Dynamic Software Architectures Verification using DynAlloy

Antonio Bucchiarone[1,3] and Juan P. Galeotti[2]

IMT of Lucca, Italy[1]
a.bucchiarone@imtlucca.it
Universidad de Buenos Aires, Argentina[2]
jgaleotti@dc.uba.ar
ISTI-CNR of Pisa, Italy[3]

Abstract: Graph Grammars have been often used for modeling dynamic changes in software architectures. In particular, we have previously characterized some classes of dynamicity in terms of particular aspects of graph grammars. Moreover we have identified classes of properties that can be naturally associated to any of such kinds of dynamicities. In this paper we approach the problem of verifying such properties over graph grammars specifications. In particular, we use DynAlloy for attempting this task and we have concentrated on proving properties associated to a particular programmable dynamic software architecture.

Keywords: Dynamic Software Architectures, Typed Graph Grammars, Verification and DynAlloy

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. Therefore, the structure and the behavior of these systems is dynamic with continuously changes. These systems are known as *Global Computing Systems (GCS)*, and have to deal with frequent changes of the network environment. The principal characteristics that these systems have are summarized in the following: *Globality:* each GCS is composed of autonomous computational entities where activities are not centrally controlled, either because global control is impossible of 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:* each computational entity is 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 is always the source of each 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. The runtime behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed without compromise the 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 the system availability.

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 [Ore96], software architecture models for GCSs should be able to describe the changes of the system structure and to enact the modifications during the system execution. Such models are generally referred to as *Dynamic Software Architecture(*DSA*)*, to emphasize that the system architecture evolves during runtime. In this paper we select graph grammars [Roz97] as a formal framework to model DSA [BBGM07] and we approach the problem of verifying such properties over graph grammars specifications. In particular, we use DYNALLOY [FGLA05] for attempting this task proving properties associated to a particular programmable dynamic software architecture. We use graph grammars to model programmed DSA because they provide several advantages, these include: *(i)* a graphical representation of architectural styles and configurations that is in line with the usual way architectures are represented, *(ii)* independence from any particular solution technique, *(iii)* a formal basis based on the theory of formal languages [Roz97]. From the verification point of view, we might like to guarantee that a set of architectural structure changes (e.g., adding or removing components or connectors) will preserve some architectural invariants or, in other words, that a specific sequence of structural changes will result in a new Software Architecture that satisfies a particular property of interest. For this aspect we show how to use the ALLOY language to model our DSA based on graph grammars formalism and how to use DYNALLOY [FGLA05], an extension of the ALLOY language [Jac02, Jac06], to specify operations over architectures. Finally, we show how to use the ALLOY ANALYZER to check properties of interest for DSA . In Section 2 we introduce a running example that we use to introduce our idea. In Section 3 we summarize the principal elements of our formal framework that is used to represent DSA using hypergraphs. Then, in section 4, we show the way in which, using DYNALLOY structural properties of programmed dynamic SA are verified. Finally, related works are presented in Section 5, conclusions and future directions are shown in Section 6.

2 Running Example

We use as running example a simple scenario (see [BLMT07]) inspired by the automotive case study of Sensoria Project [Sen]. 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 1 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.

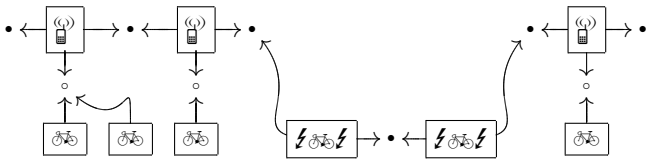

Figure 1: The road assistance scenario.

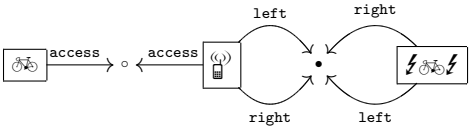

Figure 2: Type Graph T of the running example

3 Formal Model: A Typed Graph Grammar Approach

We model Software Architecture (SA) configurations using typed graph grammars [BBGM07]. Each SA is represented by an hypergraph where components (connectors) are modeled using hyperedges and their ports (roles) by the outgoing tentacles (i.e., labels). Moreover components and connectors are attached together connecting the respective tentacles to the same node. In the following we introduce the fundamental definitions that we will use in this 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. An architectural style is just a hypergraph T that describes only the types of ports, connectors, components and the allowed connections. A configuration compliant to such style is then described by the notion of a T-typed hypergraph.

Definition 2 (Typed Hypergraph) Given a hypergraph T (called the *style*), a *T-typed hypergraph* or *configuration* is a pair $\langle |G|, \tau_G \rangle$, where $|G|$ is the *underlying* graph and $\tau_G : |G| \to T$ is a total hypergraph morphism.

The graph $|G|$ defines the configuration of the system, while τ_G defines the (static) *typing* of the resources. We recall that a total hypergraph morphism $f : G \to G'$ is a couple $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ such that: $f_N(\phi_G(e)) = \phi_{G'}(f_E(e))$ (we overload f_N to denote also the homomorphic extension of f_N over strings).

Figure 2 depicts the type graph of our running example. It describes the types of components,

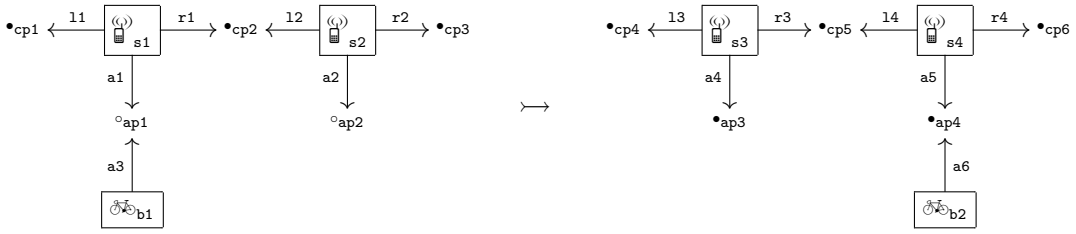

Figure 3: Reconfiguration rule that migrates a bike to the rightward station.

ports and their allowed connections. The typing morphism is defined using the τ_G that maps each element of the configuration in only one element of the type graph T.

Finally, an architecture is described by a T-typed graph grammar.

Definition 3 ((T-typed) graph grammar) A *(T-typed) graph grammar* $\mathscr{G}$ is a tuple $\langle T, G_{in}, Pr \rangle$, where G_{in} is the *initial (T-typed) graph* and Pr is a set of *productions*.

Notation. Let $\mathscr{G} = \langle T, G_{in}, Pr \rangle$ be a (T-typed) graph grammar, and G and H (T-typed) hypergraphs. We write $G \Rightarrow_p G'$ to denote that G is rewritten in one step to G' by using the production $p \in Pr$. We abbreviate the reduction sequence $G_0 \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} G_n$ with $G_0 \Rightarrow_{p_1 p_2 \ldots p_n} G_n$. We write $G \Rightarrow^* G'$ to denote that there exists a possible empty sequence $s \in Pr^*$ of derivation steps such that $G \Rightarrow_s G'$.

3.1 Software Architecture Reconfiguration

The reconfiguration of a software architecture is described by a set of rewriting productions that state the possible ways in which a SA configuration may change. Each rule is defined as a partial, injective graph morphism $p : L \rightarrow R$, where L and R are graphs, called the *left-* and *right-hand* side. Give a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation approach [EHK+97]. An application of p to a *host graph* G requires a partial graph morphism m from L to G called a *match*. A rewriting step leads to a *target graph* G'.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 G'. New nodes are not in the image of r' but in the image of m'. An example of production is shown in Figure 3, it specifies the migration of a bike from one station to the right station in a chain.

3.2 Programmed Dynamism

In this section we formalize programmed DSA in terms of graph grammars that will be used later to specify our running example. Given a grammar $\mathscr{G} = \langle T, G_{in}, Pr \rangle$, we will use the following notions:

- The set $\mathscr{R}(\mathscr{G})$ of *reachable configurations*, i.e., all configurations to which the initial configuration G_{in} can evolve. Formally, $\mathscr{R}(\mathscr{G}) = \{G | G_{in} \Rightarrow^* G\}$.

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

Programmed dynamism assumes that all architectural changes are identified at design time and triggered by the program itself [End94]. A programmed DSA $\mathscr{A}$ is associated with a grammar $\mathscr{G}_{\mathscr{A}} = \langle T, G_{in}, Pr \rangle$, where T stands for the style of the architecture, G_{in} is the initial configuration, and the set of productions Pr 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 $\mathscr{D}_{\mathsf{P}}(\mathscr{G})$, 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 $\mathscr{R}(\mathscr{G}) \subseteq \mathscr{D}_{\mathsf{P}}(\mathscr{G})$, or equivalently that $\forall G \in \mathscr{R}(\mathscr{G}) : \mathsf{P}$ *holds in* G.

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

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

4 DSA Structural Verification

4.1 DynAlloy

DynAlloy [FGLA05] is an extension of the Alloymodeling language. It allows us to define atomic actions that modify the state and build more complex actions (programs) from the atomic actions. Atomic actions are defined by means of preconditions and postconditions given as Alloy formulas. DynAlloy formulas extend Alloy formulas with the addition of a construct for building *partial correctness assertions*. From atomic actions we can build more complex programs as follows. If α is an Alloy formula, then $\alpha?$ is a test action. Operation $+$ denotes the nondeterministic choice between two programs, and ";" denotes their sequential composition. Finally, $*$ iterates a program. A partial correctness assertion of the form $\{\alpha\}p\{\beta\}$ is satisfied when no state that does not satisfy β. is reachable from $\{\alpha\}$ through program p.

One of the important features of Alloy is the automatic analysis possibilities it provides. In effect, the Alloy Analyzer allows us to automatically verify if a given assertion holds in an Alloy model. Similarly, in [FGLA05] it is showed how to translate DynAlloy specifications to Alloy specifications in order to achieve analyzability. This is due to imposing a bound to the depth of iterations (this is equivalent to fixing a maximum length of traces) and efficiently generating the *weakest liberal precondition* of the partial correctness assertion.

4.2 Designing Software Architectures and Styles with Alloy.

The approach described in this section follows [BBGM07] and models dynamic software architectures using typed graph grammars (TGG). The implementation of the approach is based on ALLOY [Jac06, Jac02]. ALLOY provides a logic, based on an extension of first-order logic with relational operators, to represent properties or constraints on the models. We have used this logic to implement concepts like architectural styles, graph transformation rules and architectural properties.

Each software architecture is represented by an hypergraph where components (or connectors) are modeled using hyperedges and their ports (or roles) by the outgoing tentacles (i.e. labels). Moreover, components and connectors are attached together connecting their respective tentacles to the same node. All basic elements (nodes, hyperedges and labels) are implemented in ALLOY using *signatures*. Instead, tentacles are defined as ternary relations between hyperedges, labels and nodes. All these elements are part of a graph signature definition that represents an architecture. Below we show an excerpt of the module TGG implementing the model of graphs. First we see the declaration of the basic signatures Node and Label which stand for nodes and labels. Signature Edge models hyperedges and includes a relation conn between signature labels and node. Note that in ALLOY syntax $\rightarrow$ indicates a binary relation. The keyword lone preceding Node constraints conn to relate each label to at most one element in Node. Thus, for each element of signature Edge, conn is partial function mapping labels to nodes. The signature Graph is used to define as a graph as structure composed of nodes, hyperedges and labels.

```
sig Node {}
sig Label {}
sig Edge { conn:  Label  -> lone Node }
sig Graph { n: set Node, he: set Edge, l: set Label }
```

After the definition of the signatures, we define a *predicate* (a property to be checked) to determine whether a graph g is well-formed and consistently typed over a type graph h by typing morphism t.

```
sig Tau {
  tauN: set Node -> set Node,
  tauE: set Edge -> set Edge,
  tauL: set Label -> set Label
}
pred isTG [g: Graph, h: Graph, t: Tau]{...}
```

The signature Tau is used to define mapping functions between each SA configuration and the architectural style. Architectural styles consist of a set of elements (components, connectors, ports and roles) that can constitute an architecture (e.g., *Bike_Vocabulary*) plus a set of invariant rules indicating how these elements can be legally connected [SG96]. A SA configuration compliant to such style is then described by the notion of a T-TYPED hypergraph [BBGM07]. We define an ALLOY module called BIKE-STYLE that contains all these elements. The type graph and its items are modeled as instances which we represent by using singleton extensions of signatures.

```
one sig access_Point, chain_point extends Node{}
...
```

```
one sig bike_typegraph extends Graph{}
fact Bike_Vocabulary {
  bike_typegraph.n= access_point + chain_point
  bike_typegraph.he = bike + station + bikestation
  bike_typegraph.l = access + left + right
  bike.conn =  access -> access_point
  ...
}
pred topology_linear[g: Graph, t: Tau]{ no e:g.he | e in e.^next ... }
```

For instance, the node `access_point` that stands for an access point is declared as a singleton (`one`) signature extending `extends` signature `Node`. The type graph depicted in Figure 2 is defined by a *fact*, i.e. a predicate assumed to hold in every considered model. For instance, the type graph definition states that the set of nodes of the type graph is the union (denoted by $+$) of `access_point` and `chain_point`, the signatures that stand for the access and the chain point nodes. Further properties of the style are defined via predicates. For instance, `topology_linear` is used to make sure that stations form an acyclic connected chain. The predicate make use of a negative form of quantification (`no`) and transitive closure ^of the relation of right-neighborhood `next`. We see the first line of the predicate below where we express that no edge `e` of a graph `g` belongs to the set of edges is reachable from `e` by moving to the rightward adjacent edge.

4.3 Programmed Dynamism with Alloy

We have defined a set of rewrite rules that state the possible ways in which a software architecture configuration may change. Each rule is defined as a partial, injective graph morphism $p : L \rightarrow R$, where L and R are graphs, called the *left-* and *right-hand* side. Give a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation [EHK+97]. A signature `Prod` implements productions as composed of graphs `lhs` and `rhs`, standing for the left- and right-hand side graphs. A reconfiguration is implemented using two distinct predicates (i.e. `rwStepPre` and `rwStepPost`) that are used to verify conditions that must hold in target and the destination graph. For instance, below we see the code of `RwStepPre` that makes use of auxiliar predicates to check the well formedness of graph `G1` being rewritten, the left- and right-hand side graphs of production `R` and `R` itself. In addition, it checks whether there is a match of the left-hand side in `G1`.

```
pred rwStepPre[G1:Graph, R: Prod, style: Graph, t1:Tau, t2:Tau, t3: Tau, M1: Fun, P:Fun]{
  isTG[G1,style,t1]
  isTG[R.lhs,style,t2]
  isTG[R.rhs,style,t3]
  isProd[R,style,t2,t3,P]
  isMatch[R.lhs,G1,style,t2,t1,M1] }
```

An example of a production is shown in Figure 3, it specifies the migration of a bike from one station to the right station in a chain. Textually, we declare the signature of the production as a singleton extension of `Prod` and define facts that characterize the left- and right-hand side graphs:

```
one sig migrate_right extends Prod{}
```

```
fact migrate_right_lhs{
  migrate_right.lhs.n = cp1 + cp2 + cp3 + ap1 + ap2
  migrate_right.lhs.he = s1 + s2 + b1
  migrate_right.lhs.l = l1 + l2 + l3 + l4 + l5 + l6 + l7 }
```

4.4 Verification using DynAlloy

In the Alloy language, the assertions are used to check specifications. Using the Alloy analyzer, it is possible to validate assertions, by searching for possible (finite) counterexamples for them, under the constraints imposed in the specification of the system. DynAlloy, instead, has proposed the use of *actions* to model state change. An action is how it transforms the system state after its execution and moreover actions can be sequentially composed, iterated or composed by nondeterministic choice [FGLA05]. We want to specify that a given property P is invariant under sequences of applications of some operations. In our case this operation is the rewriting step that from an initial graph G and a Production P generates a new graph G'. A technique useful for stating the invariance of a property P consists of specifying that P holds in the initial Graph, and that for every non initial graph and every rewriting operation, the following holds: P(G) and $rwStep(G,G') \rightarrow P(G')$. For this objective we have defined a set o properties that each SA 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`

```
pred Property1 [tgg: TGG]{
all g: tgg.graphs |all e1: g.he
   |Type[e1,Bike] => one l1: g.l, n1:g.n
   |(Type[n1,Access_Point] and Type[l1,Access]) and
   e1.conn = l1->n1
}
```

2. **Property 2**: The system can not have bikes disconnected and each bike has at most one connection.

```
pred Property2[tgg:TGG]{
   all g: tgg.graphs |
   all e1: g.he |
   Type[e1,Bike] =>  #(e1.conn)=1
}
```

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 connected [e1: HE, e2:HE]{
  univ.(e1.conn) = univ.(e2.conn)
}
pred Property3[tgg:TGG]{
  all g:tgg.graphs|
      all e1:g.he  |
      Type[e1,Bike]=> one e2:g.he |
      Type[e2,Station] &&  connected [e1, e2]
}
```

In order to have programmed dynamism and check that at each reconfiguration step a property P is valid, we proceed as follows. For example we want to provide that in each SA configuration *"if one bike is connected to an access point then must exist a unique station that is connected to the same access point"*. This means to provide the predicate `Property3`. For this, we have described in DynAlloy the action `rwStep` that from an initial configuration `G1` and a production `Pr` execute a single rewriting step, generating a new set of typed graphs `tgg'` with the new SA configuration `G2`. The corresponding DYNALLOY specification is:

```
act rwStep [tgg: TGG, Pr: Prod]{
   pre { rwStepPre[G1, Pr, style_bike, t1, t2, t3, M1, P]}
   post { rwStepPost[G1,G2,Pr,style_bike,t1,t2,t3,t4,P,M1,
   M2,F,tgg,tgg']}
}
```

Moreover, by using partial correctness statements on the set of regular programs generated by the set of atomic actions `rwStep[tgg,Pr1],rwStep[tgg,Pr2]`, we can assert the invariance of the `Property3` under finite applications of functions `rwStep` in a simple way, as follows:

```
assert Property3InvAssertion {
  assertCorrectness[tgg:TGG]{
      pre={Property3[tgg]}
      program={(rwStep[tgg,Pr1] + rwStep[tgg,Pr2])*}
      post={Property3[tgg']}
}}
```

The DynAlloy translator allows us to produce an ALLOY model ready to be analyzed using the ALLOY ANALYZER. In order to do so, we need to provide a fixed length to bound the closure operator. In this example, if we set the bound to "n", we will analyze the following finite counterpart:

```
skip+(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])+
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2]);
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])+
...
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2]);...;
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])
/ ------------- n-times ----------------------/
```

4.5 Example of Analysis

In this section we show two examples of the analysis that we have performed using the ALLOY ANALYZER. *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 labels) we can generate the set of possible software architectures composed of a precise number of components, ports and attachments.

```
module MODEL-FINDING
...
one sig G1 extends Graph{}
fact{
   G1.he=b1+s1+bs1
   G1.n=cp1+cp2+cp3+ap1
   G1.l=a1+a2+l1+r1+l2+r2}
pred show[]{}
run show for 1 Graph, 3 Edge, 4 Node, 6 Label
```

When we run the predicate `show` the ALLOY ANALYZER generates two different SA configurations that are showed in figure 4. The difference between them is the attachments of the bikestation component (i.e., `bs1` in figure 4). In the first case it is attached in the left-side of the station, in the second in the right-side.

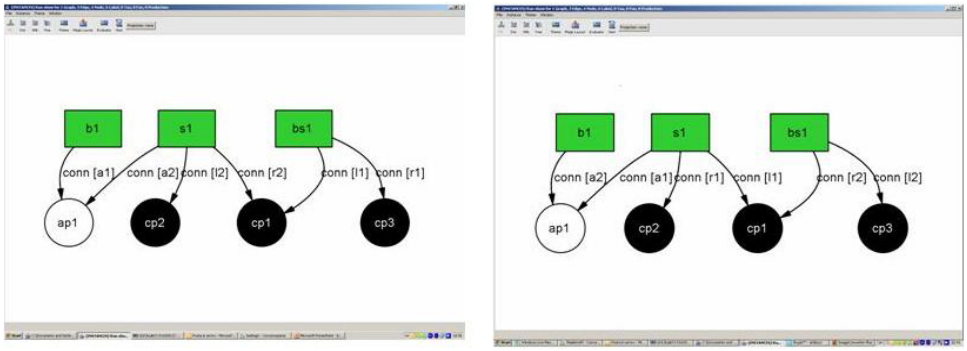

Figure 4: Software Architecture Configurations.

The second kind of analysis that we performed is called *Invariant Analysis*; its the objective to check if a property `P` is invariant under sequences of applications of reconfigurations. In other words we have executed the analysis described in 4.4. To do this we have chosen the *property3* defined before and one initial configuration generated from the previous analysis. We have defined three different modules, `PRODUCTIONS` defines the set of admissible SA reconfiguration (i.e., *Connect_Bike*, *Disconnect_bike*, etc.), `PROPERTIES` defines the set of possible properties that we want to check and `EXECUTION` is the responsible of the invariant analysis. In the last module we have defined the rewriting step action and the *assert* that we execute. The code below enables us to automatically elicit a reconfiguration trace taking the system from one configuration presented in Fig. 4 to the configuration presented in Fig. 5. Moreover, target SA configuration satisfies `Property3`.

```
module EXECUTION
....
assert Property3InvAssertion {
   assertCorrectness[tgg:TGG]
   {
      pre={Property3[tgg]}
      program={(rwStep[tgg,Disconnect_Bike])*}
      post={Property3[tgg']}
}}

check Property3InvAssertion
```

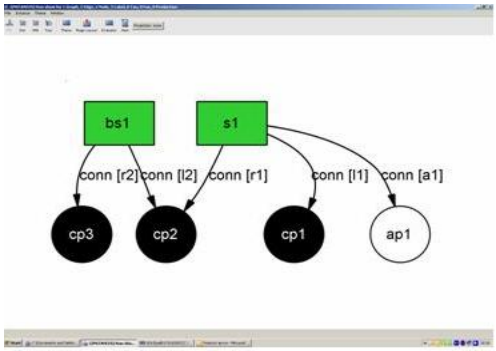

Figure 5: Reconfiguration Result.

5 Related Work

Many research works are focusing on Dynamic Software Architecture specification and validation, they can be sub-divided into three categories. The first, uses formal techniques [BCDW04] such as graph based techniques [Mét98, HIM99, BHTV04, TGM98, WLF01], logic based [End94, AM02] and process algebra based [MK96, ADG, CPT99a]. We use graph grammars and this representation is borrowed from Le Métayer approach [Mét98]. Our scope of this paper are the programmed DSA but different kind of dynamisms as Self-repairing, Ad-Hoc and Constructible have been proposed in [ADG, GS02, GMK02, SG02, Ore96, BCDW04]. The main difference w.r.t. our work is that they are aimed at providing real specification languages while we are aimed at giving an abstract representation of programmed DSA.

The second uses Architectural Description Languages (ADLs) to model adaptive SA with connection and disconnection of components and connectors [OGT+99, MT00, MK96, CPT99b, ADG] by textual or a graphical notation and using also some specific tools to verify them. On one hand, some of the mentioned ADL like Darwin [MK96], and Dynamic Wright [ADG] models DSA and verifies the models with formal techniques. However, these latter did not offer any graphical tools to display these models. On the other hand, some others such as ACME [GMW97], AADL[1],DAOP-ADL[2] and Fractal[3] provides graphical tools for modeling DSA but do not give mechanisms for validating these models. The third focuses on UML architecture modeling extending UML metamodels [KMJ+05, KKJD06] to ensure dynamic architecture and also providing UML diagram translation into formal notation in order to be analyzed[BA05]. An other analysis formal technique that we are thinking to use is Model-Checking. More research works have been done in this direction in the last few years. For example Heckel at al. in [Hec98] proposed an approach where graphs transformations systems are verified using model checking where graphs are interpreted as states and transformation rules as transitions. Rensink in [Ren03] presents a model-checking approach (i.e., GROOVE), based on a graph-specific model checking algorithm, for object-oriented systems. Our idea is to integrate the "*Model Finding*" aspect of Alloy with Model-Checking in order to have a complete analysis framework for DSA.

1 http://www.aadl.info/
2 http://caosd.lcc.uma.es/CAM-DAOP/DAOP-ADL.htm
3 http://fractal.objectweb.org/

6 Conclusion and Future Work

We have presented an approach to verify Dynamic Software Architectures modeled using Typed Graph Grammars. We have considered the programmed dynamicity of a SA 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 verification consists of representing this formalism in the ALLOY relational language and since that this kind of dynamicity requires the analysis of situations in which architectural state changes at run-time, we have considered the use of DYNALLOY language, an extension of ALLOY with actions, in order to represent state changes via actions and programs. This work represents our first step towards analysis of DSA. Three important future directions are needed. The first is to extend the approach, proving properties associated to each kind of DSA formalized in [BBGM07]. The second is to consider the verification of behavioral properties, using model-checking, of SAs that change not only the structure at run-time. Finally, in order to have an automatic and extensible framework to model and analyze DSA we are developing an Eclipse-based framework named ARMADA (**A**utomated **R**e**M**orphing **A**mbient for **D**ynamic **A**rchitectures) with the objective to facilitate DSA modeling and analysis within the software development life cycle.

Acknowledgements: Authors thank Hernán Melgratti for his valuable insights about this problem. This work has been partly supported by EU within the FET-GC2 IST-2005-16004 Integrated Project SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB Project TOCAI.IT.

Bibliography

[ADG] R. Allen, R. Douence, D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of FASE'98*.

[AM02] N. Aguirre, T. Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *ASE '02*. P. 271. IEEE Computer Society, Washington, DC, USA, 2002.

[BA05] B. Bordbar, K. Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In Guimaraes and Isaeas (eds.), *IADIS AC*. Pp. 209–216. IADIS, 2005.

[BBGM07] R. Bruni, A. Bucchiarone, S. Gnesi, H. Melgratti. Modelling Dynamic Software Architectures using Typed Graph Grammars. In *GTVC'07*. 2007. To appear.

[BCDW04] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS*. Pp. 28–33. 2004.

[BHTV04] L. Baresi, R. Heckel, S. Thöne, D. Varró. Style-Based Refinement of Dynamic Software Architectures. In *WICSA*. Pp. 155–166. 2004.

[BLMT07] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto. Style Based Reconfigurations of Software Architectures. Technical report TR-07-17, Dipartimento di Informatica, Università di Pisa, 2007.

[CPT99a] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Pp. 107–126. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999.

[CPT99b] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *WICSA*. Pp. 107–126. 1999.

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

[End94] M. Endler. A Language for implementing generic dynamic reconfigurations of distributed programs. In *In Proceedings of BSCN 94*. Pp. 175–187. 1994.

[FGLA05] M. Frias, J. Galeotti, C. Lopez Pombo, N. Aguirre. DynAlloy: Upgrading Alloy with Actions. In *in Proceedings of the 27th. ACM-IEEE International Conference on Software Engineering (ICSE) 2005*. Pp. 442–450. ACM Press, 2005.

[GMK02] I. Georgiadis, J. Magee, J. Kramer. Self-Organising software architectures for distributed systems. In *In Proceedings of the 1th Workshop on Self-Healing Systems*. Pp. 33–38. 2002.

[GMW97] D. Garlan, R. T. Monroe, D. Wile. Acme: an architecture description interchange language. In *CASCON*. P. 7. 1997.

[GS02] D. Garlan, B. R. Schmerl. Model-based adaptation for self-healing systems. In *WOSS*. Pp. 27–32. 2002.

[Hec98] R. Heckel. Compositional Verification of Reactive Systems Specified by Graph Transformation. In *FASE*. Pp. 138–153. 1998.

[HIM99] D. Hirsch, P. Inverardi, U. Montanari. Modeling Software Architecutes and Styles with Graph Grammars and Constraint Solving. In *WICSA*. Pp. 127–144. 1999.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology*. 2002.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[KKJD06] M. H. Kacem, A. H. Kacem, M. Jmaiel, K. Drira. Describing dynamic software architectures using an extended UML model. In *SAC*. Pp. 1245–1249. 2006.

[KMJ+05] M. H. Kacem, M. N. Miladi, M. Jmaiel, A. H. Kacem, K. Drira. Towards a UML profile for the description of dynamic software architectures. In *COEA*. Pp. 25–39. 2005.

[Mét98] D. L. Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE TSE* 24(7):521–533, 1998.

[MK96] J. Magee, J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, New York, NY, USA, 1996.

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

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, A. Wolf. An Architecture-based approach to self-adaptive software. In *Intelligent Systems, IEEE*. Volume 14(3), pp. 54–62. 1999.

[Ore96] P. Oreizy. Issues in the runtime modification of software architecture. *Elettronics Letters* UCI-ICS-96-35, 1996.

[Ren03] A. Rensink. The GROOVE simulator: A tool for state space generation. 2003.

[Roz97] Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. In Rozenberg (ed.), *Handbook of Graph Grammars*. World Scientific, 1997.

[Sen] Sensoria Project. Public Web Site. `http://sensoria.fast.de/`.

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

[SG02] B. R. Schmerl, D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE*. Pp. 241–248. 2002.

[TGM98] G. Taentzer, M. Goedicke, T. Meyer. Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. In *TAGT*. Pp. 179–193. 1998.

[WLF01] M. Wermelinger, A. Lopes, J. L. Fiadeiro. A graph based architectural (Re)configuration language. In *ESEC / SIGSOFT FSE*. Pp. 21–32. 2001.