Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 236 (2009) 117–130

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Graph-based Design Framework for Global Computing Systems

Antonio Bucchiarone[a,1] Greg Dennis[b,2] Stefania Gnesi[a,3]

[a] *Istituto di Scienze e Tecnologie dell'Informazione "A. Faedo", Area della Ricerca CNR di Pisa*

[b] *Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA*

Abstract

We present a framework for designing and analyzing Global Computing Systems using Dynamic Software Architectures. The framework, called TGG_A, integrates typed graph grammars and the Alloy modeling language to specify Programmed Dynamic Software Architectures that represent systems that evolve their topology at runtime. We demonstrate the benefits of the framework by applying it to the study of an Automotive Software System.

Keywords: component-based software systems, dynamic software architectures, graph grammars, Alloy

1 Introduction

Modern software systems have changed from isolated static devices to highly interconnected machines that execute their tasks in a cooperative and coordinate manner. These systems are known as *Global Computing Systems* (GCSs), and have to deal with frequent changes of the network environment. Moreover the structure of them is dynamic with continuous changes. The following properties characterizes a GCS:

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

[1] Email: antonio.bucchiarone@isti.cnr.it
[2] Email: gdennis@mit.edu
[3] Email: stefania.gnesi@isti.cnr.it

doi:10.1016/j.entcs.2009.03.018

- **Mobility**: some computational entities are mobile, due to the movement of the physical platforms or by movement of entities from one platform to another.
- **Scalability**: GCSs are able to start small and then expand over time in terms of size (i.e., more users, devices and connections) and functionalities (i.e., new service requests) insuring system availability.

Software architectural models are intended to describe the structure of a system in terms of components, their interactions, and their composition patterns [31]. Since a GCS topology may change at run-time, Software Architecture (SA) models should be able to describe the change of the system and to enact those modifications during the system execution. Such models are referred as *Dynamic Software Architectures* (DSAs) [2,3,22], to emphasize that the system architecture evolves during runtime. A variety of definitions of dynamicity for SA have been proposed in literature. Below we list some of the most prominent definitions to show the variability of connotations of the word *dynamic*.

- **Programmed [12]:** All admissible changes are defined prior to runtime and are triggered by the system itself.
- **Self-Repairing [15]:** Changes are initiated and assessed internally, i.e., the runtime behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed.
- **Self-adaptive [25]:** Systems can adapt to their environments by enacting runtime changes.
- **Ad-hoc [12]:** Changes are initiated by the user as part of a software maintenance task, they are defined at run-time and are not known at design-time.
- **Constructible [3]:** All architectural changes must be described in a given modification language, whose primitives constrain the admissible changes.

In this paper we first formalize Programmed Dynamic Software Architecture (PDSA) using typed graph grammars (TGGs), and next implement each aspect of TGGs by Alloy [21] in order to have a complete framework able to design and validate GCSs. The name of this framework is TGG_A (where A indicates the use of Alloy) and its objective is to permit to a software architect the design of the set of possible structural evolutions that a GCS can support at runtime and the verification of a set of invariant properties that each system configuration must satisfy.

1.1 Motivations

We have chosen TGGs as formal representation of GCSs since that (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 and, (iii) they have been largely used for specifying architectures [5,19,24,32].

In [8], one author of this paper, experimented with DynAlloy [14] to implement the TGG framework. The DynAlloy specification was constituted of one large monolithic file impeding its readability and usability. With DynAlloy was very difficult to debug the model to find and correct errors during the specification. Moreover, with DynAlloy was very difficult to support the formal specification directly. For these reasons we have chosen Alloy [20] to support the formal specification. Alloy provides a description language to represent software models, based on signatures and relations, which we found to be very suitable to model hypergraphs associated to PDSAs. Alloy is moreover based on a simple notation, which makes it easy to start working with it. Additionally, it provides an extension of first-order logic with relational operators to represent properties and constraints. The Alloy Analyzer can translate the model and the logical predicates into a boolean formula, using efficient SAT solvers to decide satisfiability, and provide a counterexample in case the predicate does not hold over the model. Before to choose Alloy we also tried to use graph transformation tools like AGG [1], PROGRES [26], Fujaba [27], CheckVML [30] and GROOVE [16]. All of these allow to design typed and attributed graphs. Full support of cardinality constraints, including automatic constraint checking, is only provided by AGG. Except for CheckVML, which is intended to translate graph transformation models into the input language of model checkers, all these tools allow the execution of graph transformation rules, even with negative application conditions. Their limits are in the possibility of designing typed hypergraphs. The only tool one that supports typed hypergraphs is Graph eXchange Language GXL [17], but it lacks in verification aspects. In conclusion, the above considerations made us to select Alloy as our tool.

1.2 Structure of the paper

The rest of the paper is structured as follows. In Section 2 we introduce our PDSA formalization using TGGs. Section 3 presents the Alloy implementation of each TGG aspects introduced, while Section 4 shows in which way we can design an automotive system using TGG_A. Section 5 concludes our paper with some conclusions and future work.

2 Programmed DSAs as Typed Graph Grammars

The approach described in this section follows what is discussed in [7]. It is based on modelling of dynamic software architectures using typed graph grammars (TGGs). It introduces each aspect that will be implemented in Section 3 using Alloy. These aspects come from the theory of graph grammars [9,28]. We concentrate our approach on *typed hypergraph rewriting systems* in the *single-pushout* (SPO) approach [11]. 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., the so-called type graph). In the following we present a set of definitions that we will use in our formalization.

Definition 2.1 [Hypergraph] A *(hyper)graph* is a triple $G = (N_G, E_G, \phi_G)$, where

N_G is the set of nodes, E_G is the set of (hyper)edges, and $\phi_G : E_G \to N_G^+$ describes the connections of the graph, where N_G^+ stands for the set of non-empty sequences of nodes. We call $|\phi_G(e)|$ the *rank* of e, with $|\phi_G(e)| > 0$ for any $e \in E_G$.

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

Definition 2.2 [Graphs Morphism] Let G and G' be two graphs. A pair of functions $\langle f_N, f_E \rangle$ where $f_N : N_G \to N_{G'}$ and $f_E : E_G \to E_{G'}$ is a graph morphism from G to G' if $f_N(\phi_G(e)) = \phi_{G'}(f_E(e))$.

Definition 2.3 [Typed Hypergraph] Let `T` be a graph. A typed graph `G` over `T` is a graph $|G|$, together with a graph morphism $\tau : |G| \to T$. A morphism between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with the typing, i.e. such that $\tau_{G_1} = \tau_{G_2} \circ f$.

The basic idea of graph rewriting is to consider a set of *graph rewriting rules* (i.e., reconfigurations) of the form $p : L \to R$, 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*. $G \to^p G'$ denotes the direct derivation, where the *match* $m : L \to G$ fixes an occurrence of L in a graph G. Application of rule p yields a *derived graph* G' 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, given a graph G and a rewriting rule p, a reconfiguration of G using p is realised using a single-pushout graph transformation approach [11].

Definition 2.4 [SPO direct derivation] Given a typed graph `G`, a rewriting rule `p`, and a match $m : L \to G$, we say that there is a *direct derivation* r' *from* G *to* G' *using* p, written r': $G \to_p G'$, if, for suitable morphisms *r'* and *m'*, we have the following pushout square:

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

Fig. 1. SPO-based graph rewriting.

Definition 2.5 [(T-typed) graph grammar] A *(T-typed) graph grammar* $\mathcal{G}$ is a tuple $\langle T, G_{in}, P \rangle$, where T is the Type Graph, G_{in} is the *initial (T-typed) graph* and P is a set of *rewriting rules*.

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, $\mathcal{R}(\mathcal{G}) = \{G \mid G_{in} \Rightarrow^* G\}$.

- The set $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$ of *desirable configurations* are defined as the graphs that have type T and satisfies a suitable property P.
  Formally, $\mathcal{D}_{\mathsf{P}}(\mathcal{G}) = \{G \mid G \textit{ is a } T\textit{−typed graph} \wedge \mathsf{P} \textit{ holds in } G\}$.

Programmed dynamism enables 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* $G \in \mathcal{R}(\mathcal{G})$.

In the remainder we omit the prefix 'hyper' for simplicity.

2.1 Formalization of Programmed DSAs

In this paper we consider PDSAs. They assume that all architectural changes are identified at design time and triggered by the program itself [12]. Many proposals in the literature [4,19,24] that use graph grammars for specifying PDSA present this kind of dynamism. A PDSA $\mathcal{A}$ is associated with a T-Typed graph grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P\rangle$, where T stands for the style of the PDSA, G_{in} is the initial configuration, and the set of rewriting rules P gives the evolution of the architecture. The grammar fixes the types of all elements in the architecture, and their possible connections, where the rewriting rules state the possible ways in which a configuration may change.

Each *Software Architecture configuration* of a PDSA is represented by a graph where components (resp. connectors) are modelled using edges and their ports (resp. roles) by the outgoing tentacles. Components and connectors are attached together connecting their respective ports to the same node. 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. 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.

3 Alloy Implementation

The implementation of each concept introduced in the previous section has been done using Alloy [20,21], a light-weight approach to the modelling and analysis of software models. Since that we use typed graph grammars to represent PDSAs, 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 PDSAs.

The main aspects on which we focus are concerned with:

- *representation*, i.e. convenient ways to design a PDSA, to build it, to browse it;
- *styles*, i.e. convenient ways to constrain the PDSAs under consideration to satisfy certain requirements;
- *properties*, i.e. convenient logical formalisms to express relevant structural properties;
- *analysis*, i.e. efficient techniques and tools for verification.

We show how to tackle these aspects with our approach. The outcome of our experience indicate 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.

We use Alloy to implement graphs, graphs morphisms, graph transformations, etc. Starting from this core implementation (i.e., the TGG Alloy Module) we have extended it to represent concepts like styles, configuration, reconfigurations, etc. The relations between our formal method, PDSA elements and Alloy implementations are summarized in Table 1.

Formal Elements	**PDSA Elements**	**Alloy Implementation**
Typed Hypergraph	Configuration	Signature
Type Graph	Style	Signature
Typed Partial Morphism	Reconfigurations rule	Signature
SPO graph rewriting (single step)	Evolution (single evolution)	Predicate
Typed Graph Grammar	PDSA	Facts

Table 1
TGG_A Elements

3.1 TGG_A in detail

In this section, for each aspect of the first column in Table 1 we show the respective Alloy implementation comprising the particular TGG Alloy Module. It results in the implementation of the TGG Alloy Module. After this section, using this module, we present the design and verification of an automotive software system.

3.1.1 Graphs

The three basic concepts in the model of each graph are *nodes*, *ports* and *edges* that are represented as three Alloy signatures listed in Listing 1.

```
1  abstract sig Node {}
2  abstract sig Port {}
3  abstract sig Edge {
4    conn: Port -> lone Node
5  }{
6    // non-empty connections
7    some conn
8  }
9  // Graph Definition
10 abstract sig Graph {
11   edges: set Edge,
12   nodes: set Node
13 }{
14   // edges connected to nodes in the graph
15   edges.nodes in nodes
16 }
```

Listing 1: Graphs

According to the above definition, nodes and ports are atomic concept, while edges has a field `conn` that maps each port to nodes. The keyword `lone` in the declaration indicates *multiplicity*, in this case that each port is mapped to at most one node. The signature `Graph` (lines 10-16 of Listing 1) is used to define as a graph as structure composed of nodes and edges.

3.1.2 Typed Graphs and Type Graph

To generate graphs that are conform a given type graph, we have defined in Alloy the signature `TypedGraph` that is a graph with a typing morphism. Listing 2 presents the implementation of typed graphs. It is important to note that the target graph of the morphism in each typed graph is the style of the system (line 2). A type graph consists of a set of basic elements (i.e., Edge, Node, and Ports) that can constitute a typed graph plus a set of constraints indicating how these elements can be lagally connected. For each new system that we want to design we must define an Alloy module that contains all these elements. The style of our case study is described in section 4.2.1.

```
1 sig TypedGraph extends Graph {
2   style: Graph,
3   fN: Node -> Node,
4   fE: Edge -> Edge
5 }{
6   fN in nodes -> one style.@nodes
7   fE in edges -> one style.@edges
8   all e: edges | (e.fE).conn = e.(conn.fN)
9 }
```

Listing 2: Typed Graph

3.1.3 SPO Graph Rewriting and Rewriting Rules

Given an initial graph G_{in} and a rewriting rule p, a rewriting of G_{in} using p is implemented in Alloy using the single-pushout graph transformation approach [11]. To implement it we have defined the predicate `rwStep` that executes one single rewriting step and produces the target graph G' (lines 7-9 of the Listing 3). In order to apply a rewriting step we must define a set of rewriting rules that state the possible ways in which a typed graph may change. Each rule is defined as typed partial morphisms $p : L \rightarrow R$ (shown in Listing 3 lines 1-5), where L and R are typed graphs (i.e., source and target). For each new system we specify an Alloy module that contains all reconfigurations rules that our system can execute at runtime.

```
1  // a rewrite rule
2  abstract sig Rewrite {
3    source: TypedGraph,
4    target: TypedGraph
5  }
6
7  pred rwStep[g, g': TypedGraph] {
8    some rw: Rewrite | rw.source = g && rw.target = g'
9  }
```

Listing 3: SPO rewriting

3.1.4 Typed Graph Grammars

The typed graph grammar associated to each PDSA has been defined using the code described in the listing 4. It uses an Alloy module (i.e., util/ordering) that creates a linear ordering over typed graphs. Each new typed graph (i.e., configuration) is generated executing a single rewriting step (line 4).

```
1  open util/ordering[TypedGraph]
2  fact grammar {
3    all tg: TypedGraph - last | let tg' = next[tg] |
4      tg.style = tg'.style && rwStep[tg, tg']
5  }
```

Listing 4: Typed Graph Grammar

3.2 Structural Analysis of a PDSA

In order to validate a PDSA for a structural point of view, we list a set of analysis techniques that we can perform using TGG_A.

- **Model Finding:** it is the main analysis capability offered by TGG_A. 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 the initial configuration: we need to ask for a graph instance satisfying the style facts and having a certain number of edges and nodes.
- **Invariant Analysis:** The objective of this analysis is: given a property P is invariant under sequences of applications of some operations. In our case this operation is the rewriting step. 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 an Alloy module that we present in the next section.

In the next section we use TGG_A to design an Automotive Software System.

4 Automotive Case Study

The case study presented in this paper is discussed at various places within to the Sensoria project [6,23,13]. In the next section we introduce the Route Planning System that is a sub-system of the Automotive system presented in [13].

4.1 Route Planning System

The 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. The RPS searches a sight seeing database for appropriate sights and displays them on the in-car map of the vehicle' navigation system.
- **Low Oil:** During a drive, the vehicles oil lamp reports low oil levels. This trigger 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 Road Assistance Service (RAS) that finds a best solution (Towing Service, Repair Shop, Rent a Car).
- **Bank Request for Mobile Phone Recharge:** The Bank represent an institution that provides financial services. The bank operations that are relevant for the RPS application are the charge of a credit in the driver mobile phone.
- **Vehicles Connection:** 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.

4.2 Design and Verification of RPS using TGG_A

In this section we design and verify our case study using TGG_A. Each configuration of the PDSA of the RPS system can be composed of components depicted in Figure 2, which we now describe briefly, while Table 2 spells out the abbreviations used for ports of each component.

BANK: provides the bank operations that are relevant for the RPS application;
GPS: provides Global Positioning System data (like the current position of the vehicle);
Local Discovery (LD): looks for appropriate services in the local repository;
RAS: provides services required for car repair (tow truck, garage and car rental);
External Vehicle (EV): another vehicle that can be connected during a trip;

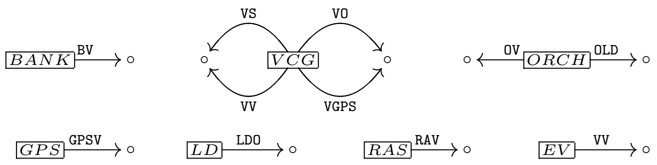

Fig. 2. Basic components of the RPS.

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 2
Description of ports.

Vehicle Communication Gateway (VCG): forwards messages to external components (*BANK*, *RAS*, *EV*);
Orchestrator (ORCH): in charge to achieve a goal by composing services, so each time a driver's request arrives it performs a dynamic binding with in external components *BANK*, *GPS*, *RAS*, *EV*, etc.

4.2.1 RPS Style

After the identification of each component that compose the RPS system, we have defined the RPS Style to which each PDSA configuration should adhere. To do this this we have defined an Alloy module that implements the `Type Graph` of Figure 3. The full Alloy code can be downloaded [29].

4.2.2 Structural analysis of the RPS

Our framework offers *Model Finding* and *Invariant Analysis* to validate a PDSA's structure. For the first one we implemented a module called `Model Finding` which, starting from the definition of the elements of our initial configuration (like components, ports, etc.), generates the set of all possible PDSA configurations composed of a precise number of components, ports and attachments. When the Alloy `run` com-

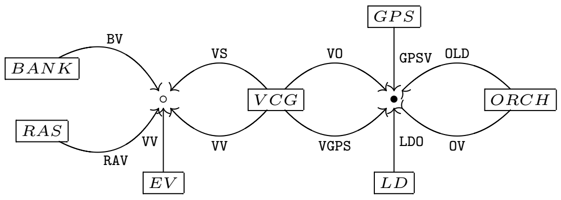

Fig. 3. Type Graph of RPS

mand is executed, the Alloy analyzer generates all possible RPS-style-conformant configurations. Two of them are shown in Fig. 4, with ia_i nodes representing communications between components inside the car and ea_j nodes communications between internal components and components in the external environment.

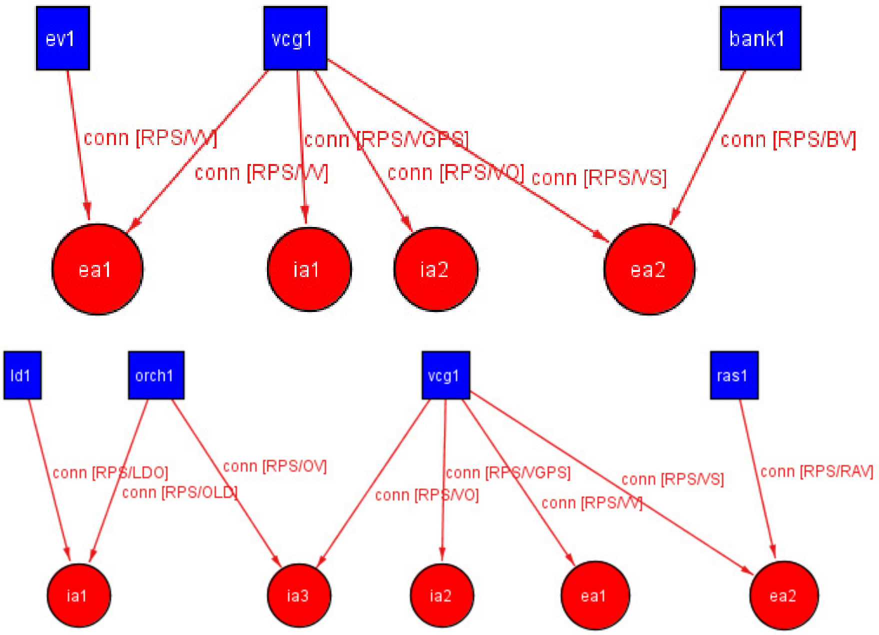

Fig. 4. Two possible style-conformant PDSA configurations of the RPS.

To execute *Invariant Analysis* we defined a set of reconfiguration rules, a set of properties to verify and, finally, an Alloy predicate `Invariant Analysis` that will be used to verify the invariants of a set of properties over our system. Fig. 5 shows a set of possible reconfiguration rules (i.e. GPS Request, Request Assistance and Call Friend) while Fig. 6 presents an example of a possible run-time evolution of the RPS as a set of transitions, each composed of a *startingState*, an *arrivalState* and a *trigger* (i.e. reconfiguration rules).

Moreover we defined some properties that each DSA configuration of the RPS must satisfy after each reconfiguration step, such as:

Property 1: no component *VCG* is connected directly to a component *LD*;

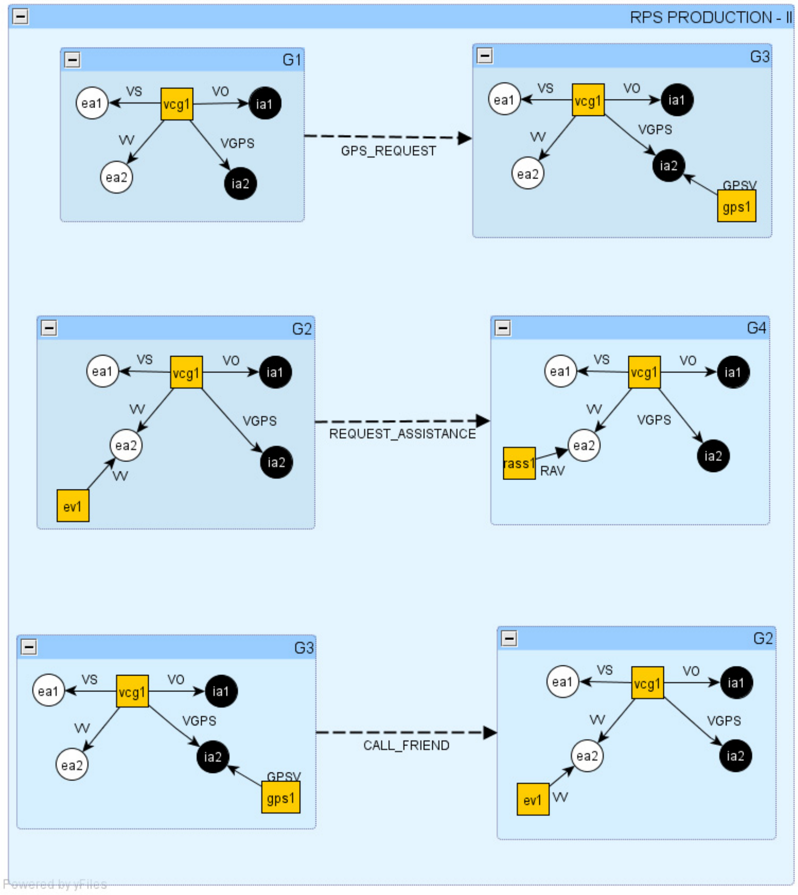

Fig. 5. Reconfiguration rules of the RPS.

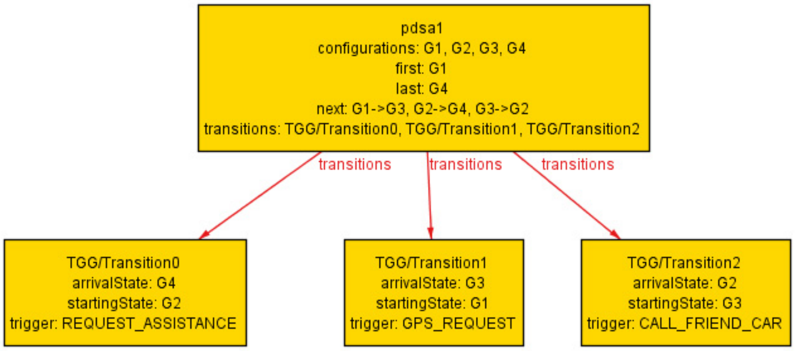

Fig. 6. Run-time evolution example of the RPS.

Property 2: if a component *LD* exists, then so does a component *ORCH* and the two must be connected;
Property 3: component *ORCH* has exactly two connections;
Property 4: a *VCG* component cannot have more than one component attached to each port.

If we check the `Invariant Analysis` predicate for each of these properties, then we obtain, using the Alloy Analyzer, that each of these properties is valid for each reachable configuration of our running example. Considering the PDSA formalization presented in [7] we can conclude that the structural specification of the RPS is correct and complete.

5 Conclusions and Future Work

In this paper we have presented TGG_A, a tool supported framework to design Global Computing Systems. It is based on modelling Dynamic Software Architectures as typed graph grammars and implementing it using Alloy. Moreover the Alloy Analyzer has been used to ensure style-consistency, perform model-finding and validate structural properties of each PDSA configuration. This framework is well suited to a reactive modelling process: the software architect builds a model and the system reacts reporting style inconsistencies. TGG_A defined 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_A expresses them by means of the same formalism used to define architectural styles, i.e. the Alloy logic. As kind of architectural dynamism, we have only considered the programmed dynamicity, in which all admissible changes are defined prior to run-time and are triggered by the system itself. Other future research extends our modeling approach. Related to this there is the necessity to use graph grammars with negative application conditions [18] 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 DSAs that have as basic elements some more complex and structured components. We are thinking about using hierarchical hypergraphs [10] where each hyperedge can represent relations among components.

Acknowledgement

This work has been partially funded by the EU project Sensoria (IST-2005-016004) and by the Italian project tocai.it.

References

[1] AGG. URL http://tfs.cs.tu-berlin.de/agg/

[2] Allen, R., R. Douence and D. Garlan, *Specifying and Analyzing Dynamic Software Architectures*, in: *FASE'98*, LNCS **1382**, 1998, pp. 21–37.

[3] Andersson, J., *Issues in dynamic software architectures*, in: *ISAW'00*, 2000, pp. 111–114.

[4] Baresi, L., R. Heckel, S. Thöne and D. Varró, *Style-Based Refinement of Dynamic Software Architectures*, in: *WICSA'04* (2004), pp. 155–166.

[5] Baresi, L., R. Heckel, S. Thöne and D. Varró, *Style-based modeling and refinement of service-oriented architectures*, Software and System Modeling **5** (2006), pp. 187–207.

[6] Berndl, D. and N. Koch, *Automotive Scenario: Illustrating Service Specification*, Technical Report no. 2, FAST-Technical Report (2007).

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

[8] Bucchiarone, A. and J. P. Galeotti, *Dynamic Software Architectures Verification using DynAlloy*, in: *GT-VMT'08*, ECEASST **10**, 2008.

[9] Corradini, A., 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: *[28]*, 1997, pp. 163–246.

[10] Drewes, F., B. Hoffmann and D. Plump, *Hierarchical graph transformation*, J. Comput. Syst. Sci. **64** (2002), pp. 249–283.

[11] Ehrig, H., 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: *[28]*, 1997, pp. 247–312.

[12] Endler, M., *A language for implementing generic dynamic reconfigurations of distributed programs*, in: *BSCN'94*, 1994, pp. 175–187.

[13] EU project Sensoria. URL http://www.sensoria-ist.eu/

[14] Frias, M., J. Galeotti, C. L. Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in: *ICSE'05* (2005), pp. 442–450.

[15] Garlan, D. and B. Schmerl, *Model-based adaptation for self-healing systems*, in: *WOSS'02* (2002), pp. 27–32.

[16] GROOVE. URL http://groove.sourceforge.net/

[17] GXL. URL http://www.gupro.de/GXL/

[18] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundam. Inform. **26** (1996), pp. 287–313.

[19] Hirsch, D., P. Inverardi and U. Montanari, *Reconfiguration of Software Architecture Styles with Name Mobility*, in: *COORDINATION'00*, LNCS **1906** (2000), pp. 148–163.

[20] Jackson, D., *Alloy: A Lightweight Object Modelling Notation*, TOSEM **11** (2002), pp. 256–290.

[21] Jackson, D., "Software Abstractions: Logic, Language, and Analysis," The MIT Press, 2006.

[22] Kacem, M. H., 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'05*, 2005, pp. 189–195.

[23] Koch, N., *Automotive Case Study: UML Specification of On Road Assistance Scenario*, Technical Report No. 1, FAST-Technical Report (2007).

[24] Métayer, D. L., *Describing Software Architecture Styles Using Graph Grammars*, IEEE Trans. Softw. Eng. **24** (1998), pp. 521–533.

[25] Oreizy, P., M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum and A. Wolf, *An architecture-based approach to self-adaptive software*, IEEE Intelligent Systems and their Applications **14** (1999), pp. 54–62.

[26] PROGRES. URL http://www-i3.informatik.rwth-aachen.de/research/projects/progres/

[27] Project, F. URL http://wwwcs.uni-paderborn.de/cs/fujaba/index.html

[28] Rozenberg, G., editor, "Handbook of graph grammars and computing by graph transformation Vol. 1," World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[29] RPS Code. URL http://www.antoniobucchiarone.it/code/RPSCode.zip

[30] Schmidt, A. and D. Varró, *CheckVML: A Tool for Model Checking Visual Modeling Languages*, in: *UML'03*, LNCS **2863**, 2003, pp. 9–95.

[31] Shaw, M. and D. Garlan, "Software Architecture: Perspectives on An emerging Discipline," Prentice Hall, NJ, USA, 1996.

[32] Wermelinger, M. and J. L. Fiadeiro, *A graph transformation approach to software architecture reconfiguration*, Sci. Comput. Program. **44** (2002), pp. 133–155.