Hierarchical Design Rewriting with Maude[1]

Roberto Bruni and Alberto Lluch Lafuente and Ugo Montanari[2]

Department of Computer Science, University of Pisa, Pisa, Italy

Abstract

Architectural Design Rewriting (ADR) is a rule-based approach for the design of dynamic software architectures. The key features that make ADR a suitable and expressive framework are the algebraic presentation and the use of conditional rewrite rules. These features enable, e.g. hierarchical (top-down, bottom-up or composition-based) design and inductively-defined reconfigurations. The contribution of this paper is twofold: we define *Hierarchical Design Rewriting* (HDR) and present our prototypical tool support. HDR is a flavour of ADR that exploits the concept of *hierarchical graph* to deal with system specifications combining both symbolic and interpreted parts. Our prototypical implementation is based on Maude and its presentation serves several purposes. First, we show that HDR is not only a well-founded formal approach but also a tool-supported framework for the design and analysis of software architectures. Second, our illustration tailored to a particular algebra of designs and a particular scenario traces a general methodology for the reuse and exploitation of ADR concepts in other scenarios.

Keywords: Maude, Rewriting Logic, SOS, Graphs, Software Architectures.

1 Introduction

Software architectures [10,27] describe the overall structure of software systems in terms of components, their logical interrelationship and their spatial distribution. They represent the organisational blueprint for the system to be designed and often also for the process of building it. In particular, software architectures define the communication and coordination mechanisms among components. *Architectural styles* establish the rationale for certain classes of architectures, e.g. patterns that should be fulfilled also in the presence of reconfigurations. Software architectures are particularly useful for model-driven development of composite systems, since they may focus on some of the relevant aspects, amenable for a specific analysis, while inessential details may be abstracted away during the generative process.

An interesting application area of the software architecture concept is within the service-oriented computing (SOC) approach, which poses serious challenges to

[1] This work has been partly supported by the EU within the FETPI Global Computing, project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB Project TOCAI.IT.

[2] Email:bruni,lafuente,ugo@di.unipi.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the software engineer in terms of scalability, open-endedness and dynamic reconfiguration. We think that formal models are needed to guide the otherwise error-prone development of complex applications that span over heterogeneous platforms. Moreover, particular care is required for the integration of languages and models at different levels of abstraction during the design, deployment and run-time phases.

Architectural Design Rewriting. The application of model driven development methodologies to service-oriented platforms is the subject of EU-IST-FET-GC2 project SENSORIA. Within SENSORIA, we study the role of architectural styles defined in terms of well known concepts of algebraic semantics, like Meseguer's rewriting logic [23] and Plotkin's structural operational semantics (SOS) [25]. Our approach is based on a design methodology called *Architectural Design Rewriting* (ADR), which has been conceived in the spirit of initiatives like [20] that promote the conciliation of software architectures and process calculi by means of graphical methods. ADR is able to model, within the same formalism, design, execution and reconfiguration. The general idea of ADR was first presented in [6]. In [5] we showed the expressiveness and flexibility of the approach by applying ADR to the design and reconfiguration of the SENSORIA Reference Modelling Language [14], an emergent paradigm inspired by the *Service Component Architecture*.

In ADR, an architectural style is an algebra where the values (i.e. the architectures) are usually graph structures representing components and their interconnections, typed by an interface which expresses the external connection capabilities for composing larger architectures. The operators are themselves graph structures with holes (referenced via variables), and the application step consists of replacing variables with values by respecting the connection capabilities and by possibly simplifying (i.e. reducing to normal form) the resulting graph structure. ADR can exploit different kinds of graph structures, depending on the applicative domain. For example, in one of its more general forms, such graph structures can contain special nodes representing some parameters of the system taking values over a finite domain, special edges representing constraints over the admissible assignments to certain groups of nodes and the simplification phase can involve performing consistency checks, merging nodes, and solving adjacent constraints. Other examples are given by taking relational structures over nodes as edges, or by taking algebraic theories as nodes and theory morphisms as edges.

The basic idea of ADR is to internalise most design details within the algebra defining the architectural style, and to offer a much more abstract (and simpler) tool to the software engineer who takes care of the software life cycle. In particular, a software architecture is likely to be inadequate to describe the state of a system in its life cycle, since design information which is needed in the reconfiguration phases might have been abstracted away. On the other end, if the full architectural information is kept, then it is likely that many useless details are recorded. Representing partial architectural information and finding a good balance of meta-information to record is a challenge that affects the effectiveness and efficiency of the approach.

Run time behaviour is modelled in ADR by possibly decorating components with their memory values and by applying conditional rewrite rules, or SOS rules, representing the desired communication semantics in a process calculi style.

Similarly, reconfiguration is also represented by suitable rewrite rules. *Local* re-

configurations are defined by ordinary rewrite rules, while *global* reconfigurations (e.g. reconfigurations which must apply inductively to all substructures) are expressed via conditional rewrite rules or via SOS rules. The main advantages of such mechanisms is that complex reconfiguration schemes can be defined inductively at any level of abstraction and that style-guarantees during reconfiguration or execution are ensured by construction.

Contribution. In this paper we introduce a flavour of the ADR approach, called *Hierarchical Design Rewriting* (HDR) which has hierarchical graphs as the underlying algebra of designs. While in previous ADR papers only flat graphs were considered, in HDR the idea is to allow also the insertion of nested graphs within the edges of the base graph model, taking advantage of the presence of symbolic operators (whose application is modelled as an explicit substitution, i.e. without the simplification phase) besides ordinary interpreted operators. This allows the representation of partial architectural information and also provides a neat visual representation of designs. However, in this paper we propose for the first time the role of derived operators as a disciplined and efficient way to deal with partial architectural information.

We describe a methodology for implementing HDR specifications in Maude [9], that can serve as a useful executable support for the HDR approach. The methodology is illustrated on the basis of a toy case study on spam filtering, which has been selected to concentrate the attention on the main features of the approach.

The choice of Maude is mainly motivated by the fact that: (i) built-in membership equational theories directly support typability of architectures, (ii) rewrite rules directly support reconfigurations based on conditional term rewriting, (iii) standard techniques for encoding labelled transitions defined via SOS rules (needed to support inductively defined reconfigurations) also exist [28], and (iv) built-in tools such as the LTL model checker can be used to perform verification.

Structure of the paper. Section 2 gives the basic background on ADR and discussed the issue of partial architectural information. Section 3 presents our implementation of the graph structures underlying HDR. Section 4 shows how to write HDR specifications. Section 5 illustrates how to analyse HDR prototypes with ad-hoc and built-in mechanisms such as rewriting strategies or the LTL model checker. Section 6 draws some conclusion and sketches our future research programme on ADR. We assume the reader has some familiarity with the Maude syntax.

2 Architectural Design Rewriting

In this section we mainly give brief overview of the key features of ADR and introduce a discussion on the use derived operators to represent partial architectural information. We refer the reader to [5,6] for a more detailed presentation and additional examples.

Roughly an ADR system can be presented as a (many-sorted) process algebra interpreted over a particular semantic domain. The idea is that architectural designs are modelled by suitable typed hypergraphs called *designs*, which are the values of the domain. Designs come equipped with their symbolic proofs of construction called *design terms*, which are the process terms of the process algebra. The interest

Fig. 1. Type graph of the spam filter example.

is thus restricted to *style-conformant* designs, i.e. those designs for which at least one corresponding design term can be found. Finally, reconfigurations are defined over design terms instead of actual architectures, taking advantage of the algebraic presentation to define them inductively as ordinary term rewrite and conditional SOS rules. One of the main objectives of ADR is the ability to guarantee by construction that reconfigurations are style-preserving.

Type graphs and designs. The use of graphs to model architectural designs has been quite popular in the literature (e.g. [2,24]) and it combines the user-friendly visual representation with formal models for graph rewriting. As explained below, 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.

A design is a well-formed architecture with a typed interface (represented by a distinguished edge) and an internal structure (called the *body* graph). The interface is an abstract view of the design as a single component, thus hiding its internal representation (except for those nodes that are exposed in the interface). More precisely, we shall fix a vocabulary T of architectural elements (called a *type graph*) and then define designs as suitable T-typed graphs.

We recall that a *graph* is tuple $G = \langle V, E, \theta \rangle$ where V is the set of nodes, E is the set of edges and $\theta : E \to V^*$ is the tentacle function. Given a graph T (called the *type graph*), a *T-typed graph* is a pair $\langle G, t_G : G \to T \rangle$, where G is the *underlying* graph and $t_G : G \to T$ is a graph morphism.

We distinguish two kinds of edges in the type graph: terminals $\mathcal{T}$ and non-terminals $\mathcal{NT}$. Likewise string grammars, terminal edges represent basic, non-refinable, concrete components of the architecture, while non-terminal edges, represent complex, refinable, abstract components. From now on we assume that all our graphs are T-typed and omit to mention it explicitly (the type graph will be always clear from the context).

Definition 2.1 A *design* is a triple $d = \langle L_d, R_d, i_d \rangle$, where L_d is the interface graph consisting of a single non-terminal edge (called *interface*) whose tentacles are attached to distinct nodes; R_d is the body graph; and $i_d : V_{L_d} \to V_{R_d}$ is the total function associating body nodes to interface nodes.

A design d is *partial* (resp. *concrete*) if R_d contains (resp. does not contain) non-terminal edges. In service-oriented applications dealing with partial designs is natural and essential: the architecture of services is only instantiated when needed after a proper discovery, selection and binding.

Fig. 2. Example of a filtered client design in the spam filter scenario.

The different classes of edges used in the spam-filtering example are represented in the type graph drawn in Figure 1, where an explicit numbering or naming of tentacles is avoided under the assumption that the order of tentacles leaving each edge is given by considering the leftward tentacle as the first one and the remaining tentacles as clockwise ordered.

Our graphical notation uses plain boxes and boxes with title bar for terminals and non-terminals, respectively. In particular, among terminals we assume that a set of basic filtering units $\{\texttt{filter}_i\}_{i\in I}$ is available that can mark message headers with spam likeliness information by applying different criteria, possibly on different parts of the message (sender, subject, body, etc.). Moreover we assume that other two terminal edges `mux` and `dmux` are available, respectively for dispatching and for collecting messages. The non-terminals account for generic filters (`Filter`), multiplexers (`Mux`) and demultiplexers (`Dmux`), a plain mail client (`Client`) and a filtered mail client (`Fclient`).

Our visual representation of a design is obtained by drawing the interface edge as an outermost box filled with the body graph. Dashed lines denote the nodes exposed in the interface. An example of a design of type `Fclient` is in Figure 2.

Architectural styles, design productions and design terms. An *architectural style* consists of a vocabulary of architectural elements (the type graph) and a set of *design productions* indicating how they can be interconnected.

Definition 2.2 A *production* is a tuple $p = \langle L_p,\ R_p, i_p, l_p\rangle$ where $\langle L_p, R_p, i_p\rangle$ is a design with n_p occurrences of non-terminal edges in R_p that are mapped by the bijection l_p on the segment $[1, 2, \ldots, n_p]$.

Each production p has an obvious functional reading $p : A_1 \times A_2 \times \ldots \times A_{n_p} \to A_p$, where $\times$ has precedence on $\to$, A_p is the type of the interface and A_k is the type of the k-th non-terminal edge e_k of R_p (i.e. $e_k = l_p^{-1}(k)$). In fact, p can be considered as the obvious graph pasting that, when applied to a tuple of designs $\langle d_1, d_2, \ldots, d_{n_p}\rangle$ (of types $A_1, A_2, \ldots A_{n_p}$, respectively), returns a design $p(d_1, d_2, \ldots, d_{n_p})$ of type A_p obtained by replacing each non-terminal edge e_k in R_p with the graph R_{d_k} (preserving the correspondence of tentacles).

However, it should be evident that ADR modelling has at least two options regarding the level of abstraction: the purely symbolic initial algebra associated with productions, where the non-terminals become sorts and each p becomes an operator of the signature, or the fully evaluated algebra of concrete designs, where the construction proof is abandoned and the only additional information retained concerns the sort of the design, i.e. its interface.

Productions admit the same compact graphical representation as designs, the

Fig. 3. Some design productions of the spam filter scenario.

only differences are the name and type of the operation in the title bar and the total ordering of the non-terminals in the body graph, which again we assume to be implicitly given by the occidental reading direction.

Some interesting productions of our spam-filtering example are depicted in Figure 3. The top-left production is `filter` and defines a basic spam filter. Actually, since we might be interested in having different classes of spam filters we shall use a parametrised version $\texttt{filter}_i$. The nearby `seq` production takes any two designs of type `Filter` and compose them in series, so that their filtering and header-marking activities will be carried out sequentially. Production `par` composes any two designs of type `Filter` in parallel, introducing dedicated `Mux` and `Dmux` components, respectively to dispatch messages to the two filters and to collect their responses. Note that the results of sequential and parallel compositions are yet designs of type `Filter`. The bottom-left production `bypass` defines the empty filter. Finally, a filtered client is obtained by prefixing a client with a filter via the bottom-right production `wrap`. According to this syntax, the design term $\texttt{wrap}(\texttt{seq}(\texttt{filter1}, \texttt{par}(\texttt{filter2}, \texttt{mux}, \texttt{dmux}, \texttt{filter3})), \texttt{client})$ describes the `Fclient` design in Figure 2.

Reconfigurations as rewrites. Reconfigurations are defined over design terms instead of actual architectures to exploit the algebraic presentation of ADR.

A local reconfiguration rule is just a rewrite rule $l \to r$. There is a very simple sufficient condition for enforcing style preservation, namely that both l and r are terms of the same type. Then, it is possible to apply the rule in any larger architecture $C[\eta]$, where η assigns design terms to variables and where $C[]$ is any term with one hole (a *context*) with the same type as l. After the reconfiguration, the well-typed architecture $C[r\eta]$ is obtained.

For example, rule $\texttt{swap} : \texttt{seq(x1,x2)} \to \texttt{seq(x2,x1)}$ reconfigures a series of two filters by swapping the elements of the sequence. The graphical representation of the interpreted rule is given in Figure 4.

Reconfigurations as SOS rules. Reconfiguration rules of the form $l \to r$ as

Fig. 4. An unconditional rewrite rule for swapping two sequential filters.

Fig. 5. A conditional rewrite rule to serialise a filter.

defined above can be applied in any enclosing context and with any arguments. In case certain local changes in the architecture are subordinated to the corresponding adaptation of the adjacent environment 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. Simple conditional rewrites take the form:

$$\frac{t_1 \xrightarrow{a_1} t'_1 \quad \dots \quad t_n \xrightarrow{a_n} t'_n}{l \xrightarrow{a} r}$$

meaning that, given an assignment η, the architecture $l\eta$ can be reconfigured according to $r\eta$ only if each $t_i\eta$ can be reconfigured to $t'_i\eta$. The labels better discriminate the various kinds of reconfigurations that are taking place. Moreover, transition labels make it possible to change the type of the rewritten design (see [5,6] for examples), while this is not allowed when unlabelled rules are considered.

For instance, suppose one needs to define a reconfiguration to completely serialise an arbitrary filter. This can be done in SOS style by the following rules (Figure 5 illustrates rule `par2seq`):

$$\texttt{filter2seq:}\quad \texttt{filter} \xrightarrow{\texttt{2seq}} \texttt{filter} \qquad\qquad \texttt{seq2seq:}\quad \frac{\texttt{x1} \xrightarrow{\texttt{2seq}} \texttt{x3} \quad \texttt{x2} \xrightarrow{\texttt{2seq}} \texttt{x4}}{\texttt{seq(x1,x2)} \xrightarrow{\texttt{2seq}} \texttt{seq(x3,x4)}}$$

$$\texttt{par2seq:}\quad \frac{\texttt{x1} \xrightarrow{\texttt{2seq}} \texttt{x3} \quad \texttt{x2} \xrightarrow{\texttt{2seq}} \texttt{x4}}{\texttt{par(x1,x5,x6,x2)} \xrightarrow{\texttt{2seq}} \texttt{seq(x3,x4)}} \qquad\qquad \texttt{wrap2seq:}\quad \frac{\texttt{x1} \xrightarrow{\texttt{2seq}} \texttt{x3}}{\texttt{wrap(x1,x2)} \longrightarrow \texttt{wrap(x3,x2)}}$$

Partial information as hierarchical designs and derived operators. The reconfiguration rules just already suggest the convenience to deal with partial architectural information. For instance, if all the reconfigurations deal with filters it is not necessary to keep the information about the construction of multiplexers, demultiplexers and clients.

One way to achieve this is by just by evaluating those part of the architecture whose structure is no more useful. In the example, one could evaluate every term of type `Mux`, `Dmux` and `Client` resulting in a hierarchical design. But a more disciplined way is to use derived operators. For instance, suppose we want to consider just single

Fig. 6. Derived operator `dpar`.

multiplexers and demultiplexers. Then we can define a derived operator `dpar` as `dpar(x1,x2) = par(x1,mux,dmux,x2)`, which is represented in Figure 6.

We now have a binary parallel operator that abstracts away from multiplexers and lets the software architect and the reconfiguration mechanisms concentrate on the filters. For instance, the rule `par2seq` becomes simpler:

$$\texttt{par2seq}: \quad \frac{\texttt{x1} \xrightarrow{\texttt{2seq}} \texttt{x3} \quad \texttt{x2} \xrightarrow{\texttt{2seq}} \texttt{x4}}{\texttt{dpar(x1,x2)} \xrightarrow{\texttt{2seq}} \texttt{seq(x3,x4)}}$$

Another possible derived operator for parallelising filters could be convenient in a situation where no multiplexer and demultiplexer are used at all since we accept that the first parallel filter that picks up the message is responsible for filtering it, i.e. we pass from a scenario where *both* parallel filters are used to a scenario where *some* of the parallel filters is applied.

3 Implementing Hierarchical Design Rewriting

The big picture of our implementation is represented in Figure 7. We have implemented a set of modules `DESIGN-*` implementing the ADR-based algebra of hierarchical designs. These are built on a couple of modules implementing graph related concepts: graphs, graph morphisms, etc. Concrete scenarios, like that of spam filters, follow our ADR methodology and are intended to be interpreted in one ADR-suited algebra. In this section we will focus on our running example scenario and the algebra of designs described in the previous section.

We shall offer a methodology to develop ADR-based languages and use them for the design and analysis of software systems. For this purpose we will travel through our implementation from the low level of graphs to the algebra of designs to the design of a simple scenario. Due to obvious space restrictions we shall include flashes of Maude code, only. However the complete specification of our prototypical implementation can be downloaded [6]. In particular, we neglect the declaration of variables which are identified since all of them are suffixed with a natural number and their type is clear from the context.

3.1 Graphs

Though not inherent to ADR we believe that it is worth giving some hints on our implementation of graphs in Maude which is given by a couple of functional modules `GRAPH-*`. Such modules import various standard Maude modules to deal with maps, sets and lists, for instance.

The module `GRAPH-CONSTRUCTORS` contains the definition of the main sorts of

Fig. 7. Module hierarchy of the implementation of ADR in Maude.

the algebra of graphs which are `Node`, `Edge`, `Tentacle` (a view for lists of edges), `Graph` and `[Graph]`.

Sort `Graph` is intended to represent well-formed graphs, while its kind `[Graph]` might also include bad-formed graphs. Below we see the operation to construct graphs. It sticks to our formal model, i.e. it constructs a graph from a set of nodes, a set of edges and a map from edges to lists of nodes.

```
op graph : Set{Node} Set{Edge} Map{Edge,Tentacle} -> [Graph] .
```

It is easy to see that not any such tuple defines a proper graph, i.e. the tentacle function might relate edges and nodes that are not part of the graph. That is why the operation returns the kind and not the sort of `Graph`. Well-formedness is defined by a membership equation which assigns sort `Graph` to a term constructed via `graph` whenever the domain of the tentacle function is contained in the set of edges and the codomain in the set of nodes. This check is computed via function `domcodom`.

```
cmb graph(N1,E1,M1) : Graph if domcodom(N1,E1,M1) .
```

Note that also other graph operators could (but should not) produce bad-formed graphs. This mechanism serves also for debugging purposes, i.e. obtaining a bad-shaped graph is a sign of possible bugs. This technique is applied in almost all our modules. Thus we will not explain it again. It suffices to note that kinds are used to refer to possibly bad-formed structures and for each of them there is a membership axiom to determine well-formedness.

The functional module `GRAPH-MORPHISMS` implements graph morphisms. Below we see the signature of the constructor of graph morphisms: it takes two graphs and two pairs of mappings (from nodes to nodes and edges to edges) and returns a graph morphism.

```
op morphism : Graph Map{Node,Node} Map{Edge,Edge} Graph -> [GraphMorphism] .
```

Further modules implement utilities like composition functions for morphisms, membership equations for isomorphisms or the *refreshment* of graphs w.r.t to another one (i.e. renaming of the items of a graph to not clash with those of another one). This operation is fundamental in the implementation of the hyperedge replacement operation.

3.2 Hierarchical Designs

The algebra of hierarchical designs consists of two sorts `tDesign` and `hDesign` that correspond to well-formed typed *flat* designs and hierarchical designs, respectively.

Flat and hierarchical designs have constructors `tdesign` and `hdesign`. Operation `tdesign` constructs a flat design from a pair of graph morphisms (that respectively correspond to the typed interface and body of a design), the map of interface nodes to the body nodes and the list of non-terminal edges in the body.

```
op tdesign : GraphMorphism GraphMorphism Map{Node,Node} List{Edge} -> [tDesign] .
op hdesign : tDesign List{hDesign} ->  [hDesign] .
op apply : hDesign -> hDesign .
```

Operation `hdesign` takes as argument a flat design and a list of hierarchical designs. The idea is that the i-th hierarchical design corresponds to the i-th non-terminal of the flat design.

The main operation over hierarchical designs consists of flattening the first non-terminal edge, i.e. substituting the non-terminal edge by the corresponding hierarchical design. The operation is called `apply` (in honour of the analogy of making effective the symbolic application) and it basically implements the concept of type-consistent hyper-edge replacement [16].

Further operations are based on it to flatten designs at will. In particular, operation `flat` performs all possible applications to a hierarchical design such that if no symbolic occurrence is present the design is completely flattened.

3.3 Further implementation issues

In order to reason about structural properties of graphs and designs we have implemented a Courcelle's monadic second-order logic of graphs [11]. Roughly, allows to quantify over graph items and sets of graph items, to compare items, etc. The logic is expressive enough to express reachability, for instance.

To facilitate debugging activities and to offer a visual representation of our internal graph and design structures we have implemented modules that export to various graphical formats such as dot [15] and GraphML [4] so that we can then use visual tools such as Graphviz [15] and yEd [12]. In these paper we are using the latter due to the nice result in layouting hierarchical graphs.

There are several performance-related issues that might be worth discussing. Unfortunately, we do not have space for treating all of them in their deserved deepness. We want, however, mention one ADR-inherent interesting issue which regards the use of membership equations. We note that in the low level modules `GRAPH-*` there are a lot of checks that are superfluous since well-formedness is guaranteed in ADR by the way in which operations are interpreted at the higher levels. Having checked the well-formedness for the interpretations once, most equational axioms could be bypassed with a considerable saving during the rewrite and reduction steps, specially during the evaluation of flattening functions.

4 Specification with HDR

4.1 Symbolic specification

The first thing to do when designing a scenario is to define the signature of the architectural style. Below we see an excerpt of module `FILTER-STYLE` which defines the sorts for our example and the signature of the various operations. It is worth noticing the relation with the style vocabulary, i.e. the type graph (see Figure 1). In general, one should define a sort for each non-terminal edge.

```
sorts Mux Dmux Filter Client Fclient .
...
ops mux bypass : -> Mux [ctor] .
ops demux bypass : -> Dmux [ctor].
ops bypass filter : -> Filter [ctor] .
op seq : Filter Filter -> Filter [assoc id: bypass frozen ctor] .
op par : Filter Filter -> Filter [frozen ctor] .
op wrap : Filter Client -> Fclient [ctor frozen].
op client : -> Client [ctor].
```

Observe that for some of them we have introduced axioms and attributes. For instance the fact that a `bypass` is the identity element of `seq`, the operation that puts two filters in sequence. The axiomatisation of the free algebra is not always trivial and depends on the axiomatisation of the intended interpreted algebra: axioms should not identify symbolic terms as equivalent if the corresponding evaluations are not equivalent. In particular we want to work with designs up to isomorphism and this means that our symbolic axioms should not identify non-isomorphic designs.

The axiomatisation of the symbolic algebra might be a complicated and error-prone task. The readers are invited to test this by reasoning why we did not define `par` to be commutative and associative and to figure out what axioms would hold for a derived operator where multiplexers and demultiplexers are bypassed.

It is also worth noticing that some operations are declared to be `frozen`. This is needed to encode SOS-based reconfiguration rules as we shall see. Recall that a frozen operators inhibit rewrite at their subterm arguments [7].

Once fixed the signature we can already define a module `FILTER-RECONFIGURATION` with reconfiguration rules that act at the symbolic level of abstraction. Recall that ADR combines both rewriting logic and SOS-like rewriting rules.

As an example of an ordinary rewrite rule consider the operation that swaps a sequence of filters. The graphical representation can be seen in Figure 4.

```
rl [swap] : seq(x1,x2) => seq(x2,x1) .
```

In ADR one can design more complex reconfigurations like the one that takes an arbitrarily complex filter and serialises it. We see below how this is implemented in Maude. Our encoding of SOS follows [28]. Recall, that in our notation unlabelled rules correspond to reactive rules while labelled ones are in SOS style. Here *label* refers to the SOS label, i.e. the prefix of the target used in SOS rules, while the *name* of the rule refers to the Maude label (the tag between the brackets).

Thus, a first set of rules has the purpose of propagating unlabelled rules so that they can be applied in any context. This is necessary since operations are declared to be frozen. Below we see the propagating rules for context `seq`.

```
crl [prop] : seq(x1,x2) => seq(x3,x2) if x1 => x3 .
crl [prop] : seq(x1,x2) => seq(x1,x3) if x2 => x3 .
```

Terms are rewritten into their serialised version prefixed with label `2seq`. For instance, a `filter` is trivially serialised into a `filter` via rule `filter2seq`. Rewriting `ser` and `par` occurrences require both their arguments to be appropriately rewritten with label `2seq`. This is performed by rules `seq2seq` and `par2seq`. A last unlabelled rule `wrap2seq` allows to rewrite a wrapped client given that its filter is serialised. The last rule is unlabelled and thus closes the SOS inference. This means that it can be applied in any larger context where the filtered client can occur.

```
rl [filter2seq] : filter => {'2seq}filter .
crl [seq2seq] : seq(x1,x2) => {'2seq}seq(x3,x4)
 if x1 => {'2seq} x3 /\ x2 => {'2seq} x4  .
crl [par2seq] : par(x1,x5,x6,x2) => {'2seq}seq(x3,x4)
 if x1 => {'2seq} x3 /\ x2 => {'2seq} x4 .
crl [wrap2seq] : wrap(x1,y1) => wrap(x2,y1) if x1 => {'2seq} x2 .
```

Recall that Figure 5 depicts the graphical representation of rule `par2seq`.

4.2 Interpreted Specification

The next step to do is to give the interpretation of the abstract view in the selected ADR-based algebra. We show our choice for handling the algebra of hierarchical designs. Module `FILTER-DESIGN` starts defining the type graph that constitutes our architectural vocabulary (see Figure 1).

```
eq typegraph = graph( (filterpoint) ,
                      (filter-ed , Filter-ed , Fclient-ed, Client-ed, client-ed, mux-ed,
                       dmux-ed, Mux-ed, Dmux-ed),
                      (filter-ed |-> filterpoint filterpoint ,
                       Filter-ed |-> filterpoint filterpoint,
                       mux-ed |-> filterpoint filterpoint filterpoint ,
                       dmux-ed |-> filterpoint filterpoint filterpoint ,
                       Mux-ed |-> filterpoint filterpoint filterpoint ,
                       Dmux-ed |-> filterpoint filterpoint filterpoint ,
                       Fclient-ed |-> filterpoint , Client-ed |-> filterpoint ,
                       client-ed |-> filterpoint)) .
```

To avoid confusion we do not overload operations and sorts and prefer to use suffix with `-ed` for each type of edge.

Next, for each design production we define a constant (with name suffixed with `-dp`) consisting of the design that corresponds to the design production. For instance `seqdp` is the design defined below. Note how edges `e(1),e(2)` are the non-terminals of type `Filter` that correspond to the arguments of the operation.

```
eq seq-dp = tdesign(morphism(graph((n(1) , n(2)),e(1),(e(1) |-> n(1) n(2))),
                                 (n(1) |-> filterpoint , n(2) |-> filterpoint),
                                 (e(1) |-> filterse),
                                  typegraph),
                    morphism(graph((n(1) , n(2) , n(3) ),
                                 (e(1) , e(2)),(e(1) |-> n(1) n(3) , e(2) |-> n(3) n(2) ) ),
                                 (n(1) |-> filterpoint , n(2) |-> filterpoint ,
                                  n(3) |-> filterpoint ),
                                 (e(1) |-> Filter-ed , e(2) |-> Filter-ed),
                                 typegraph),
                    (n(1) |-> n(1) , n(2) |-> n(2)),
                    (e(1) e(2))) .
```

Now, for each operation `f` we define an interpreted version `f-i`. We prefer to do like this instead of using equations in order to be free to combine symbolic and interpreted operations and to have rewrite rules to interpret or uninterpret (i.e. typing) at will. We see below how we use the constants we mentioned above and the operation `flat` to define the interpreted operations.

```
eq bypass-i = hdesign(bypassdp,nil) .
eq filter-i = hdesign(filterdp,nil) .
eq seq-i(x1,x2) = flat(hdesign(seqdp,(x1 x2))) .
```

```
eq par-i(x1,x2,x3,x4) = flat(hdesign(pardp,(x1 x2 x3 x4))) .
eq wrap-i(x1,y1) = flat(hdesign(wrapdp,(x1 y1))) .
```

The design in Figure 2 is the result of evaluating:

```
wrap-i(seq-i(filter-i,par-i(filter-i,mux-i,dmux-i,filter-i)),client-i)
```

4.3 Derived operators and Partially Interpreted Designs

We now return to the issue of considering partial architectural information. As we already stated, one way to deal with it is to just evaluate some parts of a design term. In order to have design terms that mix evaluated and symbolic parts we need to declare the sorts of our style algebra to be subsorts of `hDesign`. For instance, a term like

```
\texttt{iwrap(seq(filter, par(filter, imux, imux, filter)), iclient)}
```

preserves architectural information of filters only.

It is worth noticing that a partially interpreted term is, implicitly, a derived operator. However, if one does not use an explicit symbolic operator, writing rules and matching them becomes cumbersome and inefficient. We believe that a disciplined way of dealing with partial information is by means of suitable derived operators, which can be seen as encoding of atomic design steps.

Consider, for example, the variant of the scenario that we suggested in Section 2 where one is interested in single multiplexers and demultiplexers only. We define the signature for `dpar`.

```
op dpar : Filter Filter [ctor].
```

The declaration as constructor and the direction of the equation are not an accident: we want now the derived operator to be in normal form, to be able to use it efficiently in the left-hand side of reconfigurations.

Its interpreted version can now be given directly or in terms of the initial algebra.

```
eq dpar-i(x1,x2) = par-i(x1,mux-i,dmux-i,x2) .
```

We can now adapt the reconfiguration to serialise parallel filters:

```
crl [par2seq] : dpar(x1,x2) => {'2seq}seq(x3,x4) if x1 => {'2seq} x3 /\ x2 => {'2seq} x4 .
```

4.4 Further specification issues

ADR is well suited for a design-by-refinement approach. In order to automatise the procedure we can implement a module `FILTER-REFINEMENT` where we give a rewrite theory that simulates the refinement process. In the code excerpt below we see how we can construct all possible design terms of type `Filter`. Constant `Filter-nt` stands for a non-terminal symbol of type `Filter`. The rules then can be seen as the context-free graph grammar view of our design productions. For instance, rule `seq` refines a filter as a sequence of filters.

```
op Filter-nt : -> Filter [ctor] .
rl [bypass] : Filter-nt => bypass .
rl [filter] : Filter-nt => filter(0) .
rl [filter] : Filter-nt => filter(1) .
...
rl [seq] : Filter-nt => seq(Filter-nt,Filter-nt) .
rl [par] : Filter-nt => par(Filter-nt,Mux-nt,Dmux-nt,Filter-nt) .
```

In general, the construction of such a rewrite theory for a given set of productions is straightforward: all one needs is a non-terminal symbol for each sort and a

rule Σ-nt => f(Σ_1-nt,...,Σ_n-nt) for each operation f : Σ_1 ... Σ_n -> Σ. A useful application of such rewrite theories is that of model finding via rewrite strategies [21].

Following the analogy of the design, where we have an abstract view and an interpreted one, we have can also implement specification mechanisms tailored to the symbolic level. A possible approach is to follow the same principles as standard spatial logics (e.g. [8]) where each operation of the algebra has modal operator as counterpart. A more ad-hoc approach consists of defining ad-hoc properties directly (and thus more efficiently).

For instance, we can implement a binary predicate `!>` to be used as proposition in the built-in LTL model checker. Predicate `f1!>f2` holds whenever `f2` does not precede `f1`. It is based on predicate `appears`. The predicate `appears(f1)` holds if the filter `f1` is used. Below, we see the implementation of the satisfaction relation `|=` for predicate `!>`.

```
op _ !> _ : Filter Filter -> Prop .
eq wrap(f1,c1) |= f2 !> f3 = f1 |= f2 !> f3 .
eq bypass |= f1 !> f2 = true .
eq filter(id1) |= f1 !> f2  = true .
eq seq(fp1,fp2) |= f1 !> f2
   = fp1 |= f1 !> f2
     and fp2 |= f1 !> f2
     and not (fp1 |= appears(f2) and fp2 |= appears(f1)) .
eq par(f1,m1,d1,f2) |= f3 !> f4
   = f1 |= f3 !> f4 and f2 |= f3 !> f4 .
```

5 Analysis with ADR

In this section we give a brief illustration on how we can use an ADR specification to analyse software architectures. A typical software analysis activity is *model finding*, i.e. to look for models violating or satisfying some properties in order to better understand and eventually correct the kind of architectures (e.g. the style) under consideration. As we have already mentioned, model finding is supported in our implementation of ADR by rewrite theories simulating the design-by-refinement process.

Suppose for instance, that we are interested in finding configurations where filter occurrences satisfy some ordering constraints. We start defining some strategies regarding the refinement process: `nt-refinement(n)` requires to perform n non-terminal refinement steps, `insert-filter` introduces a filter. Strategy `(amatch Filter-nt ? fail : idle)` ensures that no filter remains undefined. In order to ensure that the resulting design satisfies the desired ordering constraints we can combine the `match` strategy operator with the satisfaction of predicates. In the example below, for instance, we require (occurrences of) filter 0, 1 and 2 to appear such that filter 1 does not precede filter 0, filter 2 does not precede filter 1 and filter 0 precedes (`<`) filter 2.

```
Maude> srew in FILTER-MODELCHECK : FClient-nt using nt-refinment(3); insert-filter *;
       (amatch Filter-nt ? fail : idle) ;
       match fk1 such that modelCheck(fk1, appears(filter(2))
         /\ appears(filter(1)) /\ appears(filter(0))
         /\ filter(0) !> filter(1)
         /\ filter(1) !> filter(2)
         /\ filter(0) < filter(2) ) .
...
Solution 1
rewrites: 821950 in 3292ms cpu (3289ms real) (249665 rewrites/second)
result FClient: wrap(seq(filter(0), filter(1), filter(2)), Client-nt)
```

```
...
Solution 7
rewrites: 2580108 in 11184ms cpu (11191ms real) (230681 rewrites/second)
result FClient: wrap(par(filter(1), Mux-nt, Dmux-nt, seq(filter(0), filter(2))), Client-nt)

No more solutions.
rewrites: 2948352 in 12300ms cpu (12307ms real) (239688 rewrites/second)
```

We obtain 7 different configurations. We can choose one of them, say the last, and verify if some of the ordering constraints are invariantly preserved by reconfigurations. For instance, we can drop the requirement regarding filters 2 and 0 from the invariant and, of course, we exclude the swap rule from our reconfigurations (it would lead to all possible orderings).

```
Maude> red modelCheck(
       wrap(par(filter(1), Mux-nt, Dmux-nt, seq(filter(0), filter(2))), Client-nt),
       [] (filter(0) !> filter(1) /\ filter(1) !> filter(2))) .
...
rewrites: 16466 in 88ms cpu (88ms real) (187103 rewrites/second)
result ModelCheckResult:
counterexample(
{wrap(par(filter(1), Mux-nt, Dmux-nt, seq(filter(0), filter(2))), Client-nt),'prop}
...
{wrap(par(filter(1), mux, dmux, par(filter(0), mux, dmux, filter(2))), Client-nt),'parall})
```

The model checker returns a counterexample. Indeed, the serialisation of filters can violate some of the ordering properties. Now, we could ask whether there is some configuration satisfying the model constraints as well as the invariant. This time we combine rewriting strategies with the LTL model checker:

```
Maude> srew in FILTER-MODELCHECK : FClient-nt using nt-refinment(3); insert-filter *;
       (amatch Filter-nt ? fail : idle) ;
       match fk1 such that modelCheck(fk1, appears(filter(2))
         /\ appears(filter(1)) /\ appears(filter(0))
         /\ [] (filter(0) !> filter(1)/\ filter(1) !> filter(2))
         /\ filter(0) < filter(2) ) .

...
Solution 1
rewrites: 927963 in 3640ms cpu (3641ms real) (254918 rewrites/second)
result FClient: wrap(seq(filter(0), filter(1), filter(2)), Client-nt)

Solution 2
rewrites: 1842210 in 8304ms cpu (8306ms real) (221832 rewrites/second)
result FClient: wrap(seq(par(filter(0), Mux-nt, Dmux-nt, filter(1)), filter(2)), Client-nt)

Solution 3
rewrites: 3121703 in 14156ms cpu (14158ms real) (220507 rewrites/second)
result FClient: wrap(seq(filter(0), par(filter(1), Mux-nt, Dmux-nt, filter(2))), Client-nt)

No more solutions.
rewrites: 3635581 in 15716ms cpu (15719ms real) (231315 rewrites/second)
```

We see that there are three possible such configurations. By an exhaustive analysis of the respective state spaces using command `search` we can indeed confirm that the invariant holds.

6 Conclusion

We have presented a prototypical implementation of *Hierarchical Design Rewriting* (HDR), a flavour of ADR [5] with the concept of *hierarchical design*, which allows for system specifications where some parts remain at the most abstract level and others are interpreted. More precisely, we have explained how we are experimenting with our algebra of hierarchical designs in Maude and have illustrated the use of HDR for the development of a simple spam-filter architecture scenario.

We believe that the presentation of our prototypical implementation offers sev-

eral contributions. First, we show that ADR is not only a well-founded theoretical approach but also a tool-supported framework for the design and analysis of software architectures. Second, for the sake of this presentation, even if we concentrate on one particular algebra of designs and one particular scenario, we trace a methodology for developing other ADR-suited algebras and scenarios. Last but not least, we offer a further validation of the suitability of rewriting logic as a formalism for the development and analysis of software systems.

Related Work. Our approach has been mainly inspired by graph-based approaches to architectural styles [17,24]. Indeed, ADR recast the work presented [17] in algebraic terms, enriching it with standard rewrite mechanisms.

We have also taken inspiration from initiatives that promote the conciliation of software architectures and process calculi by means of graphical methods [20] and the unifying treatment of software refactoring, synthesis and development as algebras over programs [3].

ADR also shares concepts with various approaches ranging from process calculi that deal with reconfigurable component based architectures (e.g. [1]) to graphical representation of concurrent systems such as those based on Synchronized Hyperedge Replacement [13] or Bigraphs [18].

Maude has already been used in approaches to software architectures. For instance, in [19] Maude is used to model and verify software architectures given in LfP, a system description language with hierarchical behaviour. Another example is [26] where a rewriting semantics of the CBabel architecture description language is defined.

ADR does not marry a particular model or language and its principles could be applied to the above mentioned approaches. For instance, one could try to define ADR architectural styles for SHR or Bigraph-based specifications.

Current and Future Work. We plan to work further on our prototypical implementation. We are also interested in a deeper treatment of specification and verification features with a special focus on model checking techniques to tackle the state space problem, e.g by considering equational abstractions [22] guided by the hierarchy. We will also work on validating the usability and expressiveness of ADR by implementing encodings of various of the process calculi that are being developed within SENSORIA, with a special regard to those that consider SOC-related aspects such as sessions, transactions and compensations.

References

[1] N. Aguirre and T. S. E. Maibaum. Hierarchical temporal specifications of dynamically reconfigurable component based systems. *ENTCS*, 108:69–81, 2004.

[2] 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.

[3] D. S. Batory. Program refactoring, program synthesis, and model-driven development. In S. Krishnamurthi and M. Odersky, editors, *Proceedings of the 16th International Conference on Compiler Constuction (CC'07)*, volume 4420 of *LNCS*, pages 156–171. Springer, 2007.

[4] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. GraphML progress report: Structural layer proposal. In *Proceedings 9th International Symposium on Graph Drawing (GD '01)*, volume 2265 of *LNCS*, pages 501–512. Springer, 2002. http://graphml.graphdrawing.org/.

[5] 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)*, volume 4912 of *LNCS*, pages 186–203. Springer, 2007.

[6] 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. See http://www.albertolluch.com/adr.html.

[7] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

[8] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194–235, 2003.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude — A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 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] yEd graph editor homepage. http://www.yworks.com/en/products_yed_about.htm.

[13] 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.

[14] J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *Proceedins of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'06)*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.

[15] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 2000. http://www.graphviz.org/.

[16] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer Verlag, 1992.

[17] D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *ENTCS*, 109:97–109, 2004.

[18] O. H. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.

[19] C. Jerad, K. Barkaoui, and A. Grissa-Touzi. Hierarchical verification in Maude of LfP software architectures. In F. Oquendo, editor, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2007.

[20] B. König, U. Montanari, and P. Gardner, editors. *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems, 6.-11. June 2004*, volume 04241 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005. http://www.dagstuhl.de/04241/.

[21] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. *ENTCS*, 117:417–441, 2005.

[22] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Proceedings of the 9th International Conference on Automated Deduction (CADE'03)*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.

[23] J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3):213–237, 2007.

[24] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[25] G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[26] A. Rademaker, C. de O. Braga, and A. Sztajnberg. A rewriting semantics for a software architecture description language. *Electr. Notes Theor. Comput. Sci.*, 130:345–377, 2005.

[27] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.

[28] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.