Università di Pisa
Dipartimento di Informatica

Technical Report: TR-07-17

Style-Based Architectural Reconfigurations

R. Bruni[1], A. Lluch Lafuente[1], U. Montanari[1], E. Tuosto[2]

July 31, 2007

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Style-Based Architectural Reconfigurations*

R. Bruni[1], A. Lluch Lafuente[1], U. Montanari[1], E. Tuosto[2]

[1] Department of Computer Science, University of Pisa
{bruni,lafuente,ugo}@di.unipi.it
[2] Department of Computer Science, University of Leicester
et52@mcs.le.ac.uk

Abstract. We introduce *Architectural Design Rewriting* (ADR), an approach to deal with the design of reconfigurable software architectures. The key features we promote are: (i) rule-based approach (over graphs); (ii) hierarchical design; (iii) algebraic presentation; and (iv) inductively-defined reconfigurations. Architectures are suitably modeled by graphs whose edges and nodes respectively represent components and connection ports. Architectures are designed hierarchically by a set of edge replacement rules that fix the architectural style. Depending on their reading, productions allow: (i) top-down design by refinement, (ii) bottom-up typing of actual architectures, and (iii) well-formed composition of architectures. The key idea is to encode style proofs as terms and to exploit such information at run-time for guiding reconfigurations. The main advantages of ADR are that: (i) instead of reasoning on flat architectures, ADR specifications provide a convenient hierarchical structure, by exploiting the architectural classes introduced by the style, (ii) complex reconfiguration schemes can be defined inductively, and (iii) style-preservation is guaranteed.

1 Introduction

Autonomous and adaptive systems challenge software engineering to deal with issues like scalability, dynamicity and openness at the right level of abstraction. In this context, software architectures can help in the specification, formalization and manipulation of such systems by restricting and disciplining the admissible shapes and patterns to be considered.

The architecture of a software system basically consists of the structure of components and the way they are interconnected. The configuration of a system consists of the present components and interconnections (i.e. the architecture), together with their current state. Ordinary computation can change the state, but not the architecture. Changes in the configuration that compromise the system might trigger a proper reaction, which is called *reconfiguration*, aimed to re-establish a non-compromised configuration by restructuring the architecture.

* Research supported by the EU within the FET-GC II Project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB Project TOCAI.IT.

When designing an architecture, it is desirable to consider the concept of *architectural style* [17], i.e. some set of rules indicating which components can be part of the architecture and how they can be legally interconnected. Typical architectural styles include client-server and pipelines. Architectural styles can be applied to reuse existing design patterns. More importantly, they offer a further benefit when architectural information is carried over the execution of the system, since: (i) it can be used to better localize and isolate the small parts of the system to be reconfigured, and (ii) one can control whether changes in the system imply changes in the architecture and eventually in its style. While changes in the architecture are acceptable and even necessary, in most cases the architectural style should be preserved by reconfigurations. For instance, in a system with client-server architectural style, clients connecting and disconnecting from the server are permitted, while a client connecting to a client is not.

Contribution. The main contribution of our approach is the proposal of a simple algebraic formalism called *Architectural Design Rewriting* (ADR). It can be used for describing development issues like refinement, abstraction and composition, and run-time issues like execution and reconfiguration. The methodology associated with ADR guarantees that all reconfigurations are style-preserving and can be conveniently formulated inductively on hierarchical designs.

We represent software architectures by suitable graphs, whose edges model components and whose nodes model the ports through which components are connected. We see architectural styles not just as a "flat" typing (e.g. a graph morphism from the actual architecture to the type graph representing the style), but rather as the way to impose some hierarchy over the architecture (e.g. to better parse it and reason on it). In general, it can be the case that, within a fixed style, several different readings are possible for the same actual architecture.

Taking inspiration from the use of process calculi in the modeling of communicating mobile systems, we specify architectural styles in terms of suitable algebras, interpreted over a specific graph-based model. We emphasize here the algebraic reading of both style-based design and reconfiguration rules. Design rules (or productions) are seen as the basic operations for composing well-defined architectures according to their types, yielding well-typed results. Hence, architectural information can be kept at run-time just by recording (possibly an abstraction of) the term associated with the well-typedness proof from the design phase. Style-preserving reconfigurations can then be expressed as ordinary term rewrite rules on such architectural information. The hierarchical nature of ADR allows for the specification of rules that take typed architectures as parameters.

Thanks to the fact that term rewrites can always occur in any larger context, ADR reconfigurations span over the least part of the system involved in the reconfiguration. In case certain local changes in the architecture are subordinated to the corresponding adaptation of the adjacent environment, then we can use conditional reconfiguration rules, expressing that a composed architecture can be rewritten only if its sub-components are suitably transformed first. This step makes the formalism very powerful. For example, complex reconfigurations like the nested wrappings of a hierarchy of components are easy to express and check.

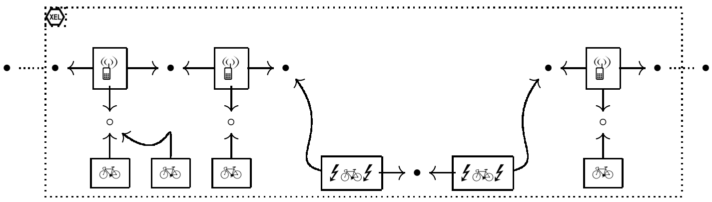

Fig. 1. The road assistance scenario.

Although not discussed here, by pushing further this mechanism, ADR can also represent the normal behaviour of systems. Thus ADR offers a unified setting where design development, runtime execution and reconfiguration could be defined on the same foot.

Running example. We illustrate our approach with a simple scenario, where bikes access wireless services via ad-hoc stations that are situated along a road. Figure 1 depicts a simple configuration of such a system. Each bike () is connected to the service access point (○) of a station () which is possibly shared with other bikes (attached to the same station). A station and its accessing bikes form a *cell* (). Bikes can move away from the range of the station of their current cell and enter the range of another cell. A handover protocol permits bikes to migrate to adjacent cells as in standard cellular networks. Stations, in addition to the service access point, use two other communication ports that we call chaining ports (•). Such ports are used to link cells in larger cell-chains (). Stations can shut down, in which case their *orphan* bikes are connected to other stations. This is tackled by appropriate system reconfigurations. 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.

Structure of the Paper. We give an incremental presentation of the main features of ADR. Section 2 defines our notion of software architectures as *designs*, which are basically graphs with a typed interface. Section 3 explains how to define *architectural styles* by a set of design productions. Section 4 presents the main contribution of our approach, namely, hierarchical style-based reconfigurations via term rewriting. Section 5 concludes the paper and outlines future work.

2 Software Architectures

An architecture is an abstraction or view of the implementation of a system. The basic view describes the present components and their interconnections but other aspects (e.g. behavior) can also be considered, leading to an aspect oriented development. In this paper we focus on structural aspects. This section explains our domain for representing and manipulating software architectures.

2.1 Architectural Models

Graphs offer a natural model for software architectures. Take, for instance, the typical component and connector view of software architectures [2] which is at the base of architectural description languages (ADLs) like ACME [6]. Basically, an architecture is viewed as a collection of interconnected components which represent the main computational entities. The interface of a component is represented by ports where *connectors* are attached. Connectors model the pathways of communication and their interfaces consist of roles. Connectors are attached to components by assigning ports to roles. The corresponding graph has components and connectors as hyperedges whose *tentacles* represent roles and ports. The nodes of the graph are the attach points.

Example 1. The components of our running example appearing in Figure 1 are bikes and stations and are represented as hyperedges. For simplicity connectors are not considered and nodes model interconnections of components, namely two components are connected when their tentacles (outgoing arrows) share a node. For instance, a station is a hyperedge with three tentacles: the leftward and the rightward are attached to chain ports (type $\bullet$) for communicating in each direction along the chain, while the downward tentacle connects the station to bicycles and is thus attached to access ports (type $\circ$).

Of course, other choices like representing a component as a node are possible and have been indeed proposed (e.g. in [14]). Anyway, the suitability of graphs as a modelling formalism for software architectures has become accepted. In addition, because our refinement mechanism is based on hyperedge replacement, we shall tipycally prefer to see complex, refinable architectural entities as edges, rather than nodes. Let us now give a formal definition for graphs.

Definition 1. *A* graph *is tuple $G = \langle V, E, t\rangle$ where V is the set of nodes, E is the set of edges and $t : E \to V^*$ is the tentacle function.*

We leave implicit the order of tentacles exiting from each edge in our figures, since in our example their order is clearly given by the outoging direction of the tentacles (say that the first one is the leftwards tentacle and the rest are ordered clockwise). Otherwise a suitable numbering or naming of tentacles becomes necessary in the figures.

2.2 Architectural Elements

The vocabulary of an architectural style consists of a set of architectural elements. For instance, in a client-server architecture the obvious choice is to take two classes of architectural elements: clients and servers. In our example, we have bikes, stations, cells and ports. A suitable way to represent architectural elements consists of using type graphs, which have one edge and node for each different type of edge and node elements, respectively. One can also have a type hierarchy with sub- and super-types relations as in [1] but we restrict to a one-level typing hierarchy for the sake of simplicity. The relationship between actual

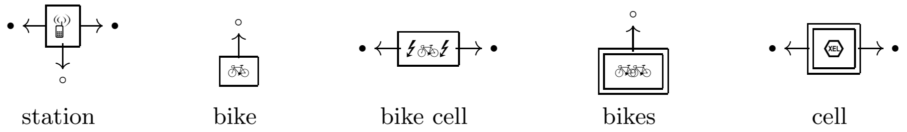

Fig. 2. Architectural elements.

elements of an architecture and their abstract classes is suitably represented by graph morphisms, mapping each instance to its type. In our graphical representation, edge types are explicitly represented as labels inside the corresponding box.

Example 2. The architectural elements of our running example are in Figure 2. The types of edges are , , , and which respectively represent stations, chains of cells, bikes in normal mode, bikes in cell mode and collections of bikes. The nodes have either type $\circ$ (ports where bikes in normal mode and cell stations are attached to), or $\bullet$ (nodes connecting stations). Bikes have one connection port only (of type $\circ$) while stations have three of them (one of type $\bullet$ on both its left and right tentacle and one of type $\circ$ on the bottom one). Now, in our approach we distinguish two kind of architectural components: abstract (or non-terminal, refinable) and concrete (or terminal, basic, non refinable). A doubly-boxed edge represents a non-terminal edge. Non-terminal edges labeled by and are used to represent collections of bikes and chains of stations, respectively. For instance, we shall see that a chain of cells can be refined as a concatenation of chains of cells or as a single cell (cf. Figure 4).

Typed graphs are hence defined as graphs equipped with a typing morphism.

Definition 2. *Let G and H be two graphs. A pair of functions $\langle f_V, f_E \rangle$ where $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ is a* graph morphism *from G to H if f_V and f_E preserves the tentacle functions, i.e. $f_V^* \circ t_G = t_H \circ f_E$, where f_V^* is the homomorphic extension of f_V to V_G^*.*

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

The type graph of our running example is in Figure 3.

Technically, edge types (i.e., edges E_T of the type graph T) are partitioned into several alphabets. A first distinction is due to the tentacle function: E_T is partitioned into families indexed by tuples of node types. Even if there is only one node type, edge types are ranked according to the number of tentacles. An additional distinction is between terminals $\mathcal{T}$ and nonterminals $\mathcal{NT}$. As for ordinary string grammars, (non) terminal edges, namely edges labeled by a (non)

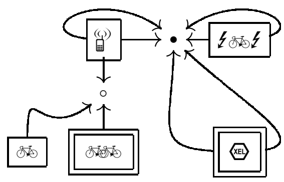

Fig. 3. Type graph.

terminal symbol, represent abstract components which cannot (can) be refined. Thus, terminal edges represent basic components of the architecture.

In our approach a *design* is an assembly of basic components that have a typed interface, labeled by a non-terminal edge. The idea is that the type of the interface edge represents the abstract component class, while its tentacles represent the exposed nodes.

Definition 4. *A* design *is a graph with interface, i.e a triple* $d = \langle L_d, R_d, i_d \rangle$*, where* L_d *is a (typed) graph consisting only of a nonterminal and by distinct nodes attached to its tentacles;* R_d *is a (typed) graph without nonterminal edges; and* $i_d : V_{L_d} \to V_{R_d}$ *is a total function.*

The notion of graph morphism is trivially extended to morphisms between designs. The nodes of R_d in the image of i_d are the interface nodes of the design, ordered according to the tentacles of the nonterminal edge in L_d, while the label L_d gives the type (or role, meaning, etc.) of the design. We shall see that this notion of design is suited for system development by refinement, abstraction or composition. By abuse of notation, we shall use the term *graph* for both a typed graph with interface and its underlying graph. In addition we sometimes say that a graph is of type A meaning that the interface of the graph is an edge of type A.

Example 3. Figure 1 depicts a design of type : the graph inside the dotted box is the architecture. The interface edge is represented by the dotted box itself, whose type is indicated by a dotted box at its upper-left corner. The tentacles are represented by the dotted lines, which also represent the edge morphisms.

3 Architectural Design

Designing a software architecture is a process that might consist of putting together existing designs (composition), describing the internal structure of an abstract component (refinement) or specifying that an assembly of components is actually an abstract component (abstraction). When considering styles, all these operations should be governed by a mechanism consistent with the style of the system. We believe that graph grammars are a suitable formalism to achieve this (see e.g [8] for an early work proposing graph transformations as architectural design formalism).

3.1 Architectural Styles

An architectural style consists of the set of components that can be part of the architecture (the vocabulary), and a set of rules indicating how they can be legally interconnected. In ADR the vocabulary of a style is given by the type graph, while the legal interconnections are defined by productions. They are very much like designs (see Definition 4), but where the underlying graph can have non-terminal edges. Any architecture constructed following a style corresponds to a term built out from such productions: the term represents the way in which the architecture was constructed and its value is the architecture itself.

Definition 5. *A* (design) production *p is a tuple $\langle L_p,\ R_p, i_p, l\rangle$ where L_p is a (typed) graph consisting only of a nonterminal labeled by say A_p and by distinct nodes attached to its tentacles; R_p is a (typed) graph with both terminal and non-terminal edges; $i_p : V_{L_p} \to V_{R_p}$ is a function; and l is a bijection mapping the non-terminal edges of R_p on an initial segment $[1, 2, \ldots, n_p]$ of positive numbers.*

The *type* of a production p is $A_1 \times A_2 \times \ldots \times A_{n_p} \to A_p$, where A_k is the nonterminal symbol labeling the k-th nonterminal edge e_k of R_p, namely with $l(e_k) = k$. The functional type $A_1 \times A_2 \times \ldots \times A_{n_p} \to A_p$ associated to p is not an accident. In fact, p can be considered a function that when applied to a tuple $\langle d_1, d_2, \ldots, d_{n_p}\rangle$ of designs of types $A_1, A_2, \ldots A_{n_p}$, respectively, returns a design $d = p(d_1, d_2, \ldots, d_{n_p})$ of type A_p. The definition is obvious: $d = (L_p, R_d, i_p)$, where R_d is obtained from R_p by replacing edge e_k in it with graph R_{d_k} respecting the tentacle function i_{d_k}, $k = 1, \ldots, n_p$.

This view corresponds to a bottom-up design development: a design is constructed by putting together some component designs. However, the dual view is also possible: a production can be seen as refinement of an abstract component of type A as an assembly of concrete and abstract components, the latter being of type $A_1, A_2, \ldots A_{n_p}$. As a matter of fact, design productions can be seen as hyperedge replacement rules which are the essence of the graph grammar based approaches to architectural styles (e.g. [14] and [8]).

Example 4. The design productions and their corresponding types for our example are illustrated in Figure 4. Let us briefly comment just a couple of them (see also Example 5). Production `chain` takes any two designs of type XEL and returns a design of type XEL. In words, it connects two chain of cells resulting in a chain of cells. The figure can also be interpreted as a refinement rule stating that a chain of cells can be decomposed into two consecutive chains of cells. Production `bikes` is analogous and used to build larger collections of bikes. Note that, unlike in the chain rule, here the order of bikes in the collection is inessential.

The construction above yields a many-sorted algebra $\mathcal{M}$, where sorts are nonterminal symbols $\mathcal{NT}$, values are designs and operations are productions of the corresponding types. Thus, a set of productions $\mathcal{P}$ determines an architectural style, where software architectures of type A are those in the carrier $\mathcal{M}_A$ corresponding to sort A. (Carriers are not necessarily disjoint.) Furthermore, if a term t of type A of $\mathcal{M}$ evaluates to a design d, t itself can be considered a typing proof of d.

Once we have established our algebraic setting, we can import several useful concepts from the standard algebraic machinery. A straightforward construction introduces free typed variables x into terms. As usual, an assignment η of variables to values can be uniquely extended to an evaluation $t[\eta]$ of terms with variables. Furthermore, term substitution $t[t'/x]$ can be defined. The latter construction has an obvious meaning: given a partial design $t(x)$ with a component x to be refined, and a refinement term t' (which can be considered a derived operation) of the right type, the (partial) design $t[t'/x]$ is the result of the refinement step. Conversely, given $t[t'/x]$, the identification of a subsystem t' of $t[t'/x]$, such that t' and $t(x)$ are terms in $\mathcal{M}$, is an abstraction step. Different levels of complexity of the construction depend on terms being linear (one occurrence of each variable) or not. In this paper we consider just the linear case.

Our algebraic approach is very general. For instance, the design process can be seen as the process of writing a term for a design. A refinement design process can be seen as a top-down writing of a term of a given type. A compositional design process can be seen as applying operations to given arguments (bottom-up). A style check, i.e. checking whether an architecture meets a certain style, reduces to writing a term. Note that the same graph can have several different proofs, possibly accommodating even for different types. Each proof provides an admissible justification for the well-typedness of the actual architecture, yielding a way to parse the graph and reason on it accordingly.

Example 5. Consider the productions in Figure 4. Rule `station` builds a chain of cells (consisting of a single cell, obviously) from a collection of bikes. An empty chain of cells is constructed via operation `nocell`. Complex chains are obtained via the concatenation operation `chain`. Clearly, for any x of type XEL, the terms `chain`(x, `nocell`) and x define the same architecture. Similarly, operations `bike`, `nobike` and `bikes` are used to construct (possibly empty) collections of bikes. Finally, a chain of cells (consisting of a single cell, again) can also be constructed by a bike in cell mode (given by the constructor `bikestation`). A term of type XEL defining the architecture in Figure 1 is

```
chain( chain( station(bikes(bike, bike)) , station(bike) )
     , chain( bikestation , chain(bikestation, station(bike)) ) )
```

3.2 Architectural Properties

Indeed the type of an architecture explains how the architecture is or can be constructed, i.e. by means of the operations that have that type as codomain. Such operations implicitly respect some structural constraints or properties, thus types are not merely symbolic labels. However, not all interesting classes of architectures are represented as non terminals. Recall that, as we have seen in the introduction, our flavour of structural constraints is that of a graph grammar

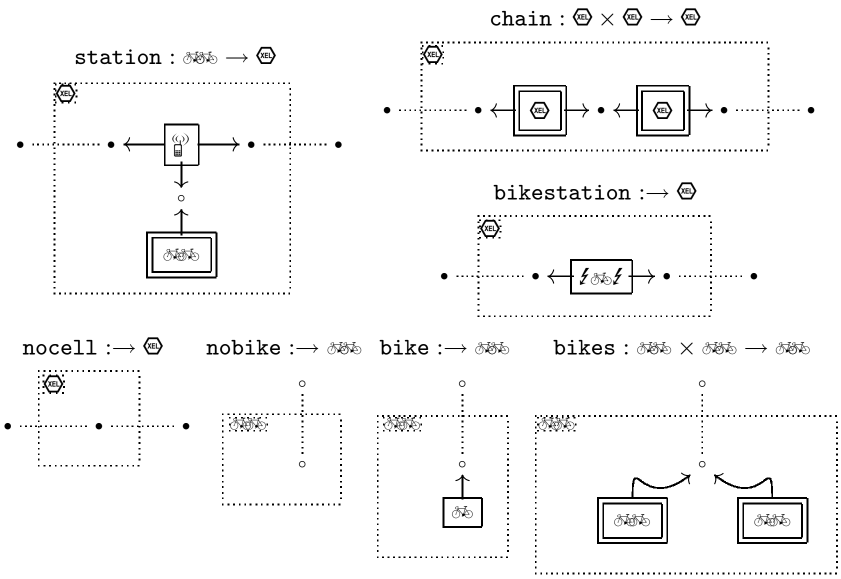

Fig. 4. Design productions.

instead of some logical formalism describing architectures that are or not part of the style. The benefits of the latter approach is that architectural properties are explictly stated, while in our approach they are implicit. For instance, with an explicit mechanism we can formally state that any system of type is indeed a connected chain of cells. In ADR, instead, this is still true by construction even if not explicitly stated. However, the operations of the architectural style offer the formal basis for proving many architectural properties by structural induction.

Example 6. In our running example, consider the property ϕ informally[1] expressed as "the graph of a cell is a unique (possibly empty) path connecting the chain ports of the interface with either stations or bikes in cell mode". By structural induction it is very easy to show (possibly relying on a model checker) that ϕ holds true for any architecture x of type . For instance, if x is `nocell` the proof is trivial since both the left and right chainports are the same. If x is either `station`(y) or `bikestation` the proof is given by the fact that both ports are connected via an edge of type in the first case and in the second one. Finally, consider the case when x is `chain`(y, z) and assume that both y and z (with type) satisfy ϕ. Then the left chain port is connected to the middle node which is in his turn connected to the right chain port. Thus, we conclude that x satisfies ϕ too, because the property ϕ is preserved under concatenation.

[1] Formally, ϕ can be stated in e.g. Courcelle's monadic second-order logic.

4 Architectural Reconfiguration

Architectures are not static but might evolve in different dimensions. We have seen that architectures can be designed at static-time by refining abstract components or assembling subsystem architectures. During run-time instead, architectures might evolve due to actions of normal behavior or reconfigurations.

Components leaving or joining the system can require correcting actions that lead the system into a proper state. Such actions are called *reconfigurations*, and are specially common in systems with dynamic architectures.

For instance, in our example, the system must deal with bikes and stations leaving and joining the system or with collections of bikes that migrate from one cell to another or join and leave the system. We also need reconfigurations to deal with a station shutting down, by migrating the collection of bikes in its cell to the adjacent ones or constructing an ad-hoc chain of bikes in cell mode.

While sometimes a reconfiguration rule can be naively described as a direct manipulation of a design or its corresponding term (without variables), we argue that reconfigurations arise more naturally and in a well-disciplined way at the abstract level of the architecture, i.e. as manipulations of partial designs (terms with variables). Indeed, the reconfigurations informally mentioned above have been written in terms of collections of bikes (abstract class level), rather than single bikes (basic element level). An additional issue that one would like to have in a reconfiguration mechanism is the capacity to give guarantees about the architectural style. For instance, whether it is preserved or not. The rest of the section discusses these issues in deeper detail and presents different reconfiguration mechanisms, starting from basic graph rewrite rules (and their disadvantages), to more sophisticated style-based rules.

4.1 Reconfigurations as graph transformations

Since our architectures are represented by graphs, reconfigurations acquire the taste of graph transformations. An eminent way to formalize graph transformations is the use of graph rewrite rules. This can be done in several flavours [16], but basically the rules come with a left-hand side graph G_L and with a right-hand side graph G_R. Operationally, the rewrite can be applied to any graph G larger than G_L by finding a suitable match (i.e. an occurrence of G_L in G) and the result is the graph obtained from G by removing that instance of G_L and releasing a fresh instance of G_R. Moreover, there can be items shared by G_L and G_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 G_R to the existing items in G).

Note that this view takes the actual architecture as a flat, unstructured graph, thus disregarding the architectural information. Consider, for instance, the problem of dealing with migrating bikes. One could define: i) a graph transformation rule to migrate one bike and then apply this rule a certain number of times to migrate any collection of bikes, or ii) infinitely many rules (one for each natural number) to migrate a collection in one step. Such solutions, however,

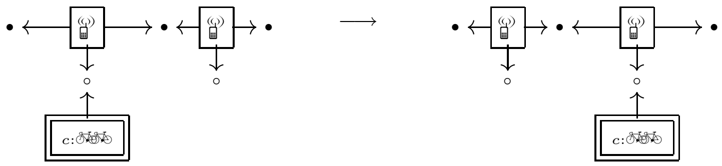

Fig. 5. Graph transformation rule for migrating a collection of bikes (left to right).

are not abstract enough, since they consider concrete numbers of bikes, while a natural abstraction is that of migrating any collection of bikes (disregarding of its actual number).

In other words, it is desirable to define a reconfiguration as a single action operating at the most appropriate abstraction level, i.e. the rewrite rules should involve both class elements and architectural elements, according to the fixed design hierarchy.

In the example, by exploiting the design class 🚲, we can write a unique rule for handling the splitting of arbitrarily large collections of bikes inside a cell. Figure 5 sketches a reconfiguration by means of a schema of graph transformation rules. The reconfiguration is tackled by detaching any collection of bikes c from a cell and attaching it to an adjacent station. Yet, an application of this rule cannot guarantee that all bikes migrate, because it can be applied in larger contexts where other bikes are connected to the leftmost station.

We have already stated that using architectural styles have several benefits. One of such benefits is that they might offer guarantees about the execution of the system. If one wants to preserve such benefits, then styles should be preserved during run-time. A naive way to guarantee style preservation is to check the style of the architecture after the application of a reconfiguration. This solution, however, is not desirable since there is no guarantee that a necessary style-preserving reconfiguration is possible.

Our methodology has style preservation as a must. We need thus a mechanism that ensures style preserving *a priori*, i.e. a mechanism that ensures that any application of a reconfiguration preserves the architectural style.

Now, while we can be convinced that the rule in Figure 5 preserves the style, this is not obvious when observing the rule itself without considering all possible contexts. We shall see that enforcing style preservation will allow us to use a neater algebraic notation for reconfigurations.

4.2 Style-preserving reconfigurations as term rewriting

We have seen that design rules can be given an algebraic formulation in terms of many-sorted operations over a suitable algebra of typed graphs (with interfaces), with terms describing a particular style-proof. Note that in this way it is possible that: (i) the same well-defined architecture can be described by different terms; (ii) the same well-defined architecture can be assigned different classes.

Since style-preserving reconfigurations essentially operate at the level of style-proofs (the abstract elements in L and R can be seen as typed variables), the algebraic view can be pushed further by term rewriting over (style-)proof terms: a graph transformation rule is seen as a rewrite rule $L \to R$, where L and R are $\mathcal{M}$-terms of the same type. Typically, both L and R are linear and all the variables in R appear in L. This is the case when reconfigurations do not add abstract components to the system. These variables can be instantiated in any way consistent with the types, and both R and L can be freely contextualized. Then, it is possible to apply the rule in any larger architecture $t(L\eta)$, where η assigns proof terms to variables and where t is any term with one hole with the same type as L. After the reconfiguration, the architecture $t(R\eta)$ is obtained.

There is a very simple sufficient condition for enforcing style preservation, namely that both the left-hand side L and the right-hand side R of the reconfiguration can be assigned the same proper abstract class. Roughly, this way we are guaranteed that whenever L occurs, R would be also allowed by the style. Moreover, since L and R can themselves contain class elements, we are guaranteed that such elements can be consistently refined by any actual design compliant to those classes, keeping the generality of hierarchical reconfigurations.

For instance, to deal with the simple reconfigurations of bikes joining and leaving the system we just need the two simple rules $\texttt{join}(x_2)$ and `leave` below. (Obviously all terms that appear in the rules are properly typed but we neglect the explicit typing in favour of a cleaner presentation.)

$$\begin{array}{rccc} \texttt{join}(x_2): & x_1 & \longrightarrow & \texttt{bikes}(x_1, x_2) \\ \texttt{leave}: & \texttt{bikes}(x_1, x_2) & \longrightarrow & x_1 \end{array}$$

The rule $\texttt{join}(x_2)$ is used for a collection x_2 of bikes to join a cell. Dually, the rule `leave` is used for a collection of bikes to leave the cell.

We now consider the migration of bikes caused by their mobility. Clearly, the problem can be tackled by a sequence of leave and join reconfigurations. However, it would be better (and sometimes necessary) to perform it in a single step as expressed by the rule `migrater` below for the rightwards migration:

$$\begin{array}{rl} \texttt{migrater}: & \texttt{chain(}\ \texttt{station}(\texttt{bikes}(x_1, x_2)\ ,\ \texttt{station}(x_3))\ \texttt{)} \\ \longrightarrow & \texttt{chain(}\ \texttt{station}(x_1)\ ,\ \texttt{station}(\texttt{bikes}(x_2, x_3))\ \texttt{)} \end{array}$$

The rule `migrater` (together with the similar rule for leftwards migration) is in Figure 6. In comparison with the sketch of Figure 5, `migrater` considers the general context of a cell, where the source cell might contain another stationary collection of bikes and the target cell might already contain a collection of bikes. The key is that now both the left- and the right-hand sides of the rules are typed terms and have type XEL. Moreover, by typability the rule takes into account all the bikes connected to the stations.

Another interesting reconfiguration deals with the problem of a station shutting down. Assuming that the station notifies this situation to its adjacent stations, these react bypassing the connection. Orphan bikes of the cell that is going

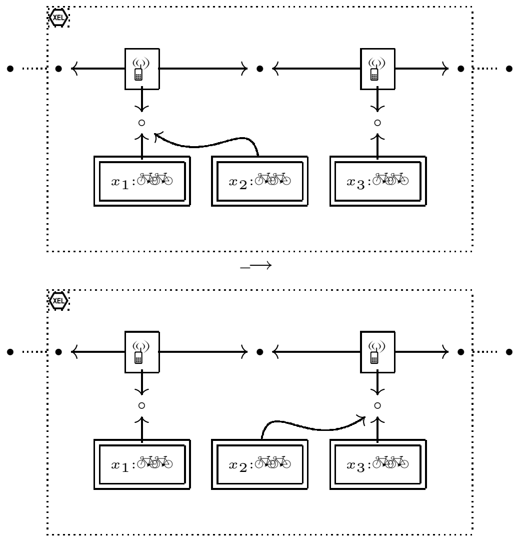

Fig. 6. Rules `migrater` (top-down) and `migratel` (bottom-up).

to disappear are adopted by the adjacent cells. (The starting up of a station is handled similarly.)

$$\begin{aligned}\texttt{shutdown}:\ &\texttt{chain}(\ \texttt{station}(x_1)\ ,\ \texttt{chain}(\texttt{station}(\texttt{bikes}(x_2,x_3)),\texttt{station}(x_4))\ )\\ \longrightarrow\ &\texttt{chain}(\ \texttt{station}(\texttt{bikes}(x_1,x_2))\ ,\ \texttt{station}(\texttt{bikes}(x_3,x_4)))\ )\end{aligned}$$

Again, by typability the rule takes into account all the bikes connected to the stations.

4.3 Inductive reconfigurations

The reconfigurations seen so far can be applied in any larger context with their parameters instantiated according to the specified types. However, it is often the case that a composed architecture can be reconfigured only if all its sub-components are suitably reconfigured first. Stretching the analogy between reconfigurations and rewrite systems the expressiveness of our reconfiguration language can be further increased by considering conditional labeled rewrites, defined inductively over the terms encoding style proofs. As mentioned in the introduction, this technique has been largely inspired from the area of process calculi, where it provides a well-established mechanism for defining the operational semantics of concurrent processes.

The first extension allows to limit the applicability of a reconfiguration to specific contexts, so that a complex architecture is reconfigured. The second extension allows to tag different families of rewrites depending on their role.

Simple conditional rewrites take the form

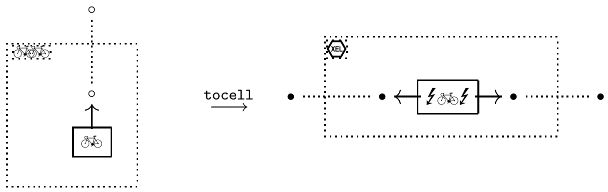

Fig. 7. Reconfiguration `bike2cell`.

$$\frac{x_1 \to x_1' \quad \ldots \quad x_n \to x_n'}{L(x_1, \ldots, x_n) \to R(x_1', \ldots, x_n')}$$

where L takes $x_1, \ldots, x_n$ as parameters, while R takes $x_1', ..., x_n'$. The meaning is that, given an assignment η of concrete architectures to the parameters $x_1, ..., x_n, x_1', ..., x_n'$, the architecture $L\eta$ can be reconfigured according to $R\eta$ only if each $x_i\eta$ can be reconfigured to $x_i'\eta$. Of course, more powerful forms of conditional rewrites are possible, e.g. where premises are arbitrary rewrites $t \to t'$. However, for the sake of our example the simple form presented is sufficient.

For instance, recall the problem of a station shutting down. We assumed that it was possible for the adjacent stations to bypass the connection and adopt the orphan bikes. We now consider the case in which this is not possible and bikes react by switching from their normal mode to a cell mode in which they become independent cells. The idea is that the bikes in the cell of the station that is shutting down will be connected as a chain. Since we need to reconfigure in one step all the bikes, we cannot provide just one ordinary style-preserving rule. Instead we can give an easy recursive definition via conditional rewrites as below.

The base reconfiguration involves a single bike (see Figure 7):

$$\texttt{bike2cell}: \ \texttt{bike} \xrightarrow{\texttt{tocell}} \texttt{bikestation}$$

The inductive case we consider is illustrated in Figure 8, where the union of two collections of bikes is reconfigured as the concatenation of the respective reconfigured cells, provided that these are possible:

$$\texttt{bikes2cells}: \ \frac{x_1 \xrightarrow{\texttt{tocell}} x_1' \qquad\qquad x_2 \xrightarrow{\texttt{tocell}} x_2'}{\texttt{bikes}(x_1, x_2) \xrightarrow{\texttt{tocell}} \texttt{chain}(x_1', x_2')}$$

Finally, the cell with the station shutting down is reconfigured by the rule illustrated in Figure 9:

$$\texttt{cell2chain}: \ \frac{x \xrightarrow{\texttt{tocell}} x'}{\texttt{station}(x) \longrightarrow x'}$$

Obviously, types are not preserved by some of these cases and thus the right- and left-hand sides of the rewriting rule cannot be applied in the same contexts.

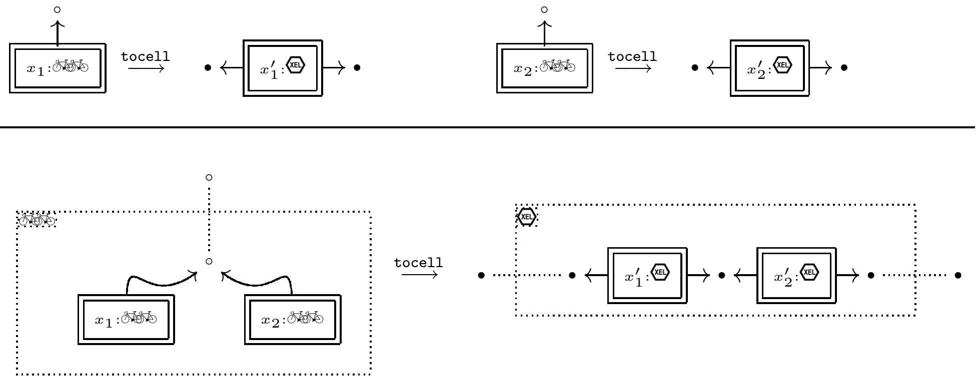

Fig. 8. Reconfiguration `bikes2cells`.

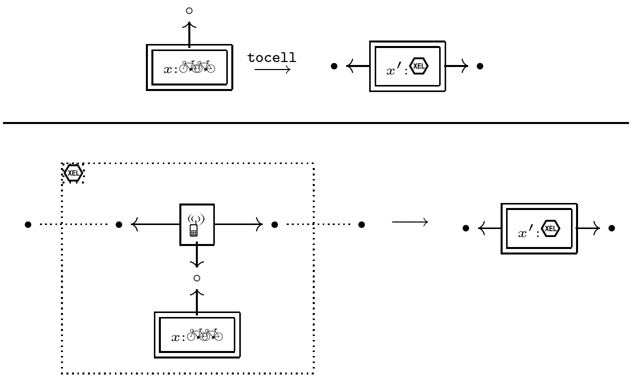

Fig. 9. Reconfiguration `cell2chain`.

But this is not a problem because rules are intended to be applied in appropriate (inductively defined) contexts. This is particularly clear in the last case of rule `cell2chain` where the premise is for a collection of bikes to become a chain cell, while the conclusion actually transforms a chain of cells into a chain of cells. The silent label makes it applicable in any larger context (unlike rewrites labeled `tocell`).

4.4 Runtime issues of reconfigurations

It is important to remark again that in our setting reconfigurations happen at the level of style proofs. Several scenarios are thus possible: (i) the architectural information is available at run-time and it is exploited in the reconfiguration; (ii) the architectural information is available at run-time but we want (or need) to construct a different proof term in order to apply a convenient reconfiguration;

(iii) no architectural information is available and we need to construct a proper proof in order to apply the reconfiguration.

Note that by *architectural information* we intend typically the proof term, but it might be also possible to consider an approximation of it that disregards of irrelevant details. Technically, this is achieved by partially evaluated terms which implicitly use derived design productions. For instance, in our example we might want to ignore the way chains were actually constructed and consider a derived production $\texttt{chain}_\texttt{i}$ that puts i designs of type in a sequence of i stations, such that e.g. for a design consisting of three cells defined by term $\texttt{chain}(\texttt{station}(x_1), \texttt{chain}(\texttt{station}(x_2), \texttt{station}(x_3)))$ the runtime information is reduced to $\texttt{chain}_3(x_1, x_2, x_3)$. Of course, the reconfiguration rules must be defined accordingly. In the particular example this would require families of reconfigurations for joining, adding and migrating with the advantage of not needing to reparse a design as it occurs if we use the original term. On the other hand, it would be very difficult to deal with the shutdown scenario where bikes change mode. In other words, there is a range of architectural information that goes from the original design construction (the entire proof term) to an abstraction of it (the proof term partially evaluated) to the actual architecture (the proof term completely evaluated). As the next paragraph discusses there is a trade-off between different aspects.

Case (i) is rather straightforward and computationally less expensive, with the advantage that even in a largely distributed system, all candidates to the reconfigurations can be statically marked and locally monitored. It is however less flexible, because when different reconfigurations are possible that are dependent on different proof terms for the same architecture, then no optimization is possible. Instead, case (ii) and (iii) require some additional global monitoring but can allow to improve the performance of the reconfigured system by taking into account non-structural aspects like those related to Quality of Service (QoS).

For instance, in our example one might want each cell to adopt those orphan bikes which are nearest to it. Considering non-structural aspects such as QoS is a current research activity that we do not report due to lack of space, but it is worth highlighting that our approach puts the basis for those scenarios in which several reconfigurations are possible and given by different but still equivalent proofs for the same architecture and one can choose the most appropriate one taking into account some criteria (based on QoS measures, for instance).

5 Conclusion

We have presented *Architectural Design Rewriting*, an approach to deal with the structural aspects of hierarchical style-based reconfigurations of software architectures. The approach is based on a simple algebra of typed graphs with interfaces, which allows for a unifying treatment of style-based design, style checking, execution and reconfiguration. This results in a simple and formal mechanism for designing architectures according to a style, for checking that an architecture is an instance of a style and for ensuring style preservation during reconfigurations.

The use of graphs and graphs transformations to model architectural styles has been proposed by several authors (see [17], for instance) who based their approaches on the concept of *shapes* in programming languages [4]. Amongst such works our paper is closely related to the approach of [10] which we extend with a simpler, more intuitive and flexible formalism (roughly speaking, we use an algebra of proof terms instead of higher-order λ-expressions). Within this research line, our algebraic approach is original. In comparison with [14], which proposes software architectural styles as graph grammars and reconfigurations as graph transformations our approach has several advantages. First, our hierarchical and inductively based approach allows us to compactly represent complex reconfiguration rules. Second, while in [14] one has to formally prove the correctness of each reconfiguration rule (which in some cases can be done by means of an algorithm), in our approach such correctness is automatically given by the fact that proof terms, rather than architectures, are rewritten.

The paper is also related to approaches that deal with reconfigurations in software architectures defined by an ADL. As far as we are aware of, our approach differs from such frameworks in that they do not consider hierarchical and inductive reconfigurations. Another difference is that we use graph rewriting as a unifying model to represent architectural design, behavior and reconfiguration, while most ADLs use different, separated formalisms for such issues. For instance, various approaches [7] mix the ingredients of a concrete ADL and the Alloy [11, 12] language to respectively describe the vocabulary and constraints that define an architectural style. In addition, we think that our proposal provides a formal basis for extending existing architectural description languages (ADLs) [13] with the novel concepts of our approach.

We believe that ADR has multiple applications. First, it could serve as a formal basis for extending existing methodologies or ADLs for dealing with architectural styles. Then, it can also be used as the basis of graphical representations of process algebras. For instance, a representation of π-calculus [15] based on the graphical encoding [5] is currently being defined. Pushing further the connection with process algebras, we are investigating suitable notions of architectural equivalences. For instance, two architectural terms can be regarded as structurally equivalent if they evaluate to the same architecture (graph), and equivalent w.r.t. to the reconfiguration rules if they are bisimilar (any possible reconfiguration of one archiecture can be mimicked by the second one resulting again in equivalent architectures).

We plan to analyze and eventually enrich our approach to support the design and management of service-oriented architectures, where design concepts include service assembly and composition [3], reconfigurations dealing with modes [9] and service invocation, discovery and binding [3], and non-structural aspects like QoS become crucial.

References

1. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. *Software and Systems Modeling*, 5(2):187–207, June 2006.
2. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
3. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *Proceedins of WS-FM'06, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.
4. P. Fradet and D. L. Métayer. Shape types. In *Proceedings of POPL '97, 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM Press.
5. F. Gadducci. Graph rewriting for the π-calculus. *Mathematical Structures in Computer Scence*, 17(3):407–437, 2007.
6. D. Garlan, R. T. Monroe, and D. Wile. ACME: an architecture description interchange language. In J. H. Johnson, editor, *Proceedings of CASCON'97, conference of the Centre for Advanced Studies on Collaborative Research*, page 7. IBM, 1997.
7. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In D. Garlan, J. Kramer, and A. L. Wolf, editors, *Proceedings of WOSS'02, 1st Workshop on Self-Healing Systems*, pages 33–38. ACM, 2002.
8. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architecures and styles with graph grammars and constraint solving. In P. Donohoe, editor, *Proceedings of WICSA1, TC2 First Working IFIP Conference on Software Architecture*, volume 140 of *IFIP Conference Proceedings*, pages 127–144. Kluwer, 1999.
9. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *Proceedings of EWSA'06, 3rd European Workshop on Software Architectures*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.
10. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. In *Proceedings of GT-VMT'04, 4th Workshop on Graph Transformation and Visual Modelling Techniques*, volume 109, pages 97–109, 2004.
11. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002.
12. J. S. Kim and D. Garlan. Analyzing architectural styles with alloy. In *Proceedings of ROSATEA'06, ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM Press.
13. 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.
14. D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.
15. R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1992.
16. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
17. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.