

# Graph-Based Design and Analysis of Dynamic Software Architectures<sup>\*</sup>

Roberto Bruni<sup>1</sup>, Antonio Bucchiarone<sup>2,3</sup>, Stefania Gnesi<sup>3</sup>, Dan Hirsch<sup>4</sup>,  
Alberto Lluch Lafuente<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of Pisa

{bruni,lafuente}@di.unipi.it

<sup>2</sup> IMT Alti Studi Lucca, Italy.

<sup>3</sup> ISTI-CNR, Pisa, Italy.

{antonio.bucchiarone,stefania.gnesi}@isti.cnr.it

<sup>4</sup> Argentina Software Development Center (Intel), Argentina

dan.hirsch@intel.com

**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 are tackled, including representation, modelling process, property specification and analysis.

## 1 Introduction

It is about 25 years ago, when Ugo Montanari started to promote the use of graphs as a multifaceted, unifying framework for the specification, modelling and analysis of concurrent and distributed systems [8, 12, 13].

Since then the way in which software artifacts are conceived and their engineering practises 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, the last few years have witnessed the growth of *service-oriented computing* (SOC), where the basic computational entities are autonomous and can be assembled dynamically by complex applications that span over heterogeneous platforms. For these reasons, SOC is a paradigm that poses serious challenges in terms of scalability, dynamicity and open-endedness for the reuse of known methodologies. Thus, formal models are needed to guide the otherwise error-prone development of SOC applications. Ugo Montanari is collaborating within the EU funded project SENSORIA [28] to the development of a novel, comprehensive approach to the engineering of software systems for service-oriented computing.

---

<sup>\*</sup> 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 TOCAL.IT.

Within SENSORIA, many efforts are devoted to the proposal and validation of sound architectural design [10, 29] principles, that focus on *architectural styles* governing the overall 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.

In this paper we compare two graph-based approaches to the specification and modelling of architectural designs of SOC systems. 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 [15], an area of research in which Ugo Montanari has played a relevant role. The approach follows [2] and models dynamic software architectures using typed graph grammars (TGG).

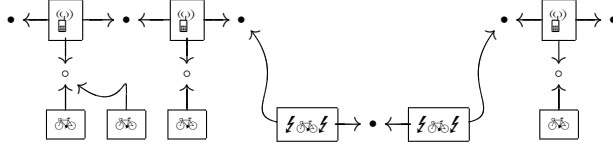
The second approach is a recent proposal coauthored by Ugo Montanari called *Architectural Design Rewriting* (ADR) [3–5]. ADR has been conceived by combining and reconciling apparently different formalism, very much in the spirit that pervade many of Ugo Montanari’s contributions. In particular, ADR takes inspiration of research lines in which Ugo Montanari has been deeply involved like graph-grammar based approaches to architectural styles [20], graphical models of process algebras [16] and rewriting formalisms [18].

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 [22] a formal light-weight approach to the modelling and analysis of software models that is raising the interest of leading researchers in the are of software architectures (see e.g. [24]). Instead, the implementation of ADR is based on Maude [9], a high-performance tool implementing Rewriting Logic [26]. We refer to the implementations as  $TGG_A$  and  $ADR_M$ .

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 configuration graphs, allowing for the uniform modelling of dynamic aspects such as computation, discovery, binding and reconfiguration, as graph transformations.

The main aspects on which we focus are concerned with:

- Architectural Representation*, i.e. convenient ways to represent the graph, to build it, to browse it;
- Architectural Styles*, i.e. convenient ways to constrain the graphs under consideration to satisfy certain requirements;
- Architectural Properties*, i.e. convenient formalisms (e.g. logics) to express relevant graph properties;
- Architectural Analysis*, i.e. efficient techniques and tools for the verification of properties.



**Fig. 1.** The road assistance scenario.

Through the paper we shall rely on *hypergraphs*, where a single (hyper)edge can be attached to one, two or many nodes, but will omit the prefix ‘hyper’ for simplicity. Ordinary directed graphs are a particular instance of hypergraphs.

We use as running example a simple scenario (see [5]) inspired by 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 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 ( $\text{♻}$ ) is connected to the service access point ( $\circ$ ) of a station ( $\text{Ⓜ}$ ) 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 ( $\text{♻}$ ), in which they become standalone stations.

There are many typical questions that arise during the design and analysis of the example. How do we represent architectural configurations? How do we formalise the above informally described architectural style? How do we construct style conformant architectures? How do we model reconfigurations? How do we ensure style consistency? How do we express and verify properties such as requiring chain of stations not to be broken?

We show how to tackle this questions with both approaches. The outcome of our experience suggests that the  $\text{TGG}_A$  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,  $\text{ADR}_M$  is more convenient for an advanced phase, 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 offers a quick background on our design and analysis setting. Section 3 and 4 describes  $\text{TGG}_A$  and  $\text{ADR}_M$ , respectively. Section 5 compares both approaches with a special focus on analysis

issues. Section 6 concludes the paper and suggests future research avenues. Along with the paper we present flashes of code. For a complete vision of the code the reader is referred to [3, 14], but we remark that the code reported in this paper has been adapted for the sake of readability.

## 2 Design and Analysis of Dynamic Software Architectures

We overview our vision of the design, specification and analysis of *dynamic software architectures*.

### 2.1 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. In our running example, for instance, components include bikes and stations.

An architecture is *dynamic* if it can change during run-time. Typical such 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. In our running example, reconfigurations include the migration of bikes from one cell to another, or repairing actions triggered by stations shutting down.

### 2.2 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 ArchJava [1] or Java/A [23], passing through architectural description languages (ADLs) [25]. Both our approaches model software architectures as suitable graphs. In Figure 1, for instance, we see how bikes and stations are represented as hyperedges (boxes) whose tentacles (outgoing arrows) are attached to nodes (circles).

*Architectural Styles.* When designing an architecture, it is desirable to consider the concept of an *architectural style* [29], 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 (possibly infinite) set of *valid* architectures. Typical architectural styles include client-server and pipelines.

The style of our example has been informally described in the Introduction above. In few words, it 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 very interesting example since it mixes features of well-known architectural

styles. Indeed, style underlying bikes and stations is basically a client-server architecture, while chains of cells resemble pipelines.

There are basically two approaches to the definition of an architectural style. The first approach consists on defining a grammar that allows to produce all the instances of the style. This is the approach first proposed in [27] and subsequently followed in [20], 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 [24], use Alloy [21] to specify such constraints.

### 2.3 The Specification of Architectural Properties

We shall consider mechanisms to express the properties that we expect from 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 our example, we shall 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 the initial configuration and its possible reconfigurations. In our example such properties might be the absence of deadlocks or the fact that a migration is always possible.

*Stylistic properties.* 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.

### 2.4 The Analysis of Software Architectures

We shall consider various analysis problems.

*Model finding.* We consider the problem of analysing the state space of all possible architectures. Such analysis can serve both as a computer-aided design process as a debugging method to find out inconsistencies in the model or in its specification

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

*Model checking.* We consider the problem of verifying that a given architecture satisfies its properties.

### 3 Typed Graph Grammars with Alloy

The approach described in this section follows [2] and models dynamic software architectures using typed graph grammars (TGG). The implementation of the approach is based on Alloy [22]. Alloy provides a description language, based on signatures and relations, to represent software models that we have found very suited to model graphs. Alloy also provides a logic, based on an extension of first-order logic with relational operators, to represent properties or constraints of the 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 and uses efficient SAT solvers to decide satisfiability and provide a counterexample in negative case. We use these capabilities to ensure style-consistency, perform model-finding activities and validate architectural properties.

#### 3.1 Designing Software Architectures

*Architectures.* Each software architecture is represented by a graph where components (or connectors) are modelled 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 `->` indicates a binary relation. The keyword `lone` preceding `Node` constrains `conn` to relate each label to at most one element in `Node`. Thus, for each element of signature `Edge`, `conn` is partial function (i.e. the tentacle function) mapping labels to nodes. The signature `Graph` is used to define 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 }
```

*Architectural Styles.* We represent architectural styles by means of a type graph and a set of constraints. The type graph describes the types of components, connectors, ports and roles and their allowed *local* connections. A configuration compliant to a type graph  $T$  is described by the notion of a *T-typed hypergraph*. Figure 2 depicts the type graph of our running example. The signature `Tau` will be used to model graph morphisms and hence the typing of the architectural

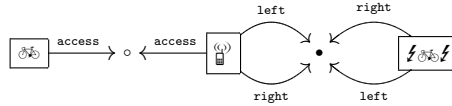


Fig. 2. Type Graph  $T$  of the running example

elements. 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]{...}
```

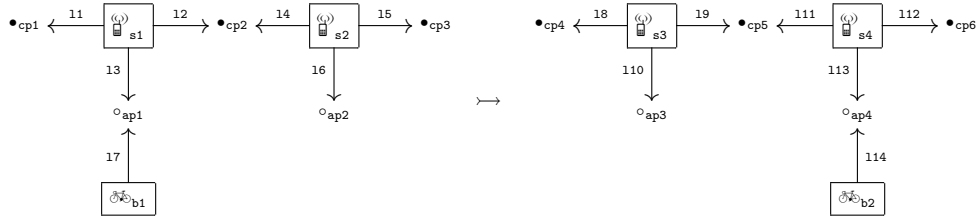
Structural constraints can also be used to impose style restrictions on the way the items are locally or *globally* composed in a configuration. We define an Alloy module called `BIKE-STYLE` that contains all these elements. The type graph and its items are modelled as instances which we represent by using singleton extensions of signatures.

```
one sig access_point, chain_point extends Node{}
one sig bike, station, bikestation extends Edge{}
one sig access, left, right extends Label{}
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
  station.conn = left -> chain_point + right -> chain_point + access -> access_point
  bikestation.conn = left -> chain_point + right -> chain_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 definition is given 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 makes use of a negative form of quantification (`no`) and transitive closure (`^`) of the relation of right-neighbourhood `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



**Fig. 3.** Reconfiguration rule that migrates a bike to the rightward station.

edge. The use of predicates instead of constraints or facts is due to the need of considering configurations that temporarily violate some style constraints, e.g. a repairing reconfiguration might need to perform various steps where the configuration violates the architectural style before the desired, style conformant configuration is reached.

*Dynamism.* 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 [15]. 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 auxiliary 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 characterise 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 = 11 + 12 + 13 + 14 + 15 + 16 + 17 }
```

### 3.2 Specifying Architectural Properties

*Structural 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 a configuration. 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 [17], 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 [17]. This issue is under current work [6].

## 4 Architectural Design Rewriting with Maude

The approach presented in this section is based on ADR [5], 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 [9]. 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.

### 4.1 Designing Software Architectures with Maude

*Architectures.* Software configurations 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 architectural configuration. The interface of a design is represented by an edge. The internal structure (called *body*) of a design is the configuration graph.

We have implemented various modules named **GRAPH-\*** with the necessary machinery to construct graphs. Designs are implemented in a functional module 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 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 [19].

```

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

```

*Architectural Styles.* The principle of ADR consists of 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 [27]. 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 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.

```

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 inductive and can be used to define complex behaviours and reconfigurations.

Below we see how we implement rule `migrate-right`. 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 saw a rule able to migrate one bike at each step, only.

```
r1 [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 `'to cell`. 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.

```
r1 [bike2cell] : bike => {'to cell}bikestation .

cr1 [bikes2cell] : bikes(x1,x2) => {'to cell}chain(y1,y2)
  if x1 /= nobike /\ x2 /= nobike /\ x1 => {'to cell}y1 /\ x2 => {'to cell}y2 .

cr1 [cell2chain] : cell-with-station(x1) => y1
  if x1 => {'to cell}y1 .
```

## 4.2 Specifying Architectural Properties 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.

The natural way to reason about design terms is the use of a suitable spatial logic (e.g. [7]). Basically, for each design production used to compose designs the logic incorporates a spatial operator 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 [11]. 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  $\Box$  `at-least-k-bikes(2)`, where  $\Box$  denotes the *always* temporal operator.

*Stylistic Properties.* The philosophy of ADR is that architectural constraints of a style are implicitly modelled by the corresponding *design productions*. Style-consistency and preservation are given by construction and, contrary to  $TGG_A$ , need not be checked.

## 5 Comparison and Analysing Software Architectures

We overview a brief comparison of the presented approaches w.r.t. the issues discussed in Section 2.

### 5.1 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.

## 5.2 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 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.

## 5.3 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 configurations: 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” [22]. This means that examining small architectures is often enough to detect possible 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.

## 6 Conclusion

In our experience,  $TGG_A$  and  $ADR_M$  are both valid formal methods for the design and analysis of dynamic software architectures. Moreover, we promote their combined 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 implement 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. 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 algebra, in addition to graph. Constraints, for instance, seem well suited and appealing.

## References

1. Archjava. <http://archjava.fluid.cs.cmu.edu/>.
2. R. Bruni, A. Bucchiarone, S. Gnesi, and H. Melgratti. Modelling dynamic software architectures using typed graph grammars. In *Proceedings of the International Workshop on Graph Transformation for Verification and Concurrency (GTVC'07)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2007. To appear.
3. R. Bruni, A. Lluch Lafuente, and U. Montanari. Hierarchical design rewriting with maude. In *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, Electronic Notes in Computer Science. Elsevier, To appear.
4. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Service oriented architectural design. In *Proceedings of the 3rd International Symposium on Trustworthy Global Computing (TGC'07)*, LNCS. Springer, 2007. To appear.
5. 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.
6. A. Bucchiarone and J. P. Galeotti. Dynamic Software Architectures Verification using DynAlloy. In *Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08)*, To Appear.
7. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer and F. Trigueiro Ruiz *et alii*, editors, *Automata, Languages and Programming*, volume 2380 of LNCS, pages 597–610. Springer, 2002.
8. I. Castellani and U. Montanari. Graph grammars for distributed systems. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 1982.
9. 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 *Lecture Notes in Computer Science*. Springer Verlag, 2007.
10. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
11. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 313–400. World Scientific, 1997.
12. P. Degano and U. Montanari. Concurrent histories: A basis for observing distributed systems. *J. Comput. Syst. Sci.*, 34(2/3):422–461, 1987.
13. P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, 1987.
14. Dynamic Software Architectures for Global Computing Systems. <http://fmt.isti.cnr/~antonio>.
15. 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 G. Rozenberg, editor, *Handbook of Graph Grammars*, pages 247–312. World Scientific, 1997.
16. G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of LNCS, pages 22–43. Springer, 2005.

17. M. Frias, J. Galeotti, C. Lopez Pombo, and N. Aguirre. Dynalloy: Upgrading alloy with actions. In *in Proceedings of the 27th. ACM-IEEE International Conference on Software Engineering (ICSE) 2005*, pages 442–450. ACM Press, 2005.
18. F. Gadducci and U. Montanari. Comparing logics for rewriting: rewriting logic, action calculi and tile logic. *Theor. Comput. Sci.*, 285(2):319–358, 2002.
19. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
20. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *Electronic Notes on Theoretical Computer Science*, 109:97–109, 2004.
21. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002.
22. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.
23. Java/A. <http://www.pst.ifi.lmu.de/Research/current-projects/java-a/java-a/>.
24. J. S. Kim and D. Garlan. Analyzing architectural styles with Alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM Press.
25. 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.
26. J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3):213–237, 2007.
27. D. L. Métyayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.
28. Sensoria Project. Public web site. <http://sensoria.fast.de/>.
29. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.