Graph-Based Design and Analysis of Dynamic Software Architectures*

Roberto Bruni[1], Antonio Bucchiarone[2,3], Stefania Gnesi[3], Dan Hirsch[4], Alberto Lluch Lafuente[1]

[1] Department of Computer Science, University of Pisa
{bruni,lafuente}@di.unipi.it
[2] IMT Alti Studi Lucca, Italy.
[3] ISTI-CNR, Pisa, Italy.
{antonio.bucchiarone,stefania.gnesi}@isti.cnr.it
[4] Argentina Software Development Center (Intel), Argentina
dan.hirsch@intel.com

Dedicated to Ugo Montanari in occasion of his 65th birthday

Abstract. We illustrate two ways to address the specification, modelling and analysis of dynamic software architectures using: i) ordinary typed graph transformation techniques implemented in Alloy; ii) a process algebraic presentation of graph transformation implemented in Maude. The two approaches are compared by showing how different aspects can be tackled, including representation issues, modelling phases, property specification and analysis.

1 Introduction

It is about 30 years ago when Ugo Montanari started to promote the use of graphs and graph grammars as a multifaceted, unifying framework for the specification, modelling and analysis of concurrent and distributed systems [12,16,17].

Since then, the way in which software artifacts are conceived and their engineering practices have evolved, but Ugo Montanari has always been able to extend the foundations of the graph-based approach along innovative ideas so to adapt it to emergent computational paradigms and software development techniques.

In particular, Ugo Montanari is collaborating within the EU funded project SENSORIA [38] to the development of a novel, comprehensive approach to the engineering of software systems for service-oriented computing. Within SENSORIA, many efforts are devoted to the proposal and validation of sound architectural design [14,39] principles, that focus on *architectural styles* governing the overall

* Research partially supported by the EU within the FETPI Global Computing, project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB Project TOCAI.IT.

structure of software systems in terms of components, their logical interrelationship and their spatial distribution. Architectural styles establish the rationale for certain classes of architectures, e.g. patterns that should be fulfilled also in the presence of reconfigurations and that can even trigger and drive efficient reconfigurations.

There are many typical questions that arise during the design and analysis of software architectures. How do we represent architectures? How do we formalise architectural styles? How do we construct style conformant architectures? How do we model software architecture reconfigurations? How do we ensure style consistency? How do we express and verify architectural properties?

In this paper we compare two graph-based approaches to the specification and modelling of architectural designs. Both approaches are aimed to validate prototypical applications before their realisation and deployment.

The first approach is inspired by the tradition of algebraic approaches to graph transformation [19], an area of research in which Ugo Montanari has played a relevant role. The approach follows [5] and models dynamic software architectures using typed graph grammars (TGG).

The second approach is a recent proposal called *Architectural Design Rewriting* (ADR) [6,7,8]. ADR has been conceived by combining and reconciling apparently different formalisms, very much in the spirit that pervade many other Ugo Montanari's contributions. In particular, ADR takes inspiration of research lines in which Ugo Montanari has been deeply involved like graph-grammar based approaches to architectural styles [26], graphical models of process algebras [21] and rewriting formalisms [23].

We show the feasibility and effectiveness of TGG and ADR by implementing them using high-performant, state-of-the art formal tools. The implementation of TGG is based on Alloy [27] a light-weight approach to the modelling and analysis of software models that is raising the interest of leading researchers in the area of software architectures (see e.g. [32]). The implementation of ADR is based on Maude [13], a high-performance tool implementing Rewriting Logic [36]. We refer to the two implementations as TGG_A and ADR_M.

The main aspects on which we focus are concerned with:

Architectural Representation, i.e. convenient ways to represent a software architecture, to build it, to browse it;
Architectural Styles, i.e. convenient ways to constrain architectures under consideration to satisfy certain requirements;
Architectural Properties, i.e. convenient logical formalisms to express relevant architectural properties;
Architectural Analysis, i.e. efficient techniques and tools for verification.

We show how to tackle these aspects with both approaches. The outcome of our experience suggests that TGG is better 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. Instead, ADR is more convenient for a more advanced phase,

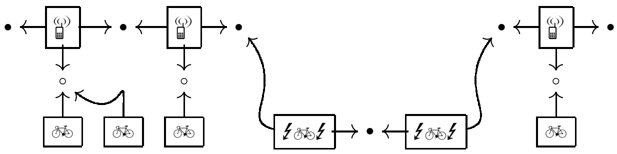

Fig. 1. The road assistance scenario.

where the style is well established and structured, thus enabling flexible and powerful reconfigurations and efficient analysis via model checking.

This paper is structured as follows. Section 2 presents our running example. Section 3 offers a brief background on our design and analysis setting. Section 4 and 5 describe TGG_A and ADR_M, respectively. Section 6 compares both approaches with a special focus on analysis issues. Section 7 gives some pointers to related works. Section 8 concludes the paper and suggests future research avenues. Along with the paper we present flashes of code. For a complete vision we refer to [6,18].

2 Running Example: Road Assistance Scenario

We use as running example a simple bike scenario (see [8]), an ecological variant of the automotive case study of Sensoria. A road assistance service platform is supported by a wireless network of ad hoc stations that are situated along a road. Bikes are equipped with electronic devices that can access services as they move along the road, e.g. to request assistance in case of breakdowns. The graph in Figure 1 depicts a simple architecture of such a system. Each bike ( ) is connected to the service access point ($\circ$) 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 ($\bullet$). 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.

The style of our example is the set of all architectures made of a connected chain of stations (or bikes in station mode) to which any bike is attached. This is a simple but very comprehensive example since it mixes features of well-known architectural styles. Indeed, the style underlying bikes and stations is basically a client-server architecture, while chains of cells resemble pipes-and-filters architectures.

3 Design and Analysis of Dynamic Software Architectures

The architecture of a software system basically consists 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. An architecture is *dynamic* if it can change during run-time. 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 redimensioning software systems.

3.1 The Design of Software Architectures

The first aspect to consider in the design of a software architecture is the formalism used to describe it. Various formalisms exist ranging from the more theoretical graph-based approaches to implementation-oriented architectural programming languages (APLs) such as Java/A [28], passing through architectural description languages (ADLs) [35].

When designing an architecture, it is desirable to consider the concept of an *architectural style* [39], i.e. some set of rules or patterns indicating which components can be part of the architecture and how they can be legally interconnected. An architectural style can also be seen as a (typically infinite) set of *valid* architectures. Typical architectural styles include client-server, pipes-and-filters, layered, multi-tier and peer-to-peer.

There are basically two approaches to the definition of an architectural style. The first approach consists in defining a grammar that allows to produce all the instances of the style. This is the approach first proposed in [37] and subsequently followed in [26], a piece of work which constitutes the main inspiration of ADR. The second approach defines a set of architectural constraints that forbid or require some structural properties. Various approaches, for instance [32], use the Alloy language [27] to specify such constraints.

In this paper we adopt graphs as a suitable formalism to describe software architectures. The use of graphs as both the specification and the execution model combines the user-friendly visual representation with the formal theory for graph rewriting. One of the advantages in the combined use relies on the fact that architectural information can be encoded itself as part of the graphs, allowing for the uniform modelling of dynamic aspects such as computation, discovery, binding and reconfiguration, as graph transformations. Particularly, each 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 the respective tentacles to the same node. Through the paper, we shall omit the prefix 'hyper' for simplicity. Ordinary directed graphs are a particular instance of hypergraph where each edge has two tentacles.

3.2 The Analysis of Architectural Properties

We consider mechanisms to express and verify the properties that we expect to be satisfied by software architectures.

Structural properties. Structural properties regard the topology of the architecture, i.e. the way components are interconnected. Note that the style definition can be considered as a special structural property. In the running example, we could consider properties regarding the number of bikes present in a cell or the effective existence of a path from the leftmost cell to the rightmost one.

Behavioural properties. Behavioural properties regard the dynamism of the architecture, i.e. the state space given by an initial architecture and its possible reconfigurations. In the running example, such properties might be the absence of deadlocks or the fact that a migration is always possible.

Among the class of properties explained above we shall focus and emphasise those that regard architectural styles. For instance, *style conformance* is a structural property that requires an architecture to be an instance of a style while *style preservation* requires all reconfigurations to preserve the style, i.e. that any reconfiguration of a style-conformant architecture results in a style-conformant architecture.

We shall consider the following analysis techniques:

Model finding. We 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. We consider the problem of verifying that a given architecture satisfies some structural or behavioural property expressed in a suitable logic.

Style matching. We consider the problem of determining whether an architecture is conformant to a certain style or whether a reconfiguration is style preserving.

4 Typed Graph Grammars with Alloy

The approach described in this section follows what discussed in [5] and it is based on the modelling of dynamic software architectures using typed graph grammars. The tool that supports this approach is Alloy [27]. Alloy 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 styles, graph transformation rules and architectural properties. 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 negative case. We will show how to use these capabilities to ensure style-consistency, perform model-finding and validate architectural properties.

4.1 Designing Software Architectures

Architectures. We model software architectures using typed graphs.The three basic concepts in the implementation of graphs are *nodes*, *labels* and *edges* that we represent as Alloy signatures:

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

According to the above definition, nodes and labels are atomic concepts, but edges have a relation (field) `conn` that maps each label to nodes. The multiplicity keyword `lone` in its declaration indicates that each label is mapped to at most one node. The previous signatures are marked `abstract`, which means that they have no element except those belonging to its signatures. Indeed we use subsignatures of `Node`, `Edge` and `Label` to represent concrete graph elements.

The signature `Graph` is used to define a graph as a structure made of nodes, edges and labels.

```
abstract sig Graph
{
  he: set Edge,
  n: set Node,
  l: set Label
}
```

In order to forbid ill-formed graphs we have defined some constraints, requiring for instance that the edges connect to nodes in the same graph. In Alloy the constraints of a model are organised into `paragraphs`. Constraints that are assumed always to hold are recorded as `facts`. In the following code we present some facts that we have defined to ensure a clear and efficient presentation of graphs, like requiring different graphs not to share items.

```
// two Edges  must have different set of labels (ports)
all e1,e2:Edge | e1!=e2 => (first[e1.conn]&first[e2.conn]) = none

// two Graphs have different node sets
all g1,g2:Graph | g1!=g2 => #(g1.n & g2.n)=0

// two Graphs have different edge sets
all g1,g2:Graph | g1!=g2 => #(g1.he & g2.he)=0

// two Graphs have different label sets
all g1,g2:Graph | g1!=g2 => #(g1.l & g2.l)=0
```

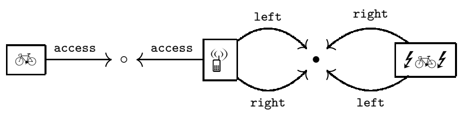

Fig. 2. Type Graph T of the running example

Architectural Styles. We represent an architectural style as a set of basic elements (modelled by a type graph plus a set of invariant constraints indicating how these elements can be legally connected). We define an Alloy module called STYLE that contains all these elements. It is subdivided in two parts, the first part defines basic elements of the style and the second one defines useful constraints.

Below we see the definition of each basic element using singleton extensions of Node, Label and Edge signatures. Note that some of them include facts in their body. For instance, a Bike component must have only one connection using an Access label to the Access_Point node. These definitions are enough to represent the type graph.

```
// Bike-Style basic elements

abstract sig Access_Point, Chain_Point extends Node{}
abstract sig Access extends Label{}
abstract sig Left extends Label{}
abstract sig Right extends Label{}
abstract sig Bike extends Edge{}
{
  #conn=1 and
  conn.univ in Access and // Projection of first column in conn
  univ.conn in Access_Point // Projection of second column in conn
}

abstract sig Bikestation extends Edge{}
{
  #conn=2 and
  conn.univ in Left+Right and
  univ.conn in Chain_Point
}

abstract sig Station extends Edge{}
{
  #conn=3 and
  conn.univ in Left+Right+Access and
  univ.conn in Chain_Point+Access_Point
}
```

The graphical representation in form of a type graph is depicted in Figure 2. Basically, a type graph offers a suitable way to type the items of a graph in a consistent manner. The idea is that the typing of a graph G over a type graph T is modelled by a total graph morphism from G to T.

The code below presents an excerpt of the additional facts needed to define our architectural style.

```
fact Style_constraints
{
  ...
  // if two stations are connected, they share one unique node
```

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

Fig. 3. SPO-based graph rewriting diagram.

```
    all disj s1,s2: Station |
    attached[s1,s2]=>#(last[s1.conn]&last[s2.conn]) = 1
    // each Chain_Point node has at most two and at least one edge connected
    all cp: Chain_Point | #(conn.cp)>0 and #(conn.cp)<=2
    ...
}
```

In order get an instance of the style, generated from the previous code, we use a dummy predicate `instance` with empty parameters and body. A predicate in Alloy is a constraint used to characterise some models of interest. Command `run` is used to find an instance of a predicate. For instance, we ask for a graph to be generated with a scope of at most two edges, one node and two labels as follows:

```
run instance for 1 Graph, 3 Station, 2 Bikestation, 4 Bike, ...
```

When we run the code above the Alloy Analyzer generates an instance and we can ask for a different one. One such instance is depicted in Figure 1 with our graphical notation.

Dynamism. The reconfiguration of a software architecture is described by a set of rewriting productions that state the possible ways in which an architecture 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. Given a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation approach [19] (see Fig. 3). Operationally, the rewrite is applied by finding a suitable match m (i.e. an occurrence of L in G) and the result is the graph obtained from G by removing that instance of L and releasing a fresh instance of R. Moreover, there can be items shared by L and R that are required to trigger the rewrite, but are just preserved by the transformation (some sort of interface, needed to properly attach the fresh copy of R to the existing items in G).

In order to implement what introduced above in the Alloy module called `TGG` we have defined various signatures and predicates that are used to execute a reconfiguration applying the SPO-based approach. First of all we have defined an abstract signature called `Fun` that will be used to define partial and total morphisms, matchings and productions. This signature has three fields that represent nodes, edges and labels functions mapping.

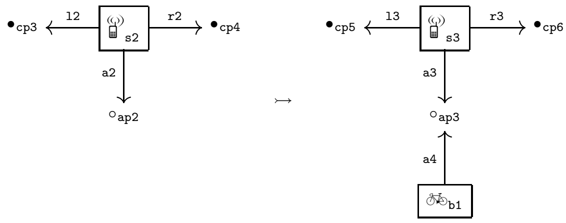

Fig. 4. Reconfiguration rule that connects a bike to a station.

```
abstract sig Fun
{
  fN: set Node -> set Node,
  fE: set Edge -> set Edge,
  fL: set Label -> set Label
}
```

This signature can be used, for example, to define a partial morphism among two graphs. Moreover, in order to verify if a partial morphism among two graphs exists, we have defined the `isPartialMorphism` predicate that takes in input two graphs (i.e. `G` and `H`), the respective typing functions (i.e. `t1` and `t2`) and a mapping function (i.e. `f`). The predicate tries to find a partial morphism from `G` to `H`, executing the set of constraints defined in its body.

```
pred isPartialMorphism [G: Graph, H: Graph, f: Fun, t1, t2: Tau]
{
 ...
 first[f.fE] = G.he
 first[f.fN] = G.n
 first[f.fL] = G.l
 all e1:G.he | all n1: G.n |
              all l1: G.l | (l1->n1) in e1.conn =>
                            f.fL[l1]->f.fN[n1] in f.fE[e1].conn &&
                            t1.tauL[l1]=t2.tauL[f.fL[l1]]
 ...
}
```

A `production` is defined using the signature `Production` that consists of a left- and right-hand side graphs and the morphism indicating the items being preserved.

```
abstract sig Production
{
  lhs: Graph,
  rhs: Graph,
  p: Fun
}
```

Using Alloy, we declare the signature of the previous production as a singleton extension of `Prod` and define facts that characterise the left- and right-hand side graphs. The rule that specifies the connection of a bike to a station is shown in Figure 4 and it is defined by the code below (where the obvious definition of `rhs1` is omitted for simplicity).

```
one sig lhs1 extends Graph{}
{
  he = s2
  n = cp3 + cp4 + ap2
  l = l2 + r2 + a2
  s2.conn = l2->cp3 + r2->cp4 + a2->ap2
}

one sig rhs1 extends Graph{} { ... }

one sig p1 extends Fun{}
{
  p.fN = cp3->cp5 + cp4->cp6 + ap2->ap3
  p.fE = s2->S3
  p.fL = l1->l3 + r2->r3 + a2->a3
}

one sig connect_bike extends Production{}
{
  lhs = lhs1, rhs = rhs1, p = p1
}
```

A single reconfiguration step is implemented using two distinct predicates (i.e., `rwStepPre` and `rwStepPost`. They are used to verify conditions that must hold in the host and target graph. The predicate `rwStepPre` checks the well formedness of the production `Pr` and the validity of matching `m1` of the left-side of the production (i.e., `Pr.lhs`) in the host graph `G`.

```
pred rwStepPre[G1:Graph, Pr: Production, m1: Fun, t1:Tau, t2:Tau, t3:Tau, t4: Tau ]
{
  // t1 defines types for G1, t2 defines typed for Pr.lhs
  isProd[Pr,Pr.p, t2,t3]
  // a production rule is applicable to a graph G1 if there is a matching of lhs into G1
  isMatch[Pr.lhs,G1,M1,t2,t1]
}
```

`rwStepPost` is responsible to execute a single SPO-based rewriting approach generating two morphisms `m2:Pr.rhs` $\rightarrow$ `G2` and `r2:G1` $\rightarrow$ `G2` and the target graph `G2`.

```
pred rwStepPost[G1:Graph, G2:Graph, Pr: Production, m1:Fun, m2:Fun, r1:Fun,
             r2:Fun,t1:Tau, t2:Tau,t3:Tau,t4:Tau]
{
  // m1 : L->G1
  isMatch[Pr.lhs,G1,m1,t2,t1]
  //m2: R->G2
  isTotalTGM[Pr.rhs,G2,m2]
  // r2: G1->G2
  isPartialMorphism[G1,G2,r2]
}
```

4.2 Analysis in Alloy

Structural Properties. In the Alloy language, assertions are constraints that should follow from the facts and must be checked. 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. It is hence possible 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` generated 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 rewriting operation, the following holds: `P(G)` and `rwStep(G,G')` $\rightarrow$ `P(G')`.

For this objective we have defined a set of properties that each architecture, after a rewriting step must satisfy. This set is memorised in an Alloy module called `PROPERTIES`. Structural properties are specified in Alloy defining functions and logical predicates. For instance, we have defined a predicate to express the property that there exists an acyclic path formed by stations that connects the left- and right-most stations. Functions `leftmost` and `rightmost` in the code below identify the left- and right-most stations of an architecture. In order to check this property we use the transitive closure of relation `next`.

```
fun leftmost[g:Graph,t:Tau]:one Edge {let e={e:g.he | t.tauE[e]=S and none= e.~next}| e }

pred Property1 [G:Graph,t:Tau]{ rightmost[G,t] in leftmost[G,t].^next }
```

Behavioural properties. Behavioural properties such as style preservation are suitably specified using DynAlloy [22], an extension of the Alloy language with syntactic sugar to define `actions` as a model of state changes. An action is the means by which the Analyzer transforms the system state after its execution. Regular expressions over actions and predicates can then be combined to express to express complex behavioural properties [22]. This issue is ongoing work [9].

5 Architectural Design Rewriting with Maude

The approach presented in this section is based on ADR [8], a formal framework for the development and reconfiguration of software architectures based on term-rewriting. An architectural style in ADR consists of a set of architectural elements and operations called *design productions* which define the well-formed compositions of architectures. Roughly, a term built out of such ingredients constitutes the proof that a design was constructed according to the style, and the value of the term, called *design*, is the constructed software architecture.

The tool support for the approach is based on Maude [13]. Maude naturally supports most of the features of ADR and also provides a powerful set of tools in the same framework including a model checker and a theorem prover.

5.1 Designing Software Architectures with Maude

Architectures. Software architectures and their constituents are represented uniformly in ADR by suitable graphs that we call *designs*. One difference with the TGG_A approach is that designs add an interface to the graph representing the architecture. The interface of a design is represented by an edge. The internal structure (called *body*) of a design is the architecture graph.

We have implemented various modules named `GRAPH-*` with the necessary machinery to construct graphs. Designs are implemented in a functional module

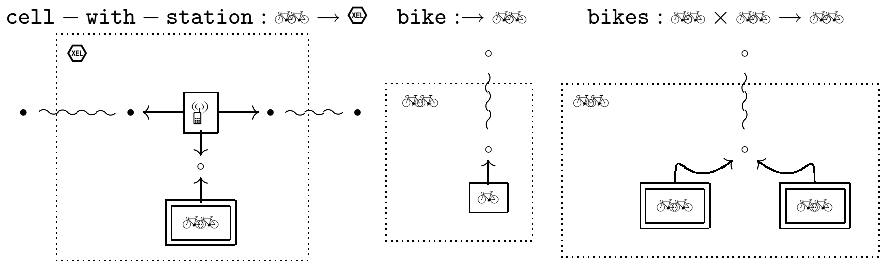

Fig. 5. Some design productions.

called DESIGN, which includes the definition of the sort Design, a constructor operation design and an operation to replace a *hole* in a design (representing an unspecified part of the architecture) by a design whose interface is compatible with the hole.

Below we include an excerpt of module DESIGN. First, we see the declaration of sort Design. In a next line, the type of operation design is defined: it builds a design from a typed graph (here implemented as graph morphisms) representing the interface, a typed graph representing the body, a mapping relating the nodes in the interface with those in the body, and a list of edges representing holes. Membership equations not shown here take care of the well formedness of the design. Last, the type of operation apply is defined: it takes two arguments of type Design (the first intended to have a hole and the second being the design with proper interface to be placed in the hole) and returns the Design after the replacement. This complex operation basically implements the concept of type-consistent hyper-edge replacement [24].

```
sort Design .
op design : GraphMorphism GraphMorphism Map{Node,Node} List{Edge} -> Design .
op apply : Design Design -> Design .
```

Architectural Styles. The principle of ADR consists in defining an architectural style as a suitable algebra over designs. This approach can be seen as an algebraic recasting of context-free graph grammar-based approaches to architectural styles [37]. So, while in TGGs a style is given by logical predicates that forbid illegal architectures, in ADR the style is given by a generative mechanism that allows us to construct legal architectures only.

In Maude we define an architectural style as a set of sorts (the architectural types) which are subsorts of Design, a set of operations, called *design productions*, on such sorts (the legal ways of composing designs) and an interpretation of terms as designs.

Below we show an excerpt of module BIKE-STYLE which implements the architectural style in an abstract manner (i.e. no interpretation over designs is yet defined). Sorts Cells and Bikes are the types of the whole system (a chain of cells) and the type of bikes allocated in the cells, respectively. We see constructors nocell and bikestation, respectively representing an empty cell and a cell

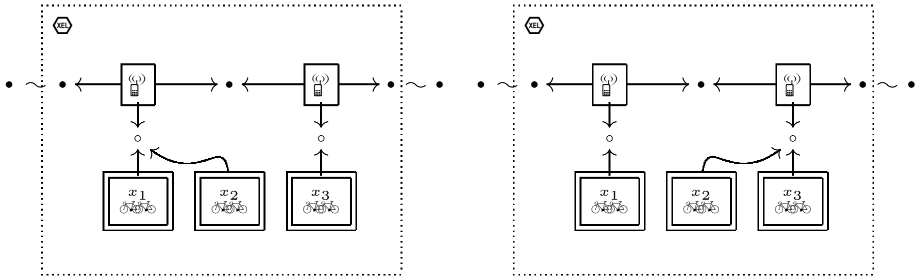

Fig. 6. Rule migrate − right.

formed by one bike in station mode. Operation `cell-with-station` allows to construct from a collection of bikes (the inclusion of a fresh station is embedded in the operation), while `chain` builds a chain of cells by concatenating two chain of cells. Observe that one can annotate operations with axioms such as the associativity of `chain` or the fact that it has `nocell` as identity. Some of these productions are graphically represented in Fig. 5. Roughly, double boxes correspond to the parameters, the dotted boxes enclose the result of the productions, with the corresponding codomain type in its left corner, while waved lines indicate how internal nodes are exposed in the interface.

```
sort Cells . *** Chain of cells
sort Bikes . *** Collection of bikes
...
op nobike : -> Bikes .
op bike : -> Bike.
op bikes : Bikes Bikes -> Bikes [assoc comm id: nobike] .
op nocell : -> Cells .
op bikestation : -> Cells .
op cell-with-station : Bikes -> Cells .
op chain : Cells Cells -> Cells [assoc id: nocell] .
```

Dynamism. Contrary to TGG, reconfiguration rules in ADR are defined as rewrite rules over design terms, instead of over plain graphs or designs. In addition, ADR rules can be conditional, labelled and inductively defined and can be used to define complex behaviours and reconfigurations.

Below we see rule `migrate-right` (see Fig. 6). Note that because `bikes` is associative, commutative and has `nobike` as identity element, the rule can be matched to perform the migration of any number of bikes, while in TGG_A we have a rule able to migrate one bike at each step, only.

```
rl [migrate-right] : chain(cell-with-station(bikes(x1,x2)),cell-with-station(x3))
                  => chain(cell-with-station(x1),cell-with-station(bikes(x2,x3))) .
```

The module also includes the more complex reconfiguration that deals with shutdown by repairing the connection with an ad-hoc chain of bikes in station mode. Rule `bike2cell` allows a bike to reconfigure itself into a bike in station mode using label `'tocell`. Rule `bikes2cell` allows to propagate such reconfigurations inside collections of bikes. Finally, rule `cell2chain` allows reconfigure a station whenever the corresponding bikes are reconfigured into a chain of cells.

```
rl [bike2cell] : bike  => {'tocell}bikestation .

crl [bikes2cell] : bikes(x1,x2) => {'tocell}chain(y1,y2)
 if x1 =/= nobike /\ x2 =/= nobike /\ x1 => {'tocell}y1 /\ x2 => {'tocell}y2 .

crl [cell2chain] : cell-with-station(x1) => y1
 if x1 => {'tocell}y1 .
```

5.2 Analysis with Maude

The property specification mechanisms of our approach is based on the use of suitable logics to reason about structural and dynamic aspects of software architectures.

Structural Properties. Recall that contrary to TGG_A, ADR_M considers two aspects of the structural information of a software architecture: the design term, which can be seen as an abstraction, a proof of style conformance or an encoding of the construction process, and the design itself. We use different logics to reason about each aspect.

A natural and structured way to reason about design terms is the use of a suitable spatial logic (e.g. [11,34]). Basically, for each design production `f` used to compose designs the logic incorporates a spatial operator `f-so` to decompose a design. For instance, formula `chain-so(phi1,phi2)` is satisfied by all those designs of the form `chain(x1,x2)`, where design `x1` satisfies formula `phi1` and design `x2` satisfies formula `phi2`.

As an example consider the property that states a collection of bikes has at least n bikes. We see that this property can be inductively defined as below. For n equal to zero the formula always holds. For $n+1$ (we use the successor constructor `s` below) the formula holds whenever the term is decomposable as the composition via operation `bikes` of one bike (`bike-so`) and a term with at least n bikes.

```
op at-least-k-bikes : Nat -> Prop .
vars n : Nat .
eq at-least-k-bikes(0) = True .
eq at-least-k-bikes( (s n )) = bikes-so(bike-so,at-least-k-bikes(n)) .
```

Our implementation also includes Courcelle's Monadic Second-Order (MSO) logic [15]. The rough idea is that the logic allows us to quantify over sets of nodes and edges. This allows us to reason about the complex interpreted structure of graphs that is hidden at the level of design terms.

Behavioural Properties. The dynamic aspects are expressed using the Linear-time Temporal Logic (LTL) for which Maude provides a module where only the state predicates have to be defined. Roughly, one is able to reason about infinite sequences of reconfigurations, by expressing properties on the ordering of state observations. Such observations are state predicates given by closed structural formulae.

As an example we state that it is always true that a collection of bikes has at least 2 bikes by formula `[] at-least-k-bikes(2)`, where `[]` denotes the *always* temporal operator.

$\mathbf{ADR_M}$	**Aspect**	$\mathbf{TGG_A}$
ADR (SOS, HR) *More flexible*	Formal Support	SPO Graph Rewriting *Well-studied, resource conscious*
Maude *Expressive, simple, performant*	Tool	Alloy *Light-weight, FO/SAT-based*
Hierarchical, Interfaced Graphs *Structured view, reusability*	Architectures	Flat Graphs *Concrete view*
Constructive (Algebra) *Style consistent*	Styles	Declarative (Alloy Logic) *Prototypical*
Term rewrites *Inductive, conditional*	Reconfigurations	Graph rewrites *Item trace*
Ad-hoc, VLRL, MSO *Expressive, flexible*	Structural properties	Alloy Logic *Simple, based on FO*
Ad-hoc, LTL *Expressive, well-studied*	Behavioural properties	Pre/Post, DynAlloy *Regular expressions, ongoing work*
Ad-hoc *Flexible, ongoing work*	Style Matching	Alloy Analyzer *Built-in, based on SAT*
By construction *Must not be proved*	Style Preservation	Alloy Analyzer *Performant*
Search strategies *Built-in, flexible*	Model Finding	Alloy Analyzer *Built-in*
LTL Model Checker *Efficient, flexible*	Model Checking	Alloy Analyzer *Bounded*

Fig. 7. Main aspects and advantages of each approach

6 Comparison

We overview a brief comparison of the presented approaches w.r.t. the issues discussed in Section 3. Figure 7 summarises the main aspects and advantages of each approach.

6.1 Designing Software Architectures.

Architectures. Both approaches represent architectures as suitable graphs. The main differences are that TGG represents architectures by flat graphs while ADR considers additional structural information. First, graphs have an interface which supports the compositionality of architectures. Second, graphs are equipped with a design term which serves various purposes: it is a proof of style consistency, it is a witness of the design process, it can be used to offer a hierarchical view at a suitable level of abstraction.

Architectural Styles. Each approach follows a different tradition. TGG_A uses explicit structural constraints by means of logic predicates. This approach is more suited to a reactive modelling process: the software architect constructs a model and the system reacts reporting style inconsistencies. ADR_M uses an implicit generative mechanism inspired by context-free graph grammars. This approach is more suited to a proactive modelling process: the software architect performs a decision procedure (i.e. refinements and compositions) that is guaranteed to be style consistent and not faulty.

Representing dynamism. Each approach defines dynamism at different levels of abstraction. TGG defines dynamism by means of local rewriting rules on flat graphs, while ADR defines rules on design terms. ADR rules are more abstract and also more expressive, because conditional labelled rules are also allowed. In addition, ADR rules guarantee style preservation by construction. On the other hand, ADR has still no mechanism to keep trace of reconfigured items, while this is done in TGG by the notion of trace morphism.

6.2 Specifying Architectural Properties.

Structural properties. Structural properties are expressed in TGG_A by means of the same formalism used to define architectural styles, namely the Alloy logic. Maude does not offer a built-in structural logic but spatial logics such as the Verification Logic for Rewriting Logic [34] arise naturally and we can adjust the property specification mechanism at will. Indeed, we implemented property specification mechanisms tailored for the abstract view of design terms and the more concrete view of designs.

Behavioural properties. Alloy does not offer a standard behavioural logic and one has to rely on ad-hoc mechanisms or recent extensions such as DynAlloy. Maude, instead, comes with a built-in implementation of a Linear-time Temporal Logic (LTL) module, for which the user must provide just the state predicates.

6.3 Analysing Software Architectures.

Style Matching. Checking style-consistency in TGG_A is done via the same mechanism as the verification of structural properties, i.e. it amounts to verify whether an architecture satisfies the predicate characterising the architectural style. Instead, in ADR_M we need a parsing mechanism, able to determine for given graph g, whether there is a design term d such that the design corresponding to d has a body graph isomorphic to g. This mechanism is under implementation.

Model Finding. 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 predicates and having a certain number of bikes and stations. 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" [27]. This means that examining small architectures is often enough to detect possible major flaws.

Model finding is not provided by Maude directly. In order to perform model finding with Maude we need basically two mechanisms: one to generate a state space of models and one to explore it. We have defined a rewrite theory that simulates a design-by-refinement process, roughly consisting of the context-free graph grammar obtained by a left-to-right reading the design productions. Another generative mechanism that we can implement produces random graphs by adding items iteratively. The resulting graph could be used as the body of a design. In order to explore such state spaces we can use the LTL model checker, the `search` command or rewriting strategies.

Model Checking. Alloy does not have traditional model checking capabilities but its model finding features can be used instead to encode some (bounded) model checking problems. Basically, models and predicates are translated into a first-order formula and solved with efficient SAT solvers. Instead, model checking in Maude is supported by an efficient built-in LTL model checker. We have also implemented the satisfaction relation for the above mentioned MSO and spatial logics.

An alternative to model checking is the use of manual or computer-assisted theorem proving, which is highly facilitated in ADR by the hierarchical nature of *design productions*, which allows for proof based on structural induction. This issue is currently being investigated.

7 Related Work

An exhaustive enumeration and a deep comparison with related work is out of the scope of this paper. We mention, however some of the works that have inspired or share concepts with our approaches.

Many research works are focusing on the design and analysis of dynamic software architectures, applying formal methods such as graphs [37,25,3], logics [20,1] and process algebras [2,10].

As a matter of fact ADR has taken inspiration from initiatives that promote the conciliation of software architectures and process calculi by means of graphical methods [33].TGG instead, follows the tradition of graph grammars and applied to software architectures [37] and combines them with logical approach of Alloy.

Our approaches also shares concepts with various approaches ranging from process calculi that deal with reconfigurable component based architectures (e.g. [1]) to graphical representation of concurrent systems such as those based on Synchronized Hyperedge Replacement [21] or Bigraphs [29].

Maude and Alloy have been already used for the design and analysis of software architectures or graph transformation systems. For instance, in [30] Maude is used to model and verify software architectures given in LfP, a system description language with hierarchical behaviour. On the other hand, the work presented in [4] proposes a methodology to analyse transformation systems by means of Alloy and its supporting tool. They encode graph transformation systems into Alloy and verify properties such as reachability of given configurations through a finite sequence of steps or whether given sequences of rules can be obtained on an initial graph, and show all the configurations that can be obtained by applying a bounded sequence of rule instances on a given initial graph.

Another source of related work regard Architectural Description Languages (ADLs) to model and analyse software architectures [35] or UML-based approaches to model dynamic architectures [31].

8 Conclusion

In our experience, TGG_A and ADR_M are both flexible formal methods for the design and analysis of dynamic software architectures. Moreover, we promote their synergic application as each of them can be focused on the development phases where they are more effective.

In the early phase of the development, the architectural style is typically not well understood and TGG looks more appealing since the software architect can follow an incremental approach, adding, removing or refining architectural constraints with the help of the inconsistencies shown by the Alloy Analyzer.

Later, when the architectural style is well understood and stable, the software architect can define the style in a generative manner in the form of an ADR style. The benefits of an ADR style arise then from its hierarchical and structured nature. First, complex reconfigurations can be inductively defined on the structure of designs. Second, properties can be defined again inductively and at an abstract level. Last, but not least, model checking and theorem proving can be applied to exploit the hierarchical structure.

This separation of phases also matches the choice of the tools. Indeed, TGG could have been easily implemented in Maude, but Maude lacks of a built-in, efficient mechanism to perform model finding. On the other hand, implementing ADR in Alloy would have required a lot more efforts to encode all the mechanisms that are native in Maude such as normal forms, memberships or conditional rewrite rules, and built-in tools such as the LTL model checker.

Current work is devoted to validate our ideas by enriching our experience with more realistic examples, taken from global computing areas like service oriented computing and grid computing. In future work we plan to extend our comparative analysis to further interesting aspects such as the ordinary behaviour of software components, the analysis of architectural styles and run-time implementation. Another interesting perspective is to investigate other ADR suited interpreted algebras, other than graphs. Constraints systems, for instance, seem well suited and appealing to deal with non functional aspects like quality of service.

References

1. N. Aguirre and T. S. E. Maibaum. Hierarchical temporal specifications of dynamically reconfigurable component based systems. *Electr. Notes Theor. Comput. Sci.*, 108:69–81, 2004.
2. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of FASE'98*, volume 1382 of *LNCS*. Springer, 1998.
3. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proceedings of WICSA'04*, pages 155–166. IEEE Computer Society, 2004.
4. L. Baresi and P. Spoletini. On the use of Alloy to analyze graph transformation systems. In *Proceedings of ICGT'06*, volume 4178 of *LNCS*, pages 306–320. Springer, 2006.
5. R. Bruni, A. Bucchiarone, S. Gnesi, and H. Melgratti. Modelling dynamic software architectures using typed graph grammars. In *Proceedings of GTVC'07*, ENTCS. Elsevier, 2007. To appear.
6. R. Bruni, A. Lluch Lafuente, and U. Montanari. Hierarchical design rewriting with Maude. In *Proceedings of WRLA'08*, ENTCS. Elsevier, 2008. To appear.
7. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Service oriented architectural design. In *Proceedings of TGC'07*, volume 4912 of *LNCS*, pages 186–203. Springer, 2007.
8. 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.
9. A. Bucchiarone and J. P. Galeotti. Dynamic software architectures verification using DynAlloy. In *Proceedings of GT-VMT'08*, 2008. To Appear.
10. C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Proceedings of WICSA'99*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.
11. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Proceedings of ICALP'02*, volume 2380 of *LNCS*, pages 597–610. Springer, 2002.
12. I. Castellani and U. Montanari. Graph grammars for distributed systems. In *Graph-Grammars and Their Application to Computer Science*, volume 153 of *LNCS*, pages 20–38. Springer, 1982.
13. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
14. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
15. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 313–400. World Scientific, 1997.
16. P. Degano and U. Montanari. Concurrent histories: A basis for observing distributed systems. *J. Comput. Syst. Sci.*, 34(2/3):422–461, 1987.
17. P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, 1987.
18. Dynamic Software Architectures for Global Computing Systems. `http://fmt.isti.cnr/~antonio`.
19. 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. World Scientific, 1997.
20. M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of BSCN'94*, pages 175–187, 1994.
21. G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In *Proceedings of FMCO'05*, volume 4111 of *LNCS*, pages 22–43. Springer, 2006.
22. M. Frias, J. Galeotti, C. Lopez Pombo, and N. Aguirre. DynAlloy: Upgrading Alloy with actions. In *Proceedings of ICSE'05*, pages 442–450. ACM Press, 2005.
23. F. Gadducci and U. Montanari. Comparing logics for rewriting: rewriting logic, action calculi and tile logic. *Theor. Comput. Sci.*, 285(2):319–358, 2002.
24. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
25. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architecutes and styles with graph grammars and constraint solving. In *WICSA'99*, volume 140 of *IFIP Conference Proceedings*. Kluwer, 1999.
26. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *Electronic Notes on Theoretical Computer Science*, 109:97–109, 2004.
27. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.
28. Java/A. `http://www.pst.ifi.lmu.de/Research/current-projects/java-a/`.
29. O. H. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.
30. C. Jerad, K. Barkaoui, and A. Grissa-Touzi. Hierarchical verification in Maude of LfP software architectures. In *Proceedings of ECSA'07*, volume 4758 of *LNCS*, pages 156–170. Springer, 2007.
31. M. H. Kacem, A. H. Kacem, M. Jmaiel, and K. Drira. Describing dynamic software architectures using an extended UML model. In *Proceedings of SAC'06*, pages 1245–1249. ACM, 2006.
32. J. S. Kim and D. Garlan. Analyzing architectural styles with Alloy. In *Proceedings of ROSATEA'06*, pages 70–80. ACM Press, 2006.
33. B. König, U. Montanari, and P. Gardner, editors. *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems, 6.-11. June 2004*, volume 04241 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005. `http://www.dagstuhl.de/04241/`.
34. N. Martí-Oliet, I. Pita, J. L. Fiadeiro, J. Meseguer, and T. S. E. Maibaum. A verification logic for rewriting logic. *J. Log. Comput.*, 15(3):317–352, 2005.
35. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
36. J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3):213–237, 2007.
37. D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.
38. Sensoria Project. `http://sensoria.fast.de/`.
39. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.