24). Finally, for each source component the corresponding state machine encoding is generated by applying the *smComponent.jet* template that implements point 4 (see line 8-10).

Due to space limitation, the templates are not reported here. However, interested readers can refer to (CHA) for downloading the full implementation of the proposed JET-based code generator. The Application of the *JET-based Code Generator* on the UMC specification of the Running Example has produced a number of ArchJava files listed on the left-hand side of the screenshot in Figure 25. In particular, for each component (i.e., Bike and Station), the corresponding encoding is generated (e.g., *Bike.archj*, *Station.archj*). The state machine specifications are also synthesized (i.e., *SM_Bike.archj*, *SM_Station.archj*) together with a MAIN.archj file (listed on the right-hand side of Figure 25) that enables the execution of the obtained system with respect to the modelled software architecture. Being more precise, in that main file all the components, the corresponding state machines and port connections are instantiated giving place to an encoding of the SA properties that constraint the execution of the hand-written code that will be filled in prearranged points (e.g., see the *try* statement in the code of Figure 25.

The complete code of the running example is listed in Appendix C.



```
/** MAIN.archj*/

package generatedArchJava;
import java.io.*;

public component class MAIN {

    /** Declaration of the state machine variables*/
    private SM_Bike behaviour_Bike;
    private SM_Station behaviour_Station;

    /**Component instantiations */
    private final Bike bike = new Bike();
    private final Station station = new Station();

    /*** Connection specifications*/
    connect bike.Bike_TO_Station, station.Bike_TO_Station;
    connect station.Station_TO_Bike, bike.Station_TO_Bike;

    public static void main(String args[]) {
        System.out.println("MAIN.main");
        new MAIN().init();
    }

    public void init(){
        System.out.println("MAIN.run");

        /**State machines instantiations*/
        behaviour_Bike = new SM_Bike();
        behaviour_Station = new SM_Station();

        /** Letting each component aware of the existing state machines*/
        bike.setBehaviours( behaviour_Bike, behaviour_Station );
        station.setBehaviours( behaviour_Bike, behaviour_Station );

    try(
        // WRITE YOUR CODE HERE

        // END OF YOUR CODE
    }catch(SMException e){
        System.out.println("Exception: "+e);
    }

}
 }
```

Figure 25: Generated Code Overview of the Running Example.

Chapter 4

Automotive Case Study

Much of the cost of research and development in vehicle production are associated to automotive software. Today vehicles are equipped with a multitude of sensors and actuators that provide different services, like `ABS` and vehicle stabilization systems, that assist people to drive safer. Thanks to current mobile technology, vehicles have the possibility to connect to the telephone and internet infrastructures. This has given birth to a variety of new services into the automotive domain. Communication in automotive software systems can takes place inside a vehicle (*intra-vehicle*), connection to vehicles in the vicinity (*inter-vehicle*) or interaction with the environment, for example through an Internet gateway (*vehicle-environment)*. The case study presented in this Chapter, and at which we apply the *traffic light* process, is expired to a lot of documents within to the Sensoria project (BK07; Koc07; EU) and some published papers (BFGL07; HKMU06; tGKM08).

4.1 Route Planning System

4.1.1 Introduction

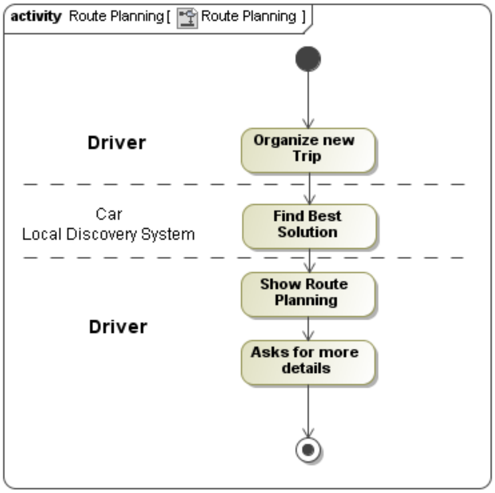
Route Planning System (RPS) is responsible for providing guiding indications to the driver. In particular it must be able to provide following functionalities:

- **Route Planning**: each vehicle plans the trip autonomously by using the information provided by a GPS system in the vehicle or internal information already present. Based on the driver's preferences that were given at the beginning of the trip, the RPS searches a sight seeing database for appropriate sights and displays them on the in-car map of the vehicles' navigation system. The driver clicks on sights he would like to visit which results in the display of more detailed information about this specific sight (e.g. opening times, guidance to parking, etc.). Figure 26 depicts activities of this scenario.

- **Low Oil Scenario**: During a drive, the vehicles oil lamp reports low oil levels. This triggers the in-vehicle diagnostic system to perform an analysis of the sensor values. The diagnostic system reports a problem with the pressure in one cylinder head, and that the car is no longer drivable, and sends a message with the diagnostic data as well as the vehicle's GPS data to the Service Discovery System that finds a best solution (i.e., Towing Service, Repair Shop or Rent a Car). If the car can not be moved the tow truck will be called , otherwise the repair shop (or RentACar Shop) coordinates are sent to the vehicle guiding system to direct it to the shop. The driver's user interface, from this point, is reconfigured to receive instructions from an external system (i.e., Repair or RentACar Shop). Figure 26 depicts activities of this scenario.

- **Reconfigurable Route Planning**: Steven and John are on their way to Italy in separate cars. Both want to spend their holidays together. John has entered the destination into his navigation system which is calculating and providing the best route during the travel. To make sure both cars take the same route, Steven's navigation system just receives route planning information from John's instead of performing route planning itself.

- **AirBag Scenario** A driver subscribed to the accident assistance service available for all vehicles of the car manufacturer. Due to a head on collision, the vehicles airbag is deployed which triggers an automated message to the Road Assistance Service (RASS) that contains



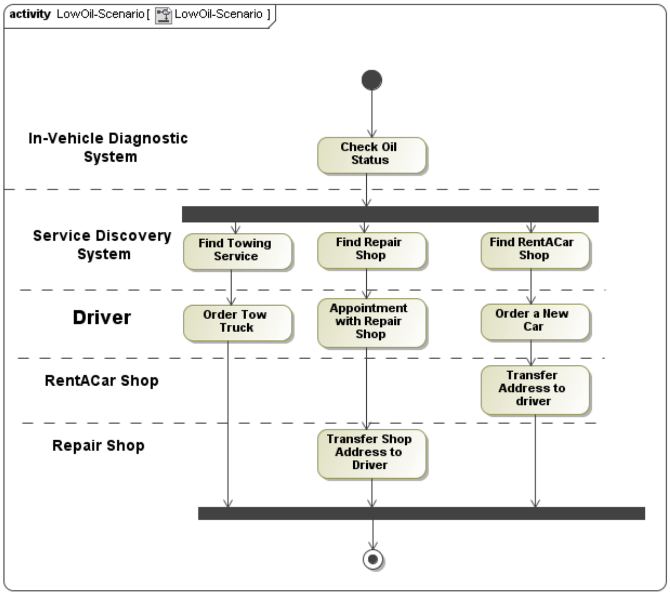

activity LowOil-Scenario [LowOil-Scenario]
In-Vehicle Diagnostic System
Check Oil Status
Service Discovery System
Find Towing Service
Find Repair Shop
Find RentACar Shop
Driver
Order Tow Truck
Appointment with Repair Shop
Order a New Car
RentACar Shop
Transfer Address to driver
Repair Shop
Transfer Shop Address to Driver

Figure 26: Activities of the Route Planning and Low Oil Scenarios

the vehicle's GPS data, the vehicle identification number and a collection of sensor data like for example the number of seatbelts in use and impact sensors for critical vehicle parts. The RASS places a call to the driver's mobile phone. Due to his injures, the driver is unable to answer the call. Based on the sensor data available to the RASS, the severity of the accident is assessed and emergency services (police, ambulance) are alerted and provided with the vehicles location. Accident information is successively passed on the next approaching vehicles.

4.1.2 Use Case Model

We use uses cases to represent functional requirements. Figure 27 shows the UML2.0 use case diagram. The Driver, a GPS and Road Assistance Service (RASS) are actors of the RPS. RASS provides the three on road services needed in the case the car cannot be driven, i.e. car repair, tow trucking, car renting, driver and emergency service calling. Moreover it sends instructions directly to the driver redirecting the vehicle to avoid traffic (or accident) problems. Each driver can plan the trip autonomously by using the information provided by a GPS system in the vehicle or internal information already presented in the RPS. Finally each driver can advices another vehicle behind of him providing, for example, route planning informations.

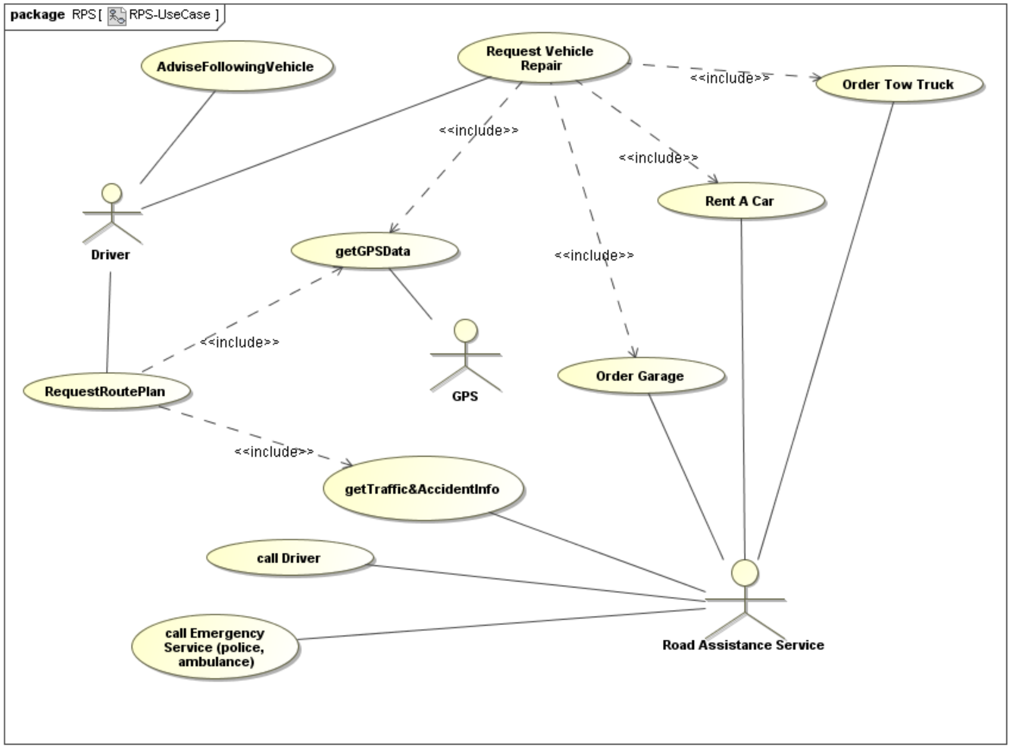


Figure 27: Use case model for the RPS.

4.2 RPS Structural Design and Analysis

In this section we execute the first step of the *traffic light* process introduced in section 2.6. We define components and style of the RPS system using the Typed Graph Grammar (TGG) approach and after we implement all aspects using Alloy. The architecture of the system is composed of six components as shown in Figure 28. A detailed description of each component with the respective behavior will be described in Section 4.3

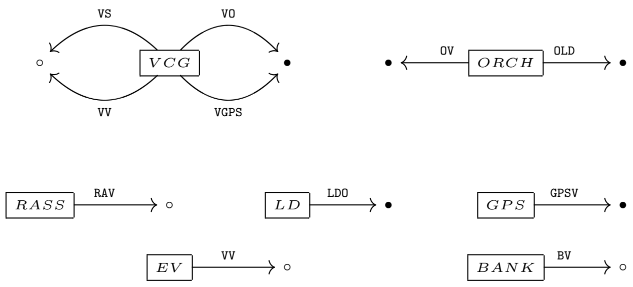


Figure 28: Basic Components of RPS

In order to understand each name assigned to each port and connections, Tables 10 and 11 describe them.

Connection	Abbreviation	Graphical Symbol
Environment Access	EA	○
Internal Access	IA	●

Table 10: Connections Description

4.2.1 RPS Style

After the identification of each component that compose the system, we have defined the Automotive Style at which each SA configuration should be conformed. To do this this we have defined an Alloy module that implements the `TypeGraph` of Figure 29. The Alloy code of the RPS Type

Abbreviation	Description
VV	Vehicle - Vehicle
VS	Vehicle - Service
VO	Vehicle - Orchestrator
VGPS	Vehicle - GPS
OLD	Orchestrator - Local Discovery
OV	Orchestrator - Vehicle
BV	Bank - Vehicle
RAV	Road Assistance - Vehicle
LDO	Local Discovery - Orchestrator
GPSV	GPS - Vehicle

Table 11: Ports Description

Graph is listed in Listing 4.1.

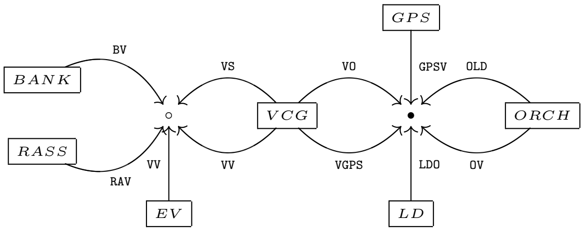


Figure 29: Type Graph of RPS

During the style specification we realised that constraints defined in it were not sufficient. The Alloy Analyzer was able to generate configurations with wrong connections. For this reason we have modified it adding more constraints on connections among components, connections multiplicities etc. In the code 4.2 we present only a sub-set of these. The complete code is listed in the Appendix A.

```
module AutomotiveStyle

open TGG
open SOFTWARE

//-------------TYPEGRAPH DEFINITION------------------
//Bike-Style basic elements
one sig EA, IA extends Node {}
one sig BV, RAV, VS,VV, VO, VGPS,
      OLD, OV, GPSV, LDO extends Tentacles {}
one sig BANK extends Edge{}
{
 tentacles = BV
 conn = BV->EA
}
one sig RASS extends Edge{}
{
 tentacles = RAV
 conn = RAV->EA
}
one sig VCG extends Edge{}
{
 tentacles = VS + VV + VO + VGPS
 conn = VS ->EA + VV->EA + VO->IA + VGPS->IA
}
one sig GPS extends Edge{}
{
 tentacles = GPSV
 conn = GPSV->IA
}
one sig LD extends Edge{}
{
 tentacles = LDO
 conn = LDO->IA
}
one sig ORCH extends Edge{}
{
 tentacles = OLD+OV
 conn = OLD->IA + OV->IA
}
fact onTypeGraph
{
 TypeGraph.n = EA+IA
 TypeGraph.he = BANK + RASS + VCG + GPS + LD + ORCH
}
```

Listing 4.1: RPS Type Graph in Alloy

```

// Bank-VCG connections
all tg: TypedGraph |
 one morph: tg.typingmorphism |
 one g1: morph.source |
 all e1,e2: g1.he | #(univ.(e1.conn)& univ.(e1.conn)) =1 and
              morph.fE[e1] =  BANK and morph.fE[e2] = VCG
              => VS in univ.conn.(univ.(e1.conn))

// RASS-VCG connections
all tg: TypedGraph |
 one morph: tg.typingmorphism |
 one g1: morph.source |
 all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e1.conn)) =1 and
              morph.fE[e1] =  RASS and morph.fE[e2] = VCG
              => VS in univ.conn.(univ.(e1.conn))

// EA connections
all tg: TypedGraph |
 one morph: tg.typingmorphism |
 one g1: morph.source |
 all n: g1.n | morph.fN[n] = EA => #(conn.n) <=2

// ORCH-LD connections
all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1 and
      morph.fE[e1] =  LD and morph.fE[e2] = ORCH
      => OLD in univ.conn.(univ.(e1.conn))
```

Listing 4.2: RPS Style Constraints

RPS Model Finding

To generate style-conformant SAs configurations, in the Listing 4.3, we open the modules `TGG` and `AutomotiveStyle` (lines 4-5). When we execute the run command (line 29) the Alloy Analyzer firstly verifies each constraints defined in both modules and after generates all possible configurations. Figure 30 presents only three of them.

```
1  module MODELFINDING
2
3  open TGG
4  open SOFTWARE
5  open AutomotiveStyle
6
7  one sig orch1 extends Edge{}
8  one sig ld1 extends Edge{}
9  one sig vcg1 extends Edge{}
10 one sig bank1 extends Edge{}
11 one sig ev1 extends Edge{}
12
13
14 one sig ia1,ia2,ia3,ea1,ea2 extends Node{}
15 one sig InitialGraph extends Graph{}
16 {
17   he= orch1+vcg1+bank1+ev1+ld1
18   n= ia1+ia2+ia3+ea1+ea2
19 }
20 one sig InitialConfiguration extends TypedGraph{}
21 {
22   typingmorphism.source = InitialGraph
23   typingmorphism.fE = orch1->ORCH + vcg1->VCG +
24                       bank1->BANK + ev1->EV + ld1->LD
25   typingmorphism.fN = ia1->IA + ia2->IA + ia3->IA +
26                        ea1->EA + ea2->EA
27 }
28 pred AutomotiveConfiguration[]{}
29 run AutomotiveConfiguration for 3
```

Listing 4.3: RPS Model Finding

RPS Reconfigurations

Figure 31 and 32 showS different graph transformation rules that we can use to reconfigure a particular SA configuration in a programmed DSA. `LOCAL REQUEST` describes the driver internal service request. In a certain time the driver could ask informations about cinema or restaurant

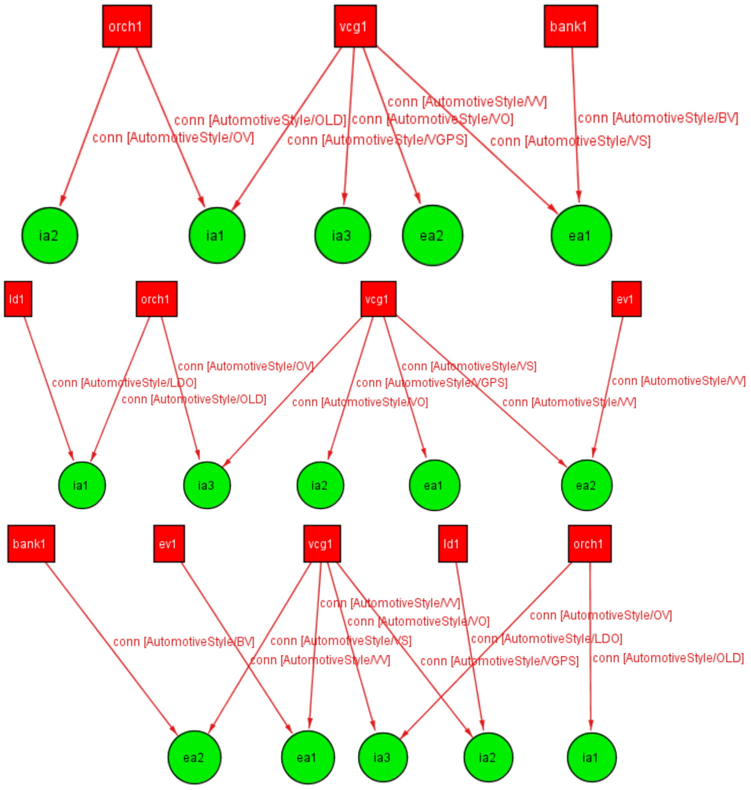

orch1
vcg1
bank1
conn [AutomotiveStyle/VV]
conn [AutomotiveStyle/OLD]
conn [AutomotiveStyle/VO]
conn [AutomotiveStyle/BV]
conn [AutomotiveStyle/OV]
conn [AutomotiveStyle/VGPS]
conn [AutomotiveStyle/VS]
ia2
ia1
ia3
ea2
ea1
ld1
orch1
vcg1
ev1
conn [AutomotiveStyle/OV]
conn [AutomotiveStyle/VS]
conn [AutomotiveStyle/LDO]
conn [AutomotiveStyle/VGPS]
conn [AutomotiveStyle/VV]
conn [AutomotiveStyle/OLD]
conn [AutomotiveStyle/VO]
conn [AutomotiveStyle/VV]
ia1
ia3
ia2
ea1
ea2
bank1
ev1
vcg1
ld1
orch1
conn [AutomotiveStyle/VV]
conn [AutomotiveStyle/OV]
conn [AutomotiveStyle/VO]
conn [AutomotiveStyle/BV]
conn [AutomotiveStyle/VS]
conn [AutomotiveStyle/LDO]
conn [AutomotiveStyle/VV]
conn [AutomotiveStyle/VGPS]
conn [AutomotiveStyle/OLD]
ea2
ea1
ia3
ia2
ia1

Figure 30: Three possible Style-Conformant SA configuration of the RPS System

that are near its GPS location. RPS searches, through the Local Discovery component (ld1 in the rule), a set of appropriate data and displays them to the driver. `BANK REQUEST` describes a request from the driver to him/her personal Bank (bank1 in the rule). For example each Driver can recharge personal mobile phone credit only communicating the number of the personal credit card. `GPS REQUEST` describes a request from the driver to the GPS System. `CALL FRIEND` is executed when a vehicle sends route planning information to another vehicle that is in the same route with the objective to arrive in the same place. `REQUEST ASSISTANCE` is executed when an accident happen. A driver subscribed to the Road Assistance Service (rass1 in the rule) can call it when there is some problem in the car or after some collision. Based, for example, on the severity of the accident, RASS can call different emergency services (police, ambulance) or can fix an appointment to the nearest garage, etc.

RPS Structural Analysis

Programmed DSA In the Listing 4.4 we present an example of Programmed DSA (e.g.,pdsa1) of the RPS system, where its initial Configuration is `G1` and the set of possible reconfigurations is composed of three productions.

```
one sig pdsa1 extends ProgrammedDSA{}
{
 first = G1
 transitions.trigger = $GPS_REQUEST + CALL_FRIEND + REQUEST_ASSISTANCE$
}
pred show[]{}
run show for 4 but 1 ProgrammedDSA
```

Listing 4.4: RPS Programmed DSA Example

In Figure 33 we can see the Alloy Analyzer output and the graphical representation of the example. The start configuration of the programmed DSA is `G1` while `G4` is the final. The latter is generated after three reconfiguration steps.

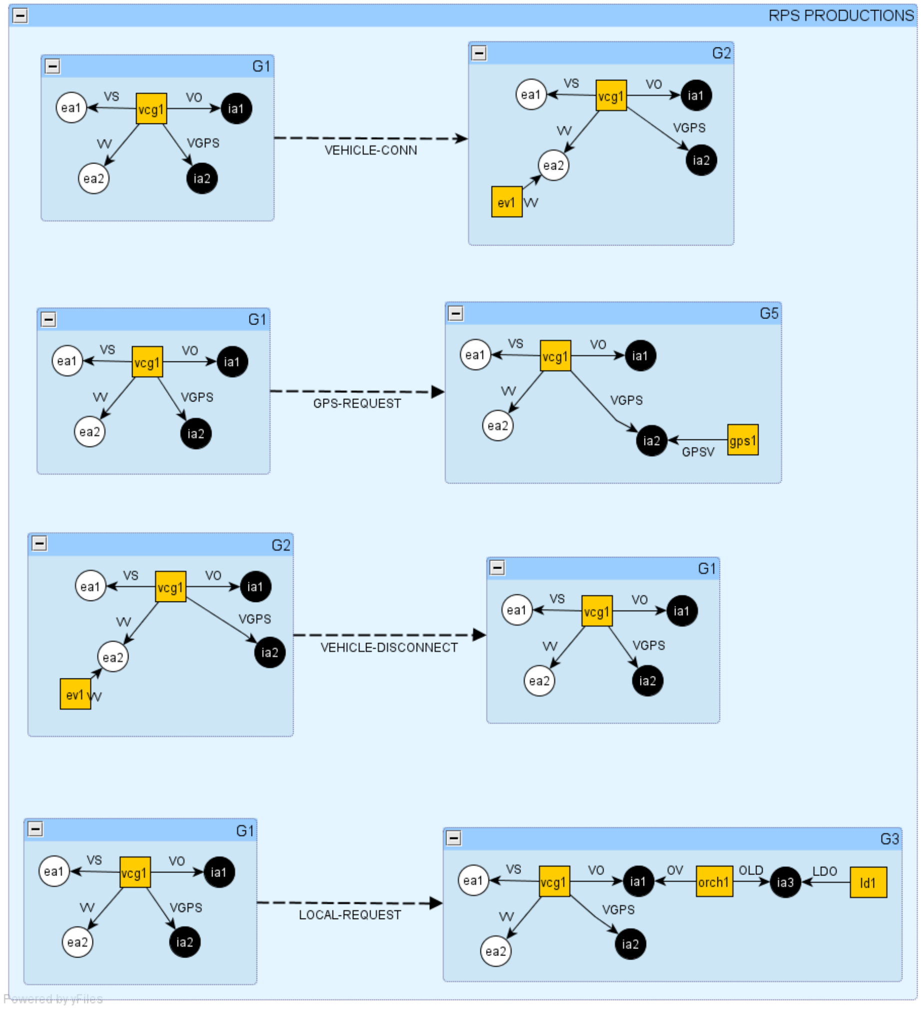


Figure 31: RPS PRODUCTIONS - I

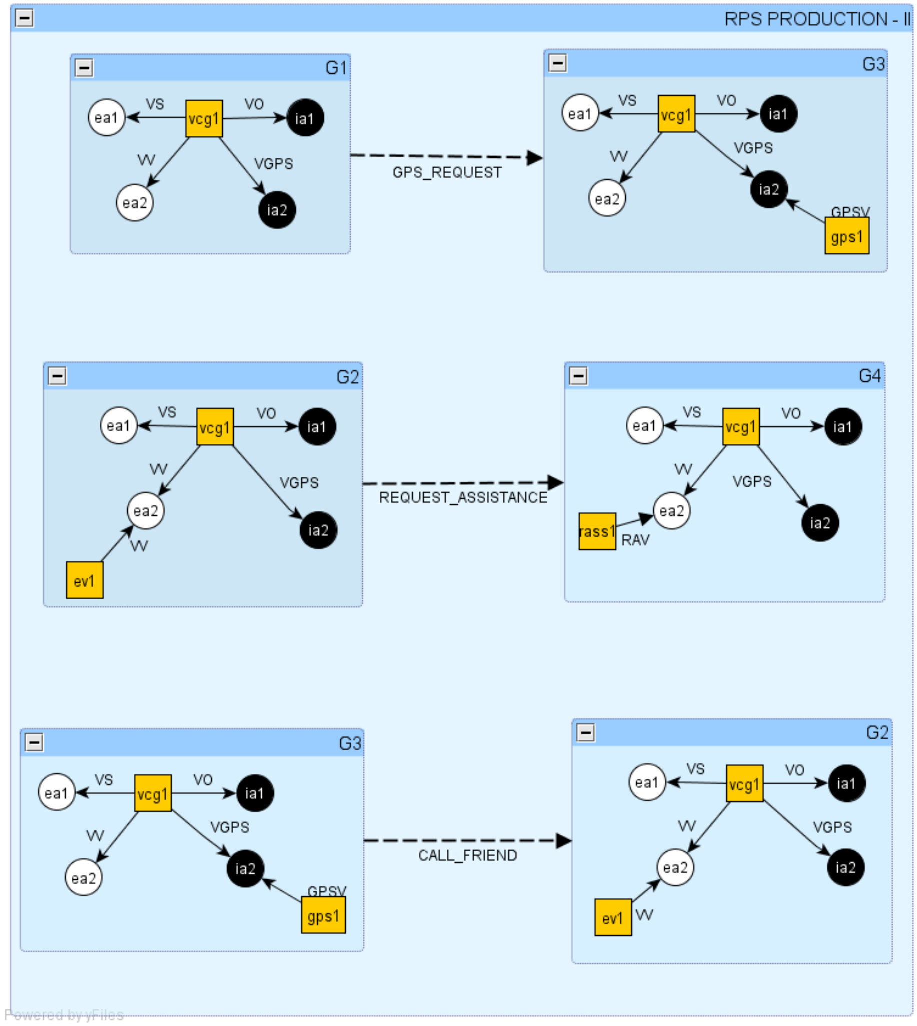


Figure 32: RPS PRODUCTIONS - II

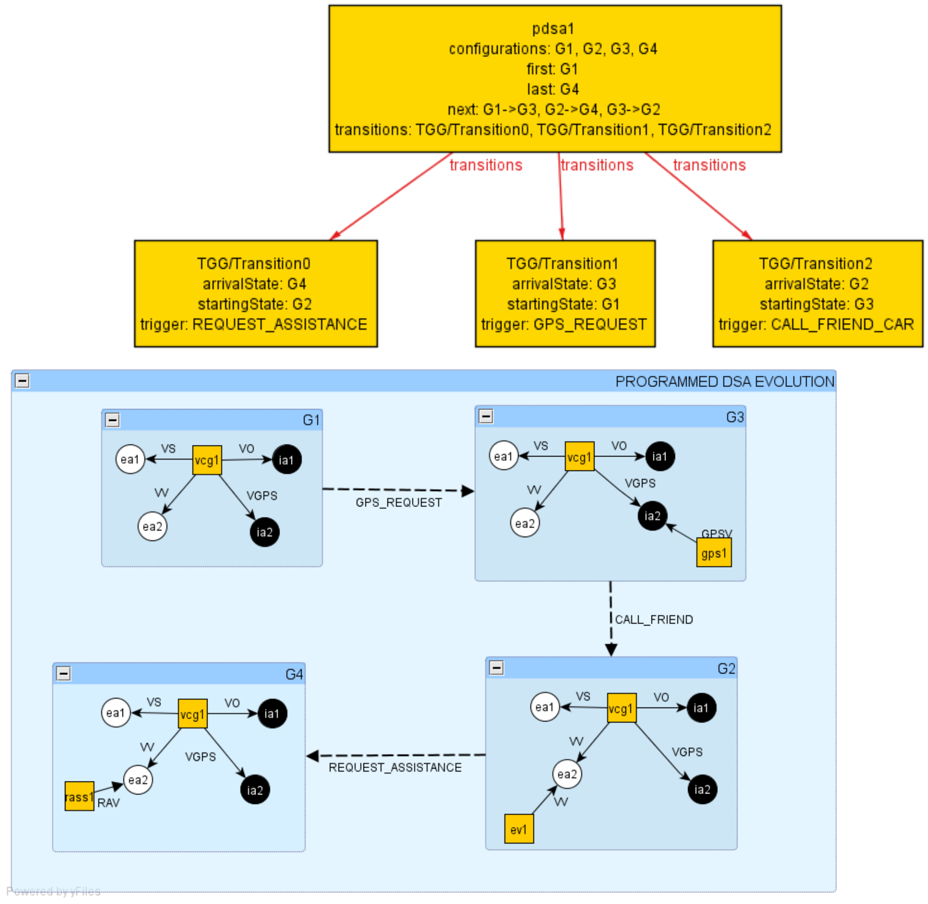


Figure 33: Programmed DSA Example

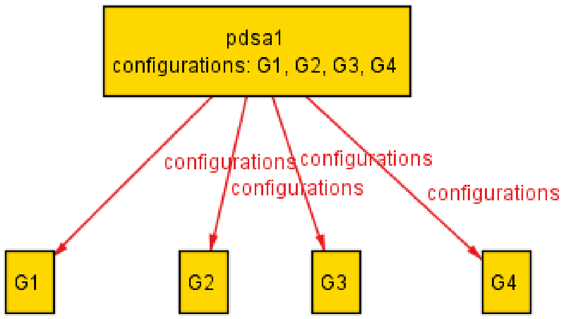


Figure 34: Reachable Configurations of the RPS System

Reachability We have used the Reachability predicate defined for the running example in order to check if some configuration is reachable. We have executed it for the `G3` configuration. The Alloy Analyzer outputs is `"No counterexample found"`, it means that `G3` is a reachable configuration in the RPS system. Figure 34 presents the set of *Reachable Configurations* of the Programmed DSA `pdsa1`. This fact demonstrates that we can check easily if a particular configuration is reachable. In order to have a correct and complete RPS system we must verify that each reachable configuration is desirable. We demonstrate this using the *Invariant Analysis* of the RPS system.

Invariant Analysis For this kind of analysis we have defined a set of properties that each DSA configuration, of the RPS system, after a rewriting step must satisfy. In the following we describe each of them with the Alloy code related

1. **Property 1**: each VCG component can not be connected to a component LD;

2. **Property 2**: When a component LD exist then must exist also a componet ORCH and both must be connected;

3. **Property 3**: the component ORCH has exactly two connections;

4. **Property 4**: a VCG component can not have more than one component attached to each port;

```
pred Property1[conf: TypedGraph]
{
  all e1,e2: conf.typingmorphism.source.he |
  conf.typingmorphism.fE[e1] =VCG and
  conf.typingmorphism.fE[e2] =LD
  =>
  #((e1.conn)&(e2.conn))=0
}
pred Property2 [conf:TypedGraph]
{
  all e1: conf.typingmorphism.source.he |
  conf.typingmorphism.fE[e1] =LD
  =>
  one e2: conf.typingmorphism.source.he |
  conf.typingmorphism.fE[e2] =ORCH and
  #((e1.conn)&(e2.conn)) =1
}
pred Property3 [conf: TypedGraph]
{
  all e1: conf.typingmorphism.source.he |
  conf.typingmorphism.fE[e1] =ORCH
  =>
  #(e1.conn) = 2
}
pred Property4[conf:TypedGraph]
{
  all e1: conf.typingmorphism.source.he |
  conf.typingmorphism.fE[e1] =VCG
  =>
  all n: univ.(e1.conn) | #(conn.n)<=2
}
```

Listing 4.5: Invariant Properties

If we check the Invariant Analysis predicate for each property in Listing 4.5 we have that each property is valid for each reachable configuration of our running example. Considering the definition presented in Section 3.2.2 we can conclude that the specification of the running example is correct and complete.

4.3 RPS Behavioural Design and Analysis

The first thing that we have done, in this phase of the process, is to specify the behaviour of each component that compose the RPS system. In the

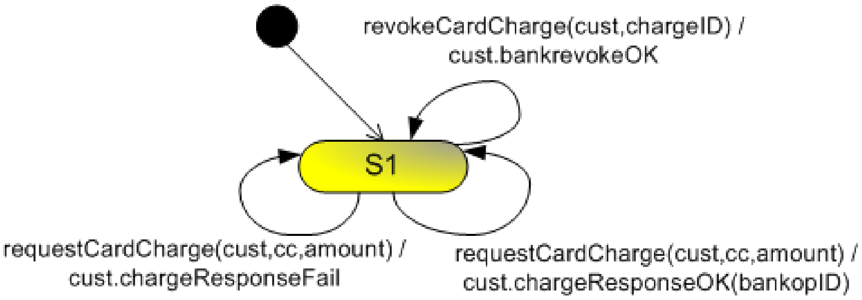

revokeCardCharge(cust,chargeID) / cust.bankrevokeOK
S1
requestCardCharge(cust,cc,amount) / cust.chargeResponseFail
requestCardCharge(cust,cc,amount) / cust.chargeResponseOK(bankopID)

Figure 35: Bank State Machine

following we present them with a shortly definition accompanied with a graphical State Machine description.

Bank Component

The Bank represents an institution that provides financial services. The bank operations that are relevant for the RPS application are the charge of a credit card and the revoke of a charge. Figure 35 depicts the Bank state machine describing the behaviour.

GPS Component

The GPS is provider of data from the Global Positioning System. The service relevant for the RPS application is the request of location, i.e. the current position of the vehicle. Figure 36 shows the state machine describing its behavior.

Local Discovery Component (LD

The Local Discovery component look fo appropriate services in the local repository. The state machine describing the behaviour is shown in Figure 37

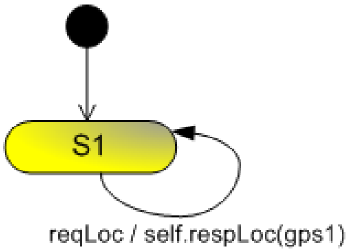


Figure 36: GPS State Machine

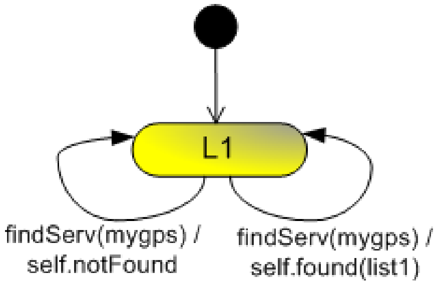

L1
findServ(mygps) / self.notFound
findServ(mygps) / self.found(list1)

Figure 37: Local Discovery State Machine

Road Assistance Component (RASS)

The Road Assistance component provides all required services for car repairing (Garage Service, Tow Truck Service and Rental Service). Its state machine is composed of three differet sub-machines, each one for each service that a drivere can request. The overall state machine is depicted in Figure 38.

Vehicle Communication Gateway (VCG)

The Vehicle Communication Gateway models sending messages to external components. These components can be other cars or the Banks, the Road Assistance that will be in charge of finding appropriate services, the selected TowTruck, Garage and RentACar service providers. The state machine describing the behavior of this component is composed of three different sub-machines, each one for each kind of communication among the vehicle and the environment, in Figure 39 we present only the submachine that describes the communication with the Garage Service.

Orchestator (ORCH)

The Orchestrator is a component that is in charge to achieve a goal by composition of service. Each time that a request arrive from a driver it performs a dynamic binding with internal and external components such as Road Assistance, Bank , GPS, etc.. These components are accessible through the Vehicle Communication Gateway. The behaviour of this component is represented by a composite state machine. In Figure 40 we present only the part related to the Card Charge Action. The complete UMC code is listed in Appendix B.

4.3.1 RPS Validation with UMC

In this section we show how to validate the RPS system using UMC. We have used UMC v3.5 to verify a set of behavioral properties specified using UCTL. In the following we firstly describe each of them both in an informal an formal way, after that we show the validation results.

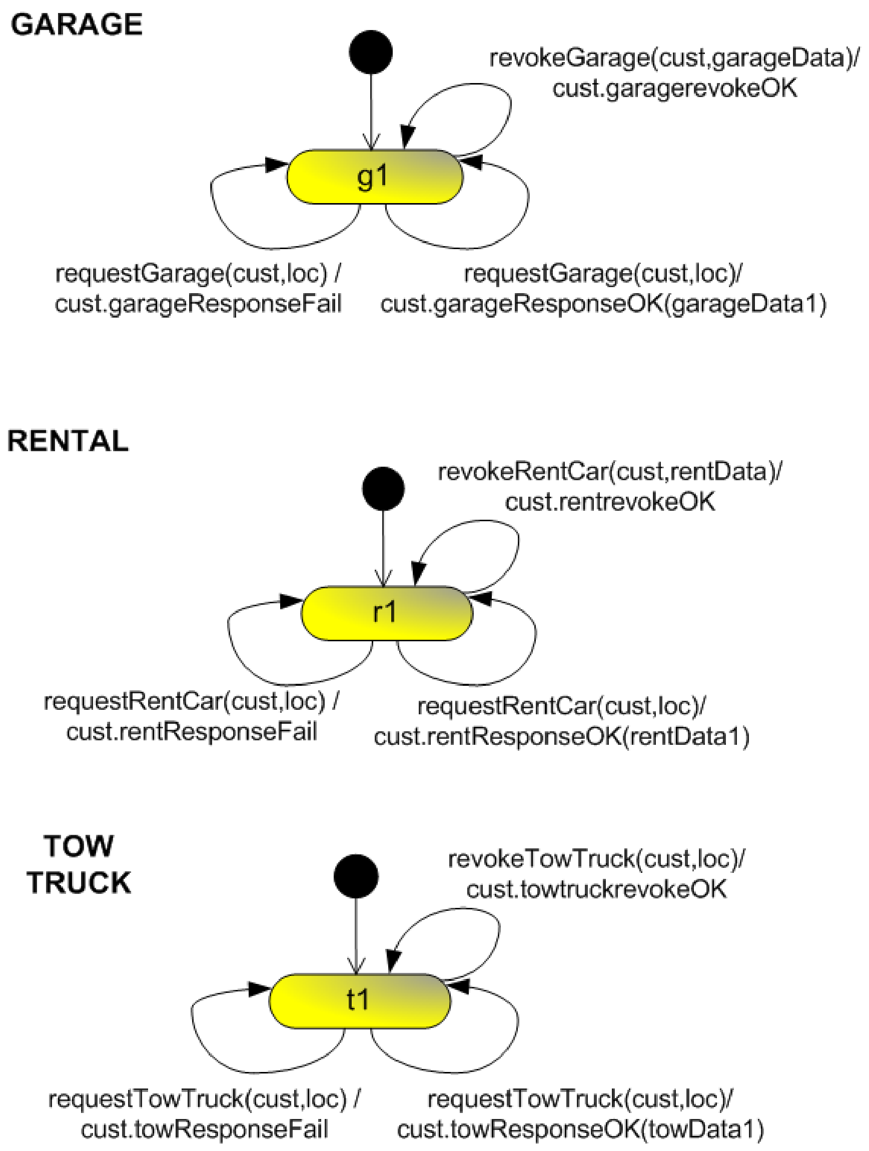

GARAGE
g1
revokeGarage(cust,garageData)/ cust.garagerevokeOK
requestGarage(cust,loc) / cust.garageResponseFail
requestGarage(cust,loc)/ cust.garageResponseOK(garageData1)
RENTAL
r1
revokeRentCar(cust,rentData)/ cust.rentrevokeOK
requestRentCar(cust,loc) / cust.rentResponseFail
requestRentCar(cust,loc)/ cust.rentResponseOK(rentData1)
TOW TRUCK
t1
revokeTowTruck(cust,loc)/ cust.towtruckrevokeOK
requestTowTruck(cust,loc) / cust.towResponseFail
requestTowTruck(cust,loc)/ cust.towResponseOK(towData1)

Figure 38: Road Assistance State Machine

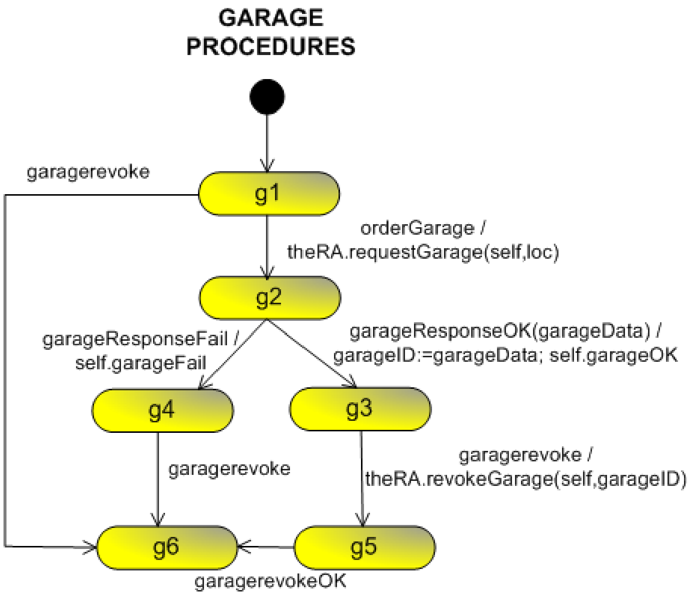

GARAGE PROCEDURES
garagerevoke
g1
orderGarage / theRA.requestGarage(self,loc)
g2
garageResponseFail / self.garageFail
garageResponseOK(garageData) / garageID:=garageData; self.garageOK
g4
g3
garagerevoke
garagerevoke / theRA.revokeGarage(self,garageID)
g6
g5
garagerevokeOK

Figure 39: Vehicle Communication Gateway State Machine

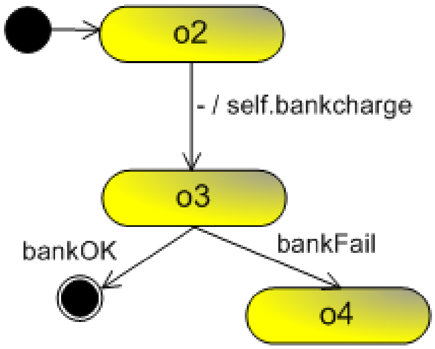


Figure 40: Orchestrator State Machine

`Bank` Responsiveness A Banks is *responsive* if it guarantees a response to each received request. We want to prove that each time action `requestCardCharge` takes place, eventually in the future the action `chargeResponseOK` or `chargeResponseFail` takes place. In detail, if a `Driver` requests to recharge a credit card to a certain `Banks`, then the `Bank` will reply with a notification of either a successful or a failed recharge.

```
 AG [car1:requestCardCharge] AF
{car1.chargeResponseOK or car1.chargeResponseFail}
true
```

We verified this formula with UMC and it is true.

`serviceRequest` Coordination It may never happen that action `chargeResponseOK` takes place if action `requestCardCharge` has not taken place before. The Car cannot receive a notification of the fact that the credit card has been charged, if it did not previously request the Bank to do so.

```
A [true{not chargeResponseOK}W{requesCardCharge}true].
```

We verified this formula with UMC and it is true.

`requestGarage` Reliability Each time action `requestGarage` takes place, eventually in the future `garageResponseOK` takes place. It means that garage requests from `Car` to `RASS` are always followed by a notification of success.

```
AG[car1:requestGarage]AF{car1.garageResponseOK}true
```

We verified this formula with UMC and it is false. This is not surprising: each `Garage` might be temporarily unable to provide the request (they send `garageResponseFail`).

`serviceRequest` Uniqueness Each time action `requestGarage` takes place, then `garageResponseFail` or `garageResponseOK` takes place and afterward it may not happen that eventually one of the latter two actions takes place again. It means that after a garage request from `Car` to `RASS`, it cannot happen that the `Car` receives more than one notification.

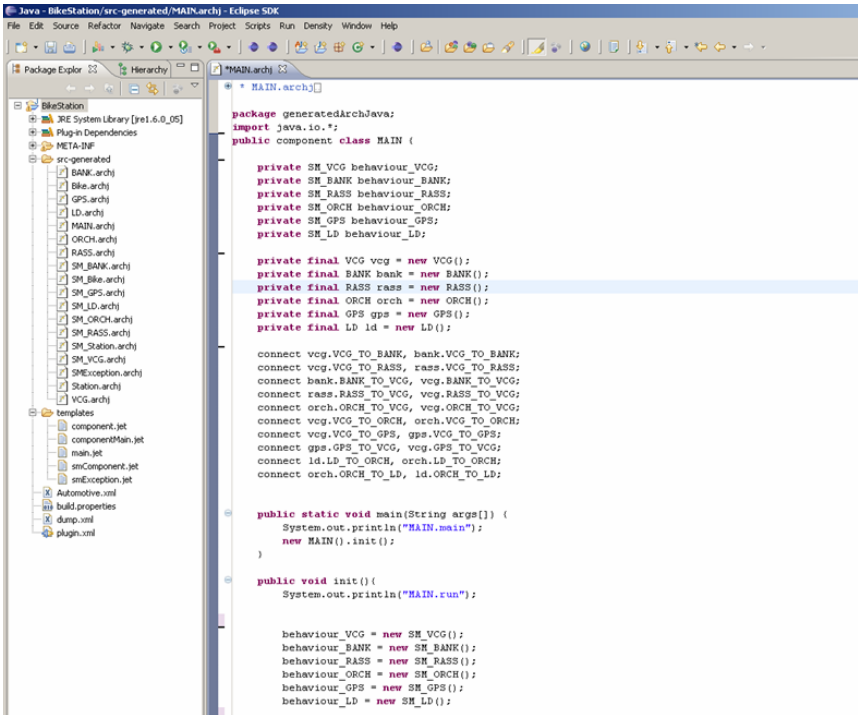
```
AG[car1:requestGarage]
  not EF<car1.garageResponseOK or car1.garageResponseFail>
  EF<car1.garageResponseOK or car1.garageResponseOK>true
```

We verified this formula with UMC and it is true.

4.4 RPS Code Generation

The Application of the *JET-based Code Generator* (outlined in Section 3.4) on the UMC specification of the RPS system produces a number of ARCHJAVA files listed on the left-hand side of the screenshot in Figure 41. In particular, for each component (i.e., BANK, GPS, ORCH, VCG, RASS, LD etc.), the corresponding encoding is generated (e.g., *BANK.archj*, *GPS.archj*, etc.). The state machine specifications are also synthesized (i.e., *SM_BANK.archj*, *SM_GPS.archj*, etc.) together with a MAIN.archj file (listed on the right-hand side of Figure 41) that enables the execution of the obtained system with respect to the modelled software architecture. Being more precise, in that main file all the components, the corresponding state machines and port connections are instantiated giving place to an encoding of the SA properties that constraint the execution of the hand-written code that will be filled in prearranged points (e.g., see the *try* statement in the code of Figure 25.

In the following we list only the ARCHJAVA code of `BANK` component with its state machine specification generated using the JET-based Code Generator. The complete code of the RPS system is listed in Appendix C.


Java - BikeStation/src-generated/MAIN.archj - Eclipse SDK
File Edit Source Refactor Navigate Search Project Scripts Run Density Window Help
Package Explor
Hierarchy
*MAIN.archj

```
BikeStation
  JRE System Library [jre1.6.0_05]
  Plug-in Dependencies
  META-INF
  src-generated
    BANK.archj
    Bike.archj
    GPS.archj
    LD.archj
    MAIN.archj
    ORCH.archj
    RASS.archj
    SM_BANK.archj
    SM_Bike.archj
    SM_GPS.archj
    SM_LD.archj
    SM_ORCH.archj
    SM_RASS.archj
    SM_Station.archj
    SM_VCG.archj
    SMException.archj
    Station.archj
    VCG.archj
  templates
    component.jet
    componentMain.jet
    main.jet
    smComponent.jet
    smException.jet
  Automotive.xml
  build.properties
  dump.xml
  plugin.xml
```

```
* MAIN.archj

package generatedArchJava;
import java.io.*;
public component class MAIN {

    private SM_VCG behaviour_VCG;
    private SM_BANK behaviour_BANK;
    private SM_RASS behaviour_RASS;
    private SM_ORCH behaviour_ORCH;
    private SM_GPS behaviour_GPS;
    private SM_LD behaviour_LD;

    private final VCG vcg = new VCG();
    private final BANK bank = new BANK();
    private final RASS rass = new RASS();
    private final ORCH orch = new ORCH();
    private final GPS gps = new GPS();
    private final LD ld = new LD();

    connect vcg.VCG_TO_BANK, bank.VCG_TO_BANK;
    connect vcg.VCG_TO_RASS, rass.VCG_TO_RASS;
    connect bank.BANK_TO_VCG, vcg.BANK_TO_VCG;
    connect rass.RASS_TO_VCG, vcg.RASS_TO_VCG;
    connect orch.ORCH_TO_VCG, vcg.ORCH_TO_VCG;
    connect vcg.VCG_TO_ORCH, orch.VCG_TO_ORCH;
    connect vcg.VCG_TO_GPS, gps.VCG_TO_GPS;
    connect gps.GPS_TO_VCG, vcg.GPS_TO_VCG;
    connect ld.LD_TO_ORCH, orch.LD_TO_ORCH;
    connect orch.ORCH_TO_LD, ld.ORCH_TO_LD;


    public static void main(String args[]) {
        System.out.println("MAIN.main");
        new MAIN().init();
    }

    public void init(){
        System.out.println("MAIN.run");


        behaviour_VCG = new SM_VCG();
        behaviour_BANK = new SM_BANK();
        behaviour_RASS = new SM_RASS();
        behaviour_ORCH = new SM_ORCH();
        behaviour_GPS = new SM_GPS();
        behaviour_LD = new SM_LD();
```

Figure 41: Generated Code Overview of the RPS.

```
/** BANK.archj*/

package generatedArchJava;
import java.io.*;

public component class BANK {
/*Declaration of the state machine variables*/
private SM_VCG behaviour_VCG;
private SM_BANK behaviour_BANK;
private SM_RASS behaviour_RASS;
private SM_ORCH behaviour_ORCH;
private SM_GPS behaviour_GPS;
private SM_LD behaviour_LD;

/** setBehaviours() */
public void setBehaviours(SM_VCG behaviour_VCG, SM_BANK behaviour_BANK,
     SM_RASS behaviour_RASS, SM_ORCH behaviour_ORCH, SM_GPS behaviour_GPS,
     SM_LD behaviour_LD ){
     System.out.println("BANK.setBehaviours");
     this.behaviour_VCG = behaviour_VCG;
     this.behaviour_BANK = behaviour_BANK;
     this.behaviour_RASS = behaviour_RASS;
     this.behaviour_ORCH = behaviour_ORCH;
     this.behaviour_GPS = behaviour_GPS;
     this.behaviour_LD = behaviour_LD;
}
/**VCG_TO_BANK Port definition */
public port VCG_TO_BANK {
}
/**BANK_TO_VCG Port definition*/
public port BANK_TO_VCG {
}

/**
* Implementation of the methods
* provided by the port VCG_TO_BANK*/
 }
```

Listing 4.6: BANK.archj automatically generated code.

```
/**SM_BANK.archj*/
package generatedArchJava;
import java.util.LinkedList;
import generatedArchJava.SMException;

/** BANK State Machine encoding*/
public class SM_BANK {
public final int S_S1= 0;

/** Transition encoding*/
 public final int T_revokeCardCharge=0;
 public final int T_cust.bankrevokeOK=1;
 public final int T_requestCardCharge=2;
 public final int T_cust.chargeResponseOK=3;
 public final int T_cust.chargeResponseFail=4;
 private int currentState=S_S1;
 private LinkedList states = new LinkedList();
 ...
}

/** State Machine constructor*/
public SM_BANK(){
   System.out.println("SM_BANK.constr");
   LinkedList S1 = new LinkedList();
   S1.add(new transition(T_revokeCardCharge, S_S1 ,1));
   S1.add(new transition(T_cust.bankrevokeOK, S_S1 ,0));
   S1.add(new transition(T_requestCardCharge, S_S1 ,1));
   S1.add(new transition(T_cust.chargeResponseOK, S_S1 ,0));
   S1.add(new transition(T_cust.chargeResponseFail, S_S1 ,0));
   states.add(S1);
}
...
}
```

Listing 4.7: ArchJava Code of the BANK state machine.

Chapter 5

Conclusions and Future Work

The objective of this chapter is to summarize the main results that I have obtained in this research and possible future directions that I can consider for my research work. I present the main results and open issues subdivided in two categories. The first regards the use of formal methods to specify and validate Dynamic Software Architectures, the second one regards the use of SA-based development process to generate and maintain code of global computing systems.

5.1 Formal Design and Validation

The approach proposed in this thesis to design dynamic software architectures uses typed graph grammars as a formal base and the Alloy logic to implement concepts like architectural styles, graph transformation rules and architectural structural properties. Moreover the *Alloy Analyzer* is used to ensure style-consistency, perform model-finding and validate structural properties of each SA configuration. Typed Graph Grammar (TGG) uses explicit structural constraints by means of logical predicates. This approach is well suited to a reactive modeling process: the software architect builds a model and the system reacts reporting style inconsistencies. TGG defines dynamism by local rewriting rules on flat graphs and it has mechanisms to keep trace of reconfigured items by the notion of trace morphism. In order to verify structural properties TGG express them by means of the same formalism used to define architectural styles, i.e. the Alloy logic.

Model Finding is the main analysis capability offered by Alloy. The Alloy Analyzer basically explores (a bounded fragment) of the state space of all possible models and is able to show example instances satisfying or violating the desired properties. For instance, we can easily use the Alloy Analyzer to construct initial architectures: we need to ask for an instance graph satisfying the style characteristics and having a certain number of components. Model finding can also serve to the purpose of analysis. For instance, to validate if the style predicates really define what the software architect means. The use of bounds is justified by Alloy's small scope hypothesis that states that *"most bugs have small counterexamples"* (Jac06). This means that examining small architectures is often enough to detect possible major flaws. Table 12 can help readers to understand which elements of a Dynamic Software Architecture we have considered and in which way we have approached it.

Aspects	**Approach Used**
Architectures	Flat Typed Graphs
Structural properties	Alloy Logic
Behavioural properties	UCTL Formulae
Style Matching	Alloy Analyzer
Style Preservation	Alloy Analyzer
Model Finding	Alloy Analyzer
Invariant Analysis	Alloy Analyzer
Reachability Analysis	Alloy Analyzer
Model Checking	UMC + UCTL
Code Generation	ARCHJAVA + JET
Communication Integrity	ARCHJAVA

Table 12: Main aspects of the Thesis.

5.1.1 Open Issues

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 (HHT96) 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. Another future research is to extend our modeling approach to model and analyze Hierarchical SAs that have as basic elements more complex and structured components. We are thinking to use Hierarchical Hypergraphs (DHP02) where each hyperedge can represent relations among components.

5.2 SA-based Development

SA-based development is based on the idea that code can be generated from the SA specificaton of a system. This thesis has provided a contribution in this direction, by showing how an architectural model defined in a model-based fashion can be used for code generation. An important aspect during this process consists in ensuring that the selected architecture provides the required qualities. We have shown how this is feasible in a specific context, where coordination properties are modelled and verified against each architectural model. As soon as the architectural model is proved to be good enough, we demonstrated that it can be used for generating code, constraining the system execution according to the architectural decisions.

5.2.1 Open Issues

In this thesis we presented an approach to automatically generate code starting from verified software architecture descriptions. The best of the state of the art, as presented in Section 2.4, is represented by ARCHJAVA that ensures the communication integrity, and by JAVA/A that constraints also the code to behave as defined in the port's protocols. These approaches (as they are today) can be used only in a context in which the system is completely implemented in-house, while neglecting the possibility of integrating external components. This is because acquired components are not necessarily implemented following one of these approaches and thus it is not possible at runtime allow the only admitted operations. Thus, an interesting future research direction consists of the ability of integrating in-house components code with automatically generated assembly code for acquired components, forcing the composed system to exhibit the properties specified at the architectural level. This integration would open the possibility to really manage dynamic system during the execution (i.e., systems in which some components have to change at runtime) where a regenerated "correct" assembly code assures that the composed system is forced to exhibit only the properties specified at the architectural level. To realize these aspects we should develop a middleware able to support dynamic software architecture adaptations. Figure 42 shows a possible realization of it already presented by Geihs et al. in (GKRS06) and used in the IST MADAM Project [1]. The Context Manager monitors and processes the context information (i.e., context sensors). The Adaptation Manager chooses an adaptation activity based on context information received. Finally the Configuration Manager starts the appropriate configuration choosing the precise set of components (stored in a precise repository) that will constitute the new SA configuration.

Additionally, feedback generated by the runtime analysis of the generated code (provided by the Context Manager through monitoring and testing techniques) could be automatically tracked back to the architectural model, so that whenever a change applies over the code, it is auto-

[1]http://www.intermedia.uio.no/display/madam/Home

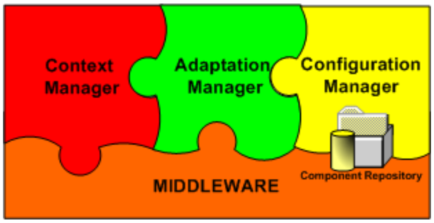


Figure 42: Middleware for Dynamic Software Architectures.

matically reflected on the architectural modele and vice-versa. Another future research direction consists in investigating how to assure that the generated code respects non functional properties and quality aspects which have been proved to be valid at the architectural level. Finally we would like to introduce aspects like reconfiguration rules in the Java Code, as modelled by the graph grammar production rules. This is important to have a strong relation between programmed dynamic reconfiguration and the code.

Appendix A

Alloy Code

This appendix presents the Alloy code introduced in this thesis. First of all I present the general Alloy Modules that must be defined each time that we model a new system. They are depicted in Figure 43. After I list the complete Alloy Code of the Running Example introduced in Chapter 3 and finally the Alloy Code of the Automotive Case Study described in the Chapter 4.

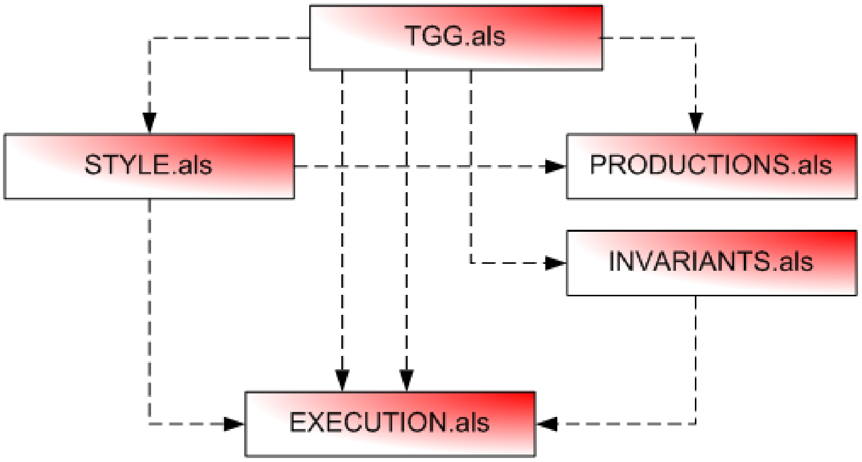


Figure 43: Alloy Modules

```
module TGG

//Graph Elements
sig Node{}
sig Tentacles{}
sig Edge
{
 tentacles: set Tentacles,
 conn: tentacles-> lone Node
}
sig Graph
{
 he: set Edge,
 n: set Node
}
// facts on Graphs and Graph elements (Nodes , Edges and Tentacles)
fact GraphElements_Constraints
{
// each element (Node and Edge) must be element of a single Graph
  all edge: Edge | some g: Graph | edge in g.he
  all node: Node | some g: Graph | node in g.n
  all t1: Tentacles | some g: Graph | some e1:g.he | t1 in e1.tentacles
// nodes at which each Edge is connected must be nodes of the same Graph
  all g:Graph| all e: g.he | univ.(e.conn) in g.n
}
```

Listing A.1: Typed HyperGraphs

```
// PartialMorphism among two Graphs
sig PartialMorphism
{
 source: Graph,
 target: Graph,
 fE: Edge -> lone Edge,
 fN : Node -> lone Node
}
{
 // mapping functions description
 fE.univ in source.he
 fN.univ in source.n
 univ.fE in target.he
 univ.fN in target.n

 // f maps a subgrapgh of the source graph
 all e1: fE.univ |univ.(e1.conn) in fN.univ
 // injectivity
 all n1,n2: source.n | fN[n1] = fN[n2] => n1=n2
 all e1,e2: source.he |fE[e1] = fE[e2] => e1=e2
 // well-formedness of the partialmorphism --
 all e1: fE.univ | e1.tentacles = fE[e1].tentacles
 //all e1: fE.univ | all t1: e1.tentacles | fE[e1].conn[t1] = fN[e1.conn[t1]]
}
```

Listing A.2: Partial Morphism

```
sig Matching
{
 source: Graph,
 target: Graph ,
 fE: Edge -> lone Edge,
 fN : Node -> lone Node
}
{
 // mapping functions description
 fE.univ = source.he
 fN.univ = source.n
 univ.fE in target.he
 univ.fN in target.n

 // f maps a subgrapgh of the source graph
 all e1: fE.univ |univ.(e1.conn) in fN.univ

 // injectivity
 all n1,n2: source.n | fN[n1] = fN[n2] => n1=n2
 all e1,e2: source.he |fE[e1] = fE[e2] => e1=e2

 // well-formedness of the partialmorphism
 all e1: fE.univ | e1.tentacles = fE[e1].tentacles
 all e1: fE.univ | all t1: e1.tentacles |
             fE[e1].conn[t1] = fN[e1.conn[t1]]
}
```

Listing A.3: Matching

```
// It describes Partial Morphism among two TypedGraphs
sig TypedPartialMorphism
{
 s: TypedGraph,
 t: TypedGraph,
 PMorphism: PartialMorphism
}
{
 PMorphism.source = s.typingmorphism.source
 PMorphism.target = t.typingmorphism.source
 all e1: PMorphism.fE.univ |
 s.typingmorphism.fE[e1] = t.typingmorphism.fE[PMorphism.fE[e1]]
 all n1: PMorphism.fN.univ |
 s.typingmorphism.fN[n1] = t.typingmorphism.fN[PMorphism.fN[n1]]
}
// Matching among two TypedGraphs
sig TypedMatching
{
 s: TypedGraph,
 t: TypedGraph,
 match: Matching
}
{
 match.source = s.typingmorphism.source
 match.target = t.typingmorphism.source
}
```

Listing A.4: Typed Partial Morphism and Typed Matching

```
sig TotalMorphism
{
 source: Graph,
 target: Graph,
 fE: Edge->Edge,
 fN: Node->Node
}
{
 univ.fE in target.he
 univ.fN in target.n
 fE.univ = source.he
 fN.univ = source.n
 //unicity
 all e1: source.he | one e2: target.he | fE[e1]=e2
 all n1: source.n | one n2: target.n | fN[n1] = n2
 // well-formedness
 all e1: source.he | e1.tentacles = fE[e1].tentacles
 all e1: source.he | all t1:e1.tentacles | (fE[e1]).conn[t1] = fN[e1.conn[t1]]
}
sig TypedTotalMorphism
{
 s: TypedGraph,
 t: TypedGraph,
 TMorphism: TotalMorphism
}
{
 TMorphism.source = s.typingmorphism.source
 TMorphism.target = t.typingmorphism.source
 all e1: TMorphism.fE.univ |
 s.typingmorphism.fE[e1] = t.typingmorphism.fE[TMorphism.fE[e1]]
 all n1: TMorphism.fN.univ |
 s.typingmorphism.fN[n1] = t.typingmorphism.fN[TMorphism.fN[n1]]
}
```

Listing A.5: Total Morphism and Typed Total Morphism

```
// TypeGraph Definition
// it is described in detail in the STYLE Module of each new model
one sig TypeGraph extends Graph{}

// To avoid reuse of identities of items of the type graph
fact NoClashWithTypeNames{
 all g1: Graph | g1!=TypeGraph => #(g1.n & TypeGraph.n) = 0
 all g1: Graph | g1!=TypeGraph => # (g1.he & TypeGraph.he) = 0
}
// TypedGraph Definition
sig TypedGraph
{
 typingmorphism : TotalMorphism,
}
{
 typingmorphism.target=TypeGraph
}
```

Listing A.6: Type Graph and Typed Graph

```
pred rwStep [G,G':TypedGraph, P: TypedPartialMorphism,
          trace: TypedPartialMorphism]
{
  one g: TypedMatching | {
  g.s=P.s and g.t = G and

  G'.typingmorphism.source.he =
  (G.typingmorphism.source.he)-
  g.match.fE[(P.s.typingmorphism.source.he)] +
  (P.t.typingmorphism.source.he)
  and
  G'.typingmorphism.source.n =
   (G.typingmorphism.source.n)-g.match.fN[(P.s.typingmorphism.source.n)] +
  (P.t.typingmorphism.source.n)
}
and
  trace.s = G and trace.t=G' and
  all n1: G'.typingmorphism.source.n |
    n1 in univ.(g.match.fN) &&
   (g.match.fN.n1) in (P.PMorphism.fN).univ
    =>
    n1 in (trace.PMorphism.fN).univ
}
```

Listing A.7: Rewriting Step

```
//--------------Description of Programmed DSA----------------
sig Transition {
  startingState: TypedGraph,
  arrivalState: TypedGraph,
  trigger : TypedPartialMorphism
}
{
  startingState != arrivalState
}
sig ProgrammedDSA
{
  configurations: set TypedGraph,
  transitions: set Transition,
  first, last: configurations,
  next: (configurations-last) one -> one (configurations-first)
}
{
   // first !=last
   // the first TypedGraph can not be an arrivalState
   first not in transitions.arrivalState
   // the last TypedGraph can not be a startingState
   last not in transitions.startingState
   next.univ in transitions.startingState
}
```

Listing A.8: Programmed DSA

```
fact GenerationTransition
{
   all pdsa: ProgrammedDSA|
   all  trans: pdsa.transitions |one tr: TypedPartialMorphism |
   rwStep[trans.startingState, trans.arrivalState, trans.trigger, tr] &&
   trans.arrivalState = trans.startingState.(univ.next)
}
```

Listing A.9: constraints on Programmed DSA

```
module RunningEXStyle
open TGG
open SOFTWARE
//-------------TYPEGRAPH DEFINITION--------------------
/* ---------------NOTATION---------------------------------
 AP = Access_Point
 CP = Chain_Point
-----------------------------------------------------------*/
//Bike-Style basic elements
one sig CP, AP extends Node {}
one sig Left, Right, Access extends Tentacles {}
one sig Station extends Edge{}
{
 tentacles = Left+Right+Access
 conn = Left->CP + Right->CP + Access->AP
}
one sig Bike extends Edge{}
{
 tentacles = Access
 conn = Access->AP
}
one sig BikeStation extends Edge{}
{
 tentacles = Left+Right
 conn = Left->CP + Right->CP
}
fact onTypeGraph
{
 TypeGraph.n = CP+AP
 TypeGraph.he = Station + Bike+BikeStation
}
```

Listing A.10: Style of the Running Example

```
one sig s1,s2,b1,b2,b3 extends Edge{}
one sig cp1,cp2,cp3,ap1,ap2 extends Node{}
one sig g1 extends Graph {}
{
 he =s1+b1
 n= cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 b1.conn = Access->ap1
}
one sig g2 extends Graph{}
{
 he = s1
 n = cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
}
one sig g3 extends Graph{}
{
 he = s1 + b2
 n = cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 b2.conn = Access->ap1
}
one sig g4 extends Graph{}
{
 he = s1+s2 + b1
 n = cp1+cp2+cp3+ap1+ap2
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 s2.conn = Left->cp2 + Right->cp3 + Access->ap2
 b1.conn = Access->ap1
}
one sig g5 extends Graph{}
{
 he = s1+s2 +b3
 n = cp1+cp2+cp3+ap1+ap2
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 s2.conn = Left->cp2 + Right->cp3 + Access->ap2
 b3.conn = Access->ap2
}
one sig g6 extends Graph{}
{
 he = s1+b1+b2
 n = cp1+cp2+ap1
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 b1.conn = Access->ap1
 b2.conn = Access->ap1
}
one sig g7 extends Graph{}
{
 he = s1+s2
 n = cp1 + cp2 + cp3 + ap1 + ap2
 s1.conn = Left->cp1 + Right->cp2 + Access->ap1
 s2.conn = Left->cp2 + Right->cp3 +Access->ap2
}
```

Listing A.11: Graphs of the Productions

```
one sig G1 extends TypedGraph{}
{
 typingmorphism.source = g1
 typingmorphism.fE = s1->Station + b1->Bike
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
one sig G2 extends TypedGraph{}
{
 typingmorphism.source = g2
 typingmorphism.fE = s1->Station
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
one sig G3 extends TypedGraph{}
{
 typingmorphism.source = g3
 typingmorphism.fE = s1->Station + b2->Bike
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
one sig G4 extends TypedGraph{}
{
 typingmorphism.source = g4
 typingmorphism.fE = s1->Station + b1->Bike + s2->Station
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP + cp3->CP + ap2->AP
}
one sig G5 extends TypedGraph{}
{
 typingmorphism.source = g5
 typingmorphism.fE = s1->Station + b3->Bike + s2->Station
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP + cp3->CP + ap2->AP
}
one sig G6 extends TypedGraph{}
{
 typingmorphism.source = g6
 typingmorphism.fE = s1->Station + b1->Bike + b2->Bike
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP
}
one sig G7 extends TypedGraph{}
{
 typingmorphism.source = g7
 typingmorphism.fE = s1->Station + s2->Station
 typingmorphism.fN = cp1->CP + cp2->CP + ap1->AP +ap2->AP + cp3->CP
}
```

Listing A.12: Typed Graphs of the Productions

```
//----------------------------LEAVE1------------------------------
one sig p1 extends PartialMorphism{}
one sig LEAVE1 extends TypedPartialMorphism{}
{
      s = G1
      t= G2
      PMorphism = p1
      PMorphism.source = g1
      PMorphism.target = g2
      PMorphism.fE = s1->s1
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1
}
//----------------------------LEAVE2------------------------
one sig p2 extends PartialMorphism{}
one sig LEAVE2 extends TypedPartialMorphism{}
{
      s = G3
      t= G2
      PMorphism = p2
      PMorphism.source = g3
      PMorphism.target = g2
      PMorphism.fE = s1->s1
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1
}
//----------------------------JOIN1--------------------------
one sig p3 extends PartialMorphism{}
one sig JOIN1 extends TypedPartialMorphism{}
{
      s = G2
      t= G1
      PMorphism = p3
      PMorphism.source = g2
      PMorphism.target = g1
      PMorphism.fE = s1->s1
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1
}
//----------------------------MIGRATION----------------------
one sig p5 extends PartialMorphism{}
one sig MIGRATION extends TypedPartialMorphism{}
{
      s = G4
      t= G5
      PMorphism = p5
      PMorphism.source = g4
      PMorphism.target = g5
      PMorphism.fE = s1->s1 + s2->s2 + b1->b3
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1+ ap2->ap2+
  cp3->cp3

}
```

Listing A.13: Running Example Productions

```
//----------------------------STATION CONNECTION--------------
one sig p6 extends PartialMorphism{}
one sig STATIONConn extends TypedPartialMorphism{}
{
      s = G2
      t= G7
      PMorphism = p6
      PMorphism.source = g2
      PMorphism.target = g7
      PMorphism.fE = s1->s1
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1
}
//----------------------------LEAVE3------------------
one sig p7 extends PartialMorphism{}
one sig LEAVE3 extends TypedPartialMorphism{}
{
      s = G5
      t= G7
      PMorphism = p7
      PMorphism.source = g5
      PMorphism.target = g7
      PMorphism.fE = s1->s1 + s2->s2
      PMorphism.fN = cp1->cp1 + cp2->cp2 + ap1->ap1 + cp3->cp3
  + ap2->ap2
}
```

Listing A.14: Station Connection and Leave3 Productions

```
module RPSStyle
open TGG
open SOFTWARE
//-------------TYPEGRAPH DEFINITION--------------------
one sig EA, IA extends Node {}
one sig BV, RAV, VS,VV, VO, VGPS, OLD, OV, GPSV, LDO extends Tentacles {}
one sig BANK extends Edge{}
{
      tentacles = BV
      conn = BV->EA
}
one sig RASS extends Edge{}
{
      tentacles = RAV
      conn = RAV->EA
}
one sig VCG extends Edge{}
{
      tentacles = VS + VV + VO + VGPS
      conn = VS ->EA + VV->EA + VO->IA + VGPS->IA
}
one sig GPS extends Edge{}
{
      tentacles = GPSV
      conn = GPSV->IA
}
one sig LD extends Edge{}
{
      tentacles = LDO
      conn = LDO->IA
}
one sig ORCH extends Edge{}
{
      tentacles = OLD+OV
      conn = OLD->IA + OV->IA
}
one sig EV extends Edge {}
{
      tentacles = VV
      conn = VV-> EA
}
fact onTypeGraph
{
      TypeGraph.n = EA+IA
      TypeGraph.he = BANK + RASS + VCG + GPS + LD + ORCH + EV
}
```

Listing A.15: RPS Style

```
//----------Constraints--------------------------------------------------
fact onTypedGraph
{
all tg: TypedGraph |
     one morph: tg.typingmorphism|
       one g1: morph.source |
       all e1,e2: g1.he |e1!=e2 and morph.fE[e1] =VCG
   and morph.fE[e2]= VCG
                                =>#(e1.conn&e2.conn)=0

all tg: TypedGraph |
       one morph: tg.typingmorphism |
       one g1: morph.source |
       all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1
  and morph.fE[e1] =  BANK
       and morph.fE[e2] = VCG =>
               VS in univ.conn.(univ.(e1.conn))


all tg: TypedGraph |
       one morph: tg.typingmorphism |
       one g1: morph.source |
       all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1
    and morph.fE[e1] =  GPS
       and morph.fE[e2] = VCG =>
               VGPS in univ.conn.(univ.(e1.conn))


all tg: TypedGraph |
       one morph: tg.typingmorphism |
       one g1: morph.source |
       all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1
  and morph.fE[e1] =  ORCH
       and morph.fE[e2] = VCG =>
               OV in univ.conn.(univ.(e1.conn))


all tg: TypedGraph |
       one morph: tg.typingmorphism |
       one g1: morph.source |
       all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1
   and morph.fE[e1] =  RASS
       and morph.fE[e2] = VCG =>
               VS in univ.conn.(univ.(e1.conn))
```

Listing A.16: RPS Style Constraints - I

```
all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all e1,e2: g1.he | #(univ.(e1.conn) & univ.(e2.conn)) =1
  and morph.fE[e1] =  LD
      and morph.fE[e2] = ORCH =>
            OLD in univ.conn.(univ.(e1.conn))

 // ORCH and GPS can not be connected
all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all e1,e2: g1.he | morph.fE[e1] =  GPS
  and morph.fE[e2] = ORCH =>
            #(univ.(e1.conn)&univ.(e2.conn))=0

//LD and GPS can not be connected
all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all e1,e2: g1.he | morph.fE[e1] =  LD
 and morph.fE[e2] = GPS =>
            #(univ.(e1.conn)&univ.(e2.conn))=0

all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all n: g1.n | morph.fN[n] = EA => #(conn.n) <=2

all tg: TypedGraph |
      one morph: tg.typingmorphism |
      one g1: morph.source |
      all n: g1.n | morph.fN[n] = IA => #(conn.n) <=2
}
```

Listing A.17: RPS Style Constraints - II

Appendix B

UMC Code

```
Class Bike is
-- public interface
Signals:
stationFailure(theStation: Station)
connectionBikeOK(theStation: Station),
serviceOK

-- private part
Vars:
startStation: Station,
actStation: Station,

State Top= START, WAIT, STOP, WAIT4HELP

Transitions:
START -> WAIT
{- / actStation:=startStation; startStation.connectBike(self)}
WAIT -> WAIT4HELP
{connectionBikeOK(theStation) /actStation:=theStation;
actStation.serviceRequest(self)}
WAIT -> WAIT
{stationFailure(theStation) [theStation/=null ] /
actStation:= theStation; theStation.connectBike(self)}
WAIT ->STOP
{stationFailure(theStation)[theStation=null] / OUT.BIKESTOPPED}
WAIT4HELP -> STOP
{serviceOK / OUT.BIKEDISCONNECTED}
WAIT4HELP -> WAIT
{stationFailure(theStation)[theStation/=null] /
actStation:=theStation; theStation.connectBike(self)}
WAIT4HELP -> STOP
{stationFailure(theStation) [theStation=null] / OUT.PHONECALL}

end Bike;
```

Listing B.1: UMC Model of Class "Bike"

```
Class Station is

Signals:
connectBike(theBike: Bike)
stopBike (theBike: Bike)
serviceRequest (theBike: Bike)

Vars:

myBike: Bike,
nextStation: Station;

State Top = EMPTY, READY, STOP

Transitions:
EMPTY -> EMPTY
{connectBike(theBike) / theBike.stationFailure(nextStation)}
EMPTY -> READY
{connectBike(theBike) / myBike:= theBike;
myBike.connectionBikeOK(self)}
READY -> STOP
{serviceRequest(theBike) / myBike:=theBike; myBike.serviceOK}
READY -> EMPTY
{serviceRequest(theBike) / myBike:= theBike;
myBike.stationFailure(nextStation)}

end Station;

Objects:
// static object instantiation
bike1: Bike (startStation -> station1, actStation -> station1)
station1: Station (nextStation-> station2)
station2: Station (nextStation->station3)
station3: Station
```

Listing B.2: UMC Model of the Class ”Station”

```
Class Bank is
 Signals:
    requestCardCharge(cust:Car, cc:Token, amount:Token);
    -- replies: cust.chargeResponseOK(chargeID)
    --             cust.chargeResponseFail
    --
    revokeCardCharge(cust:Car, chargeID:Token);
    -- replies: revokeOK
State Top = s1

Transitions:
  s1 -> s1 { requestCardCharge(cust,cc,amount) /
  cust.chargeResponseOK(bankopID) }
  s1 -> s1 { requestCardCharge(cust,cc,amount) /
  cust.chargeResponseFail }
  s1 -> s1 { revokeCardCharge(cust,chargeID) /
  cust.revokeOK }
end Bank
```

Listing B.3: UMC - RPS System: BANK Component

```
Class RoadAssistance is
  Signals:
     ------- GARAGE SERVICES -------
    requestGarage(cust:Car,loc:Token);
     -- replies: garageResponseOK(garageData) to car
     --         garageResponseFail    to car
     --
     revokeGarage(cust:Car,garageData:Token);
     -- replies: revokeOK

    -------- TOWTRUCK SERVICES -------
    requestTowTruck(cust:Car,loc:Token);
     -- replies: towResponseOK(towData) to car
     --         towResponseFail     to car
     --
     revokeTowTruck(cust:Car, towData:Token)
     -- replies: cust.revokeOK

    ------- RENTAL SERVICES -------
    requestRentCar(cust:Car,loc:Token);
     -- replies: rentResponseOK(rentData) to car
     --         rentResponseFail    to car
     --
     revokeRentCar(cust:Car, rentData:Token)
     -- replies: cust.revokeOK
     --

  State Top = Services
  State Services = GarageService / TowTruckService / RentalCarService
  State GarageService = g1
  State TowTruckService = t1
  State RentalCarService = r1

 Transitions:

   -- garage services
   g1 -> g1 { requestGarage(cust,loc)
   / cust.garageResponseOK(garageData1) }
   g1 -> g1 { requestGarage(cust,loc)
   / cust.garageResponseFail }
   g1 -> g1 { revokeGarage(cust,garageData)
   / cust.revokeOK }

   -- tow truck
   t1 -> t1 { requestTowTruck(cust,loc)
   / cust.towResponseOK(towData1) }
   t1 -> t1 { requestTowTruck(cust,loc)
   / cust.towResponseFail }
   t1 -> t1 { revokeTowTruck(cust,towData)
   / cust.revokeOK }

   -- rental
   r1 -> r1 { requestRentCar(cust,loc)
   / cust.rentResponseOK(rentData1) }
   r1 -> r1 { requestRentCar(cust,loc)
   / cust.rentResponseFail }
   r1 -> r1 { revokeRentCar(cust,rentData)
   / cust.revokeOK }

end RoadAssistance
```

Listing B.4: UMC - RPS System: RASS Component

Appendix C

ArchJava and Java Code

```
* Bike.archj
 * @generated
 */

package generatedArchJava;
import java.io.*;

public component class Bike {

      /**
      * Declaration of the state machine variables
      *@generated
      */
      private SM_Bike behaviour_Bike;
      private SM_Station behaviour_Station;

      /**
      * setBehaviours()
      *@generated
      */
      public void setBehaviours(SM_Bike behaviour_Bike,
  SM_Station behaviour_Station ){
            System.out.println("Bike.setBehaviours");
            this.behaviour_Bike = behaviour_Bike;
            this.behaviour_Station = behaviour_Station;
      }


      /**
      * Bike_TO_Station Port definition
      *@generated
      */
      public port Bike_TO_Station {
      }

      /**
      * Implementation of the methods
      * provided by the port Station_TO_Bike
      *@generated
      */
      /**
      * Station_TO_Bike Port definition
      *@generated
      */
      public port Station_TO_Bike {
      }

 }
```

Listing C.1: Bike.archj

```
* Station.archj
 * @generated
 */

package generatedArchJava;
import java.io.*;

public component class Station {

      /**
      * Declaration of the state machine variables
      *@generated
      */
      private SM_Bike behaviour_Bike;
      private SM_Station behaviour_Station;

      /**
      * setBehaviours()
      *@generated
      */
      public void setBehaviours(SM_Bike behaviour_Bike,
       SM_Station behaviour_Station ){
            System.out.println("Station.setBehaviours");
            this.behaviour_Bike = behaviour_Bike;
            this.behaviour_Station = behaviour_Station;
      }


      /**
      * Bike_TO_Station Port definition
      *@generated
      */
      public port Bike_TO_Station {
      }
      /**
      * Station_TO_Bike Port definition
      *@generated
      */
      public port Station_TO_Bike {
      }

      /**
      * Implementation of the methods
      * provided by the port Bike_TO_Station
      *@generated
      */
 }
```

Listing C.2: Station.archj

```

* SM_Bike.archj
 * @generated
 */

package generatedArchJava;

import java.util.LinkedList;
import generatedArchJava.SMException;

/** Bike State Machine encoding
 * @generated
 */
public class SM_Bike {
      /** State encoding
      * @generated
      */
      public final int S_start= 0;
      public final int S_WAIT= 1;
      public final int S_WAIT4REQUEST= 2;
      public final int S_STOP= 3;

      /** Transition encoding
      * @generated
      */
      public final int T_startStation.connectBike=0;
      public final int T_stationFailiure=1;
      public final int T_theStation.connectBike=2;
      public final int T_connectionBikeOK=3;
      public final int T_stationFailure=4;
      public final int T_actStation.serviceRequest=5;
      public final int T_serviceOK=6;
      public final int T_OUT.BIKEDISCONNECTED=7;
      public final int T_OUT.BIKESTOPPED=8;
      public final int T_OUT.PHONECALL=9;


      private int currentState=S_start;

      private LinkedList states = new LinkedList();

      private class transition{
            private int state;
            private int transition;
            private int send_receive;

            public transition(int transition, int state, int send_receive){
                  this.transition=transition;
                  this.state=state;
                  this.send_receive=send_receive;
            }

            public int getTransition(){
                  return transition;
            }

            public int getState(){
                  return state;
            }

            public int getSendReceive(){
                  return send_receive;
            }
      }
```

Listing C.3: Bike Component State Machine - I

```
        /** State Machine constructor
        * @generated
        */
        public SM_Bike(){
              System.out.println("SM_Bike.constr");


              LinkedList start = new LinkedList();
              start.add(new transition(T_startStation.connectBike, S_WAIT ,0));
              states.add(start);

              LinkedList WAIT = new LinkedList();
              WAIT.add(new transition(T_stationFailiure, S_WAIT ,1));
              WAIT.add(new transition(T_theStation.connectBike, S_WAIT ,0));
              WAIT.add(new transition(T_connectionBikeOK, S_WAIT4REQUEST ,1));
              WAIT.add(new transition(T_theStation.connectBike, S_WAIT ,0));
              WAIT.add(new transition(T_stationFailure, S_STOP ,1));
              states.add(WAIT);

              LinkedList WAIT4REQUEST = new LinkedList();
              WAIT4REQUEST.add(new transition(T_stationFailure, S_WAIT ,0));
              WAIT4REQUEST.add(new transition(T_actStation.serviceRequest,
       S_WAIT4REQUEST ,0));
              WAIT4REQUEST.add(new transition(T_serviceOK, S_STOP ,1));
              WAIT4REQUEST.add(new transition(T_stationFailure, S_STOP ,1));
              states.add(WAIT4REQUEST);

              LinkedList STOP = new LinkedList();
              STOP.add(new transition(T_OUT.BIKEDISCONNECTED, S_STOP ,0));
              STOP.add(new transition(T_OUT.BIKESTOPPED, S_STOP ,0));
              STOP.add(new transition(T_OUT.PHONECALL, S_STOP ,0));
              states.add(STOP);

        }

        public void transFire(int trans) throws SMException {
          ...
        }
}
```

Listing C.4: Bike Component State Machine - II

```
* SM_Station.archj
 * @generated
 */

package generatedArchJava;

import java.util.LinkedList;
import generatedArchJava.SMException;

/** Station State Machine encoding
 * @generated
 */
public class SM_Station {
      /** State encoding
      * @generated
      */
      public final int S_EMPTY= 0;
      public final int S_READY= 1;
      public final int S_STOP= 2;

      /** Transition encoding
      * @generated
      */
      public final int T_connectBike=0;
      public final int T_theBike.stationFailure=1;
      public final int T_myBike.connectBikeOK=2;
      public final int T_serviceRequest=3;
      public final int T_myBike.serviceOK=4;
      public final int T_myBike.stationFailure=5;


      private int currentState=S_EMPTY;

      private LinkedList states = new LinkedList();

      private class transition{
            private int state;
            private int transition;
            private int send_receive;

            public transition(int transition, int state, int send_receive){
                  this.transition=transition;
                  this.state=state;
                  this.send_receive=send_receive;
            }

            public int getTransition(){
                  return transition;
            }

            public int getState(){
                  return state;
            }

            public int getSendReceive(){
                  return send_receive;
            }
      }
```

Listing C.5: Station Component State Machine - I

```
        /** State Machine constructor
        * @generated
        */
        public SM_Station(){
              System.out.println("SM_Station.constr");


              LinkedList EMPTY = new LinkedList();
              EMPTY.add(new transition(T_connectBike, S_EMPTY ,1));
              EMPTY.add(new transition(T_theBike.stationFailure, S_EMPTY ,0));
              EMPTY.add(new transition(T_connectBike, S_READY ,1));
              EMPTY.add(new transition(T_myBike.stationFailure, S_EMPTY ,0));
              states.add(EMPTY);

              LinkedList READY = new LinkedList();
              READY.add(new transition(T_myBike.connectBikeOK, S_READY ,0));
              READY.add(new transition(T_serviceRequest, S_STOP ,1));
              READY.add(new transition(T_serviceRequest, S_EMPTY ,1));
              states.add(READY);

              LinkedList STOP = new LinkedList();
              STOP.add(new transition(T_myBike.serviceOK, S_STOP ,0));
              states.add(STOP);

        }

        public void transFire(int trans) throws SMException {
          ...
        }
}
```

Listing C.6: Station Component State Machine - II

```
* BANK.archj
 * @generated
 */
package generatedArchJava;
import java.io.*;

public component class BANK {

      /**
      * Declaration of the state machine variables
      *@generated
      */
      private SM_VCG behaviour_VCG;
      private SM_BANK behaviour_BANK;
      private SM_RASS behaviour_RASS;
      private SM_ORCH behaviour_ORCH;
      private SM_GPS behaviour_GPS;
      private SM_LD behaviour_LD;

      /**
      * setBehaviours()
      *@generated
      */
      public void setBehaviours(SM_VCG behaviour_VCG,
  SM_BANK behaviour_BANK, SM_RASS behaviour_RASS, SM_ORCH behaviour_ORCH,
   SM_GPS behaviour_GPS, SM_LD behaviour_LD ){
            System.out.println("BANK.setBehaviours");
            this.behaviour_VCG = behaviour_VCG;
            this.behaviour_BANK = behaviour_BANK;
            this.behaviour_RASS = behaviour_RASS;
            this.behaviour_ORCH = behaviour_ORCH;
            this.behaviour_GPS = behaviour_GPS;
            this.behaviour_LD = behaviour_LD;
      }

      /**
      * VCG_TO_BANK Port definition
      *@generated
      */
      public port VCG_TO_BANK {
      }
      /**
      * BANK_TO_VCG Port definition
      *@generated
      */
      public port BANK_TO_VCG {
      }

      /**
      * Implementation of the methods
      * provided by the port VCG_TO_BANK
      *@generated
      */
```

Listing C.7: Bank.archj

```
* SM_BANK.archj
 * @generated
 */

package generatedArchJava;

import java.util.LinkedList;
import generatedArchJava.SMException;

/** BANK State Machine encoding
 * @generated
 */
public class SM_BANK {
      /** State encoding
      * @generated
      */
      public final int S_S1= 0;

      /** Transition encoding
      * @generated
      */
      public final int T_revokeCardCharge=0;
      public final int T_cust.bankrevokeOK=1;
      public final int T_requestCardCharge=2;
      public final int T_cust.chargeResponseOK=3;
      public final int T_cust.chargeResponseFail=4;


      private int currentState=S_S1;

      private LinkedList states = new LinkedList();

      private class transition{
            private int state;
            private int transition;
            private int send_receive;

            public transition(int transition, int state, int send_receive){
                  this.transition=transition;
                  this.state=state;
                  this.send_receive=send_receive;
            }

            public int getTransition(){
                  return transition;
            }

            public int getState(){
                  return state;
            }

            public int getSendReceive(){
                  return send_receive;
            }
      }
```

Listing C.8: Bank Component State Machine - I

```
      /** State Machine constructor
      * @generated
      */
      public SM_BANK(){
            System.out.println("SM_BANK.constr");


            LinkedList S1 = new LinkedList();
            S1.add(new transition(T_revokeCardCharge, S_S1 ,1));
            S1.add(new transition(T_cust.bankrevokeOK, S_S1 ,0));
            S1.add(new transition(T_requestCardCharge, S_S1 ,1));
            S1.add(new transition(T_cust.chargeResponseOK, S_S1 ,0));
            S1.add(new transition(T_cust.chargeResponseFail, S_S1 ,0));
            states.add(S1);

      }

public void transFire(int trans) throws SMException {
    ...
      }
}
```

Listing C.9: Bank Component State Machine - II

```
package generatedArchJava;

public class BANK extends archjava.runtime.Component
  implements archjava.runtime.HasPorts, archjava.runtime.IComponent {

  public BANK (archjava.runtime.Parent $parentArg$) {
    super ($parentArg$);

  }public BANK () {
    super ((archjava.runtime.Parent)null);

  }

  private generatedArchJava.SM_VCG behaviour_VCG;
  private generatedArchJava.SM_BANK behaviour_BANK;
  private generatedArchJava.SM_RAS behaviour_RAS;
  private generatedArchJava.SM_ORCH behaviour_ORCH;
  private generatedArchJava.SM_GPS behaviour_GPS;
  private generatedArchJava.SM_LD behaviour_LD;
  public void setBehaviours (
   generatedArchJava.SM_VCG behaviour_VCG,
   generatedArchJava.SM_BANK behaviour_BANK,
   generatedArchJava.SM_RAS behaviour_RAS,
   generatedArchJava.SM_ORCH behaviour_ORCH,
   generatedArchJava.SM_GPS behaviour_GPS,
   generatedArchJava.SM_LD behaviour_LD
  )
  {
    java.lang.System.out.println("BANK.setBehaviours");
    this.behaviour_VCG = behaviour_VCG;
    this.behaviour_BANK = behaviour_BANK;
    this.behaviour_RAS = behaviour_RAS;
    this.behaviour_ORCH = behaviour_ORCH;
    this.behaviour_GPS = behaviour_GPS;
    this.behaviour_LD = behaviour_LD;
  }

  public generatedArchJava.BANK$port$VCG_TO_BANK VCG_TO_BANK$port$;
  public generatedArchJava.BANK$port$BANK_TO_VCG BANK_TO_VCG$port$;
  public archjava.reflect.Port[] get$ports() {
    return new archjava.reflect.Port[] {
    new BANK$port$VCG_TO_BANK.Impl(this).port,
    new BANK$port$BANK_TO_VCG.Impl(this).port };
   }
  protected void $initSubs() {
  }
}
```

Listing C.10: Java Code of the Bank Component

```java
package generatedArchJava;

public class SM_BANK {

  public SM_BANK () {
    super ();

    java.lang.System.out.println("SM_BANK.constr");
    java.util.LinkedList S1 = new java.util.LinkedList();
    S1.add(this.new transition(this.T_revokeCardCharge, this.S_S1, 1));
    S1.add(this.new transition(this.T_bankrevokeOK, this.S_S1, 0));
    S1.add(this.new transition(this.T_requestCardCharge, this.S_S1, 1));
    S1.add(this.new transition(this.T_chargeResponseOK, this.S_S1, 0));
    S1.add(this.new transition(this.T_chargeResponseFail, this.S_S1, 0));
    this.states.add(S1);
  }

  public final int S_S1 = 0;
  public final int T_revokeCardCharge = 0;
  public final int T_bankrevokeOK = 1;
  public final int T_requestCardCharge = 2;
  public final int T_chargeResponseOK = 3;
  public final int T_chargeResponseFail = 4;
  private int currentState = this.S_S1;
  private java.util.LinkedList states = new java.util.LinkedList();
  public void transFire (int trans) throws generatedArchJava.SMException{
    .....
   }
  private class transition {
   public transition (int transition, int state, int send_receive) {
     super ();
     this.transition = transition;
     this.state = state;
     this.send_receive = send_receive;
    }

    private int state;
    private int transition;
    private int send_receive;
    public int getTransition () {
      return this.transition;
    }

    public int getState () {
      return this.state;
    }

    public int getSendReceive () {
      return this.send_receive;
    }

  }
}
```

Listing C.11: Java Code of the Bank Component State Machine

References

[AB06] D. Ayed and Y. Berbers. UML Profile for the Design of a Platform-Independent Context-Aware Applications. In *MODDM '06: Proceedings of the 1st workshop on MOdel Driven Development for Middleware (MODDM '06)*, pages 1–5, New York, NY, USA, 2006. ACM. 19, 22

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking for Real-Time Systems. In *LICS'90*, pages 414–425. IEEE, 1990. 26

[Acm] Acme Studio Home Page. `http://www.cs.cmu.edu/~acme`. 35

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE'02*, pages 187–197. ACM, 2002. 10, 29, 41

[ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *FASE'98*, volume 1382 of *LNCS*, pages 21–37, 1998. 4, 14, 57

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997. 25

[And00] J. Andersson. Issues in Dynamic Software Architectures. In *Proceedings of ISAW4*, pages 111–114, 2000. 4, 5, 51, 57

[Arc] ArchJava. `http://archjava.org`. 7

[Aut] AutoFOCUS Project. Public web site. `http://autofocus.in.tum.de/index-e.html`. 26

[B̈60] J. Büchi. On a Decision Problem in Restricted Second Order Arithmetic. In E. Nagel et al., editor, *Proceedings of the 2nd International Conference on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960. 27

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Middleware'98*, pages 15–18, 1998. 15

[BBGM08] R. Bruni, A. Bucchiarone, S. Gnesi, and H. Melgratti. Modelling Dynamic Software Architectures using Typed Graph Grammars. *ENTCS*, 213(1):39–53, 2008. 45

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003. 41, 96

[BCDW04] J.S. Bradbury, J.R. 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. 4, 5, 10, 55

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. SEI Series in Software Engineering. Addison-Wesley Professional, 2003. 2

[BCM05] P. Baldan, A. Corradini, and U. Montanari. Relating SPO and DPO Graph Rewriting with Petri Nets having read, inhibitor and reset arcs. *ENTCS*, 127(2):5–28, 2005. 40, 45

[BFGL07] M. Banci, A. Fantechi, S. Gnesi, and G. Lombardi. Model Driven Development and Code Generation: An Automotive Case Study. In *SDL Forum*, pages 19–34, 2007. 102

[BFM00] J. P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the Chemical Reaction Model: Fifteen Years After. In *WMP'00*, pages 17–44, 2000. 23

[BG08] A. Bucchiarone and J. P. Galeotti. Dynamic Software Architectures Verification using DynAlloy. In *GT-VMT'08*, ECEASST, 2008. 45

[BGH01] K. S. Barber, T. J. Graser, and J. Holt. Providing Early Feedback in the Development Cycle Through Automated Application of Model Checking to Software Architectures. In *ASE*, pages 341–345, 2001. 26

[BHH+06] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. A Component Model for Architectural Programming. *ENTCS*, 160:75–96, 2006. 10, 29, 31, 32

[BHTV06] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based Modeling and Refinement of Service-Oriented Architectures. volume 5, pages 187–207, 2006. 10, 21, 22, 23, 25

[BI03] M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods*. LNCS 2804, 2003. 3, 25

[BISZ98] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of ICCDS'98*, pages 35–42, 1998. 57

[BJC05] T. V. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *EWSA*, pages 1–17, 2005. 13

[BK07] D. Berndl and N. Koch. Automotive Scenario: Illustrating Service Specification. Technical report, FAST, 2007. 102

[BKK$^+$03] H. Baumeister, N. Koch, P. Kosiuczenko, P. Stevens, and M. Wirsing. UML for Global Computing. In *Global Computing*, pages 1–24, 2003. 20, 22

[BKP05] R. J. Bril, R. L. Krikhaar, and A. Postma. Architectural Support in Industry: a reflection using C-POSH. *Journal of Software Maintenance and Evolution*, 2005. 3

[BLL$^+$95] J. Bengtsson, K. Guldstrand Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*, pages 232–243, 1995. 26

[BLMT07] R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style based reconfigurations of software architectures. Technical Report TR-07-17, Dipartimento di Informatica, Università di Pisa, 2007. 44

[BM93] J. P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Commun. ACM*, 36(1):98–111, 1993. 23

[BM03] S. Balsamo and M. Marzolla. Towards performance evaluation of mobile systems in uml, 2003. 21, 22

[BMP06] A. Bucchiarone, H. Muccini, and P. Pelliccione. A Practical Architecture-Centric Analysis Process. In *QoSA*, pages 127–144, 2006. 27

[BMP07] A. Bucchiarone, H. Muccini, and P. Pelliccione. Architecting Fault-tolerant Component-based Systems: from Requirements to Testing. *ENTCS*, 168:77–90, 2007. 27

[Bos99] P. Bose. Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. In *ASE '99*, page 102. IEEE Computer Society, 1999. 26

[BS06] L. Baresi and P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *ICGT*, pages 306–320, 2006. 76

[CC03] 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. 4

[CDE$^+$07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. 2007. 26

[CEK$^+$04] R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable Dynamic Plugin Systems. In *FASE*, pages 129–143, 2004. 4

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programmming Language Systems*, 8(2):244–263, 1986. 41, 85

[CHA] CHARMY. `http://www.di.univaq.it/charmy`. 26, 99

[CHG$^+$04] S. Cheng, A. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In *ICAC*, pages 276–277, 2004. 4

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph Processes. *Fundam. Inform.*, 26(3/4), 1996. 23

[CMR$^+$97] 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. 23, 40, 45, 46, 50

[CPT99] C. Canal, E. Pimentel, and J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *WICSA'99*, pages 107–126. Kluwer, 1999. 15, 16

[DHP02] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical Graph Transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002. 4, 131

[DN02] L. Dobrica and E. Niemelä. A Survey on Software Architecture Analysis Methods. *TSE*, 28(7):638–653, 2002. 25

[DV90] 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. 41, 85

[DV95] R. De Nicola and F. W. Vaandrager. Three Logics for Branching Bisimulation. *ACM*, 42(2):458–487, 1995. 86

[DvdHT01] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA*, pages 103–112, 2001. 16

[E. 00] E. M. Clarke and O. Grumberg and D. A. Peled. *Model Checking*. The MIT Press, January 2000. 25

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In *Handbook of Graph Grammars*, pages 247–312, 1997. 45, 46, 50, 70

[End94] M. Endler. A Language for Implementing generic Dynamic Reconfigurations of Distributed Programs. In *Proceedings of BSCN'94*, pages 175–187, 1994. 4, 5, 14, 51, 57

[EU] EU project SENSORIA (IST-2005-016004). `http://www.sensoria-ist.eu`. 4, 8, 44, 102

[FBF$^+$07] R. B. Franca, J. P. Bodeveix, M. Filali, J. F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex – experiments and roadmap. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 377–382. IEEE Computer Society, 2007. 12

[FECA05] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. 16

[FM97] J. L. Fiadeiro and T. S. Maibaum. Categorical Semantics of Parallel Program Design. *Sci. Comput. Program.*, 28(2-3):111–138, 1997. 24

[Fuj] Fujaba tool suite. `http://wwwcs.uni-paderborn.de/cs/fujaba`. 28, 39

[Gar01] D. Garlan. Software Architecture. In *Encyclopedia of Software Engineering, John Wiley & Sons*, 2001. 2

[GKRS06] K. Geihs, M. U. Khan, R. Reichle, and A. Solberg. Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices. In *IFIP Working Conference on Software Engineering Techniques*, 2006. Warsaw, Poland, October 18-20, 2006. 132

[GMK02] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of WOSS'02, 1st Workshop on Self-Healing Systems*, pages 33–38, 2002. 51, 57

[GMS04] V. Grassi, R. Mirandola, and A. Sabetta. A UML Profile to Model Mobile Systems. In *UML*, pages 128–142, 2004. 20, 22

[GMS05] V. Grassi, R. Mirandola, and A. Sabetta. An XML-Based Language to Support Performance and Reliability Modeling and Analysis in Software Architectures. In *QoSA/SOQUA*, pages 71–87, 2005. 16

[GMW97] D. Garlan, R. T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *CASCON'97*, pages 169–183, 1997. 13

[GN02] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver, 2002. 65

[GS02] D. Garlan and B.R. Schmerl. Model-based Adaptation for Self-healing Systems. In *Proceedings of WOSS'02, First Workshop on Self-Healing Systems*, pages 27–32, 2002. 4, 52, 57

[GS06] D. Garlan and B. Schmerl. Architecture-driven Modelling and Analysis. In *SCS '06: Proceedings of the eleventh Australian workshop on Safety critical systems and software*, pages 3–17. Australian Computer Society, Inc., 2006. 47

[Hac04] F. Hacklinger. Java/A - Taking Components into Java. In *IASSE'04*, pages 163–168. ISCA, 2004. 10, 31, 32

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundam. Inform.*, 26(3-4):287–313, 1996. 56, 131

[HIM00] D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In *Proceedings of COORDINATION'00, 4th International Conference on Coordination Languages and Models*, volume 1906 of *LNCS*, pages 148–163, 2000. 10, 24, 25, 51, 57

[HKMU06] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for Software Architectures. In *EWSA*, pages 113–126, 2006. 4, 102

[HM85] M. Hennessy and R. Milner. Algebraic Laws for NonDeterminism and Concurrency. *ACM*, 32(1):137–161, 1985. 86

[HNS98] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1998. 2

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 24

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003. 26

[HUG05] HUGO. Public web site, 2005. `http://www.pst.ifi.lmu.de/projekte/hugo`. 32, 35

[IMP05] P. Inverardi, H. Muccini, and P. Pelliccione. CHARMY: an extensible tool for architectural analysis. In *ESEC/SIGSOFT FSE*, pages 111–114, 2005. 26

[Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *TOSEM*, 11(2):256–290, 2002. 7, 39, 58, 59

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006. 7, 39, 58, 59, 130

[JB05] C. Jerad and K. Barkaoui. On the Use of Rewriting Logic for Verification of Distributed Software Architecture Description Based LfP. In *RSP'05*, pages 202–208. IEEE, 2005. 26

[KJKD05] M. H. Kacem, M. Jamiel, 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. 4, 10

[KM07] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. In *FOSE '07: Future of Software Engineering*, pages 259–268, 2007. 4

[Koc07] N. Koch. Automotive Case Study: UML Specification of On Road Assistance Scenario. Technical report, FAST, 2007. 102

[L. 04] L. Baresi and R. Heckel and S. Thone and D. Varro. Style-Based Refinement of Dynamic Software Architectures. In *WICSA'04*, pages 155–166. IEEE, 2004. 4, 51, 57

[LCV00] B. Lewis, E. Colbert, and S. Vestal. Developing Evolvable, Embedded, Time-Critical Systems with MetaH. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 447, Washington, DC, USA, 2000. IEEE Computer Society. 12

[Le 98] D. Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *TSE*, 24(7):521–533, 1998. 5, 10, 19, 23, 25, 51, 57

[LKW93] M. Löwe, M. Korff, and A. Waner. An Algebraic Framework for the Transformation of Attributed Graphs. In *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley & Sons Ltd, 1993. 55

[MBO$^+$07] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R. M. Greenwood. An Active Architecture Approach to Dynamic Systems Co-evolution. In *ECSA'07*, volume 4758 of *LNCS*, pages 2–10, 2007. 4, 13

[MCM00] J. Merseguer, J. Campos, and E. Mena. Evaluating Performance on Mobile Agents Software Design, 2000. 20, 22

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *ESEC'95*, volume 989 of *LNCS*, pages 137–153, 1995. 14, 28

[MHKD06] M. Jmaiel M. H. Kacem, A. H. Kacem and K. Drira. Describing dynamic software architectures using an extended UML model. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1245–1249. ACM, 2006. 19, 22

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-calculus*. Cambridge University Press, 1999. 13

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *SIGSOFT FSE*, pages 3–14, 1996. 10, 14, 28

[MKG99] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *WICSA*, pages 35–50, 1999. 25

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001. 65

[MPM07] A. N. Martnez, M. A. Prez, and J. M. Murillo. AspectLEDA: Extending an ADL with Aspectual Concepts. In Flvio Oquendo, editor, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 330–334. Springer, 2007. 16

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Inf. Comput.*, 100(1):1–77, 1992. 14

[MRRR02] N. Medvidovic, D. S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *TOSEM*, 11(1):2–57, 2002. 19

[MRT99] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *ICSE'99*, pages 44–53. ACM, 1999. 19

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *TSE*, 26(1):70–93, 2000. 4, 10, 12

[MWN+04] G. Mustapic, A. Wall, C. Norstrom, I. Crnkovic, K. Sandstrom, and J. Andersson. Real world influences on software architecture - interviews with industrial system experts. In *Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004*, pages 101–111, June 2004. 3

[OGT+99] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An Architecture-based Approach to Self-Adaptive Software. In *Intelligent Systems, IEEE*, pages 54–62, 1999. 4, 51, 57

[OMG03] OMG. OCL 2.0 specification. Available from http://www.omg.org, 2003. 15, 19

[Ore96] P. Oreizy. Issues in the Runtime Modification of Software Architecture. Technical Report UCI-ICS-96-35, University of California, 1996. 51, 57

[PACR06] Jennifer Pérez, Nour Ali, José A. Carsí, and Isidro Ramos. Designing software architectures with an aspect-oriented architecture description language. In *CBSE*, pages 123–138, 2006. 17, 28

[Pel05] P. Pelliccione. *CHARMY: A framework for Software Architecture Specification and Analysis*. PhD thesis, Computer Science Department, University of L'Aquila, May 2005. 26

[PG03] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing: Introduction. In *Communications of the ACM*, volume No. 10, pages 24–28, October 2003. 21

[PMSA04] J.E. Pérez-Martínez and A. Sierra-Alonso. UML1.4 versus UML2.0 as languages to describe software architecture. In *EWSA04*, volume 3047 of *LNCS*, pages 88–102, 2004. 19

[Pre29] R.S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, Singapore, 1987; 28-29. 3

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992. 29, 47

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, pages 479–485, 2003. 23

[RKJ04] S. Roh, K. Kim, and T. Jeon. Architecture Modeling Language based on UML2.0. In *APSEC'04*, pages 663–669. IEEE, 2004. 19

[Ros06] Rosatea '06: Proceedings of the issta 2006 workshop on role of software architecture for testing and analysis, 2006. Conference Chair-Rob Hierons and Conference Chair-Henry Muccini. 25

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing Co., Inc., 1997. 45, 46, 50

[RV93] J. C. Raoult and F. Voisin. Set-Theoretic Graph Rewriting. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, pages 312–325, 1993. 23

[SAE] SAE Architecture Analysis and Design Language. http://www.aadl.info. 12, 28

[SAP$^+$07] C. C. Soria, N. Ali, J. Pérez, J. A. Carsí, and I. Ramos. Dynamic Reconfiguration of Software Architectures Through Aspects. In *ECSA*, pages 279–283, 2007. 17

[SG96] M. Shaw and D. Garlan. Software Architecture: Perspectives on An emerging Discipline. In *Prentice Hall, NJ. USA*, 1996. 3, 9, 11, 47, 49

[SG02a] B.R. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of SEKE'02, 14th international conference on Software Engineering and Knowledge Engineering*, pages 241–248, 2002. 51, 57

[SG02b] J. P. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *WICSA*, pages 29–43, 2002. 4

[SG04] B. R. Schmerl and D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *ICSE'04*, pages 704–705. ACM, 2004. 35

[SV03] A. Schmidt and D. Varro. CheckVML: A Tool for Model Checking Visual Modeling Languages. In *UML 2003*, volume 2863 of *LNCS*, pages 9–95, 2003. 23, 39

[SWZ99] A. Schürr, A. Winter, and A. Zündorf. PROGRES: Language and Environment. In H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2, pages 487–550. World Scientific, 1999. 23

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 9, 12

[tFGM08] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/state-based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In *FMICS'07*, volume 4916 of *LNCS*, pages 133–148, 2008. 27, 41, 82, 85

[tGKM08] M. H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti. Formal Verification of an Automotive Scenario in Service-Oriented Computing. In *ICSE'08*, pages 613–622. ACM, 2008. 102

[Thö05] Sebastian Thöne. *Dynamic Software Architectures*. Phd thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, Germany, October 2005. 23, 25

[TMD08] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, 2008. 2

[UMC] UMC model checker. `http://fmt.isti.cnr.it/umc`. 7, 27, 41, 82, 86, 87

[Wer98] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. *IEE Proceedings - Software*, 145(5):130–136, 1998. 5, 57

[WF02] M. Wermelinger and J. L. Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002. 10, 24, 25

[YM92] A. Young and J. Magee. A Flexible Approach to Evolution of Reconfigurable Systems. In *Proceedings of IWCDS'92*, IEE, pages 152–163, 1992. 57