A Formal Support to Business and Architectural Design for Service-oriented Systems*

Roberto Bruni[1], Howard Foster[2], Alberto Lluch Lafuente[3], Ugo Montanari[1], and Emilio Tuosto[4]

[1] Department of Computer Science, University of Pisa, Italy
{bruni,ugo}@di.unipi.it
[2] Imperial College London, UK
howard.foster@imperial.ac.uk
[3] IMT Institute for Advanced Studies Lucca, Italy
alberto.lluch@imtlucca.it
[4] Department of Computer Science, University of Leicester, UK
emilio@mcs.le.ac.uk

Abstract. *Architectural Design Rewriting* (ADR) is an approach for the design of software architectures developed within SENSORIA by reconciling graph transformation and process calculi techniques. The key feature that makes ADR a suitable and expressive framework is the algebraic handling of structured graphs, which improves the support for specification, analysis and verification of service-oriented architectures and applications. We show how ADR is used as a formal ground for high-level modelling languages and approaches developed within SENSORIA.

1 Introduction

The IST-FET Integrated Project SENSORIA aims at developing a comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated into pragmatic software engineering processes. The development of mathematical foundations and mathematically well-founded engineering techniques for service-oriented computing constitutes a key research activity of SENSORIA.

In this paper we report the outcome of some research efforts within SENSORIA aimed at developing formalisations of high-level modelling languages for service-oriented systems. More precisely, we present *Architectural Design Rewriting* (ADR) [5] and we explain how ADR can be used as a formal model for architectural and business design and how it helps in formalising crucial aspects of the UML4SOA and SRML modelling languages, see Chapter 1-1 (UML Extensions for Service-Oriented Systems) and Chapter 1-2 (The SENSORIA Reference Modelling Language), respectively, as well as the software modes approach [9].

ADR has been inspired by the long-term experience of SENSORIA researchers on Graph Transformation Systems, process calculi and software engineering and

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

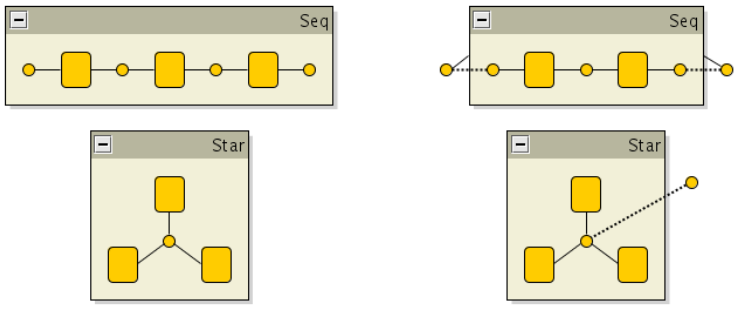


Fig. 1. "Graphs within boxes" (left) and "Graphs within edges" (right)

has been entirely developed under the SENSORIA project, allowing us to establish interesting links with many other formalisms developed within SENSORIA and making it possible to strengthen the collaboration among different research groups with different expertise within the project.

Synopsis. § 2 explains the motivating principles of ADR. § 3 introduces ADR basics and explains how ADR can be considered as an Architectural Description Language on the basis of a running example. § 4 overviews some key aspects of ADR-based formalisations of UML4SOA, SRML and Sofware Modes.

2 Rationale behind Architectural Design Rewriting

The use of graphs or diagrams of various kinds is pervasive in Computer Science, as they are very handy for describing in a two-dimensional space the logical or topological structure of systems, models, states, behaviours, computations, etc.; the reader might be familiar, for example, with the classical graphical presentations of entity-relationship diagrams, of finite state automata and labelled transition systems, of data-structures (like various kinds of lists and trees), static and behavioural UML diagrams (like class, message sequence and state diagrams), of computational formalisms like Petri nets, and so on. One obvious advantage of using graphs or diagrams lies in their ability to represent in a direct way relevant topological features of the systems or models they describe.

On the one hand, software architectural models are intended to describe the structure of a system in terms of computational components, their interactions, and its composition patterns [12]. Using *plain hypergraphs*, the above perspective can find its realisation by modelling "components" and "connectors" as hyperedges and their interconnecting "ports" as nodes. Moreover, nodes, hyperedges and their tentacles can be *typed* so to discard erroneously linked systems.

On the other hand, in [2] it is argued that structured graphs are the most appropriate ones for service-oriented and global computing systems, where scalable techniques and open-ended specifications are important issues not immediately

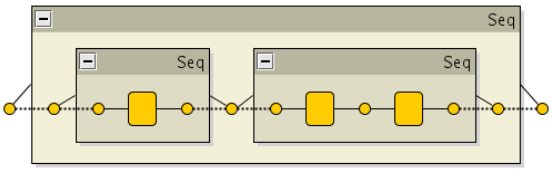


Fig. 2. "Nested graphs"

met by plain hypergraphs alone. Structured graphs offer better support for "understanding" graphs (like parsing and browsing large systems), designing systems (like expressing requirements and specifications, facilitating abstraction and refinement, allowing modularity and seamless composition), supporting automated analysis and verification (like model construction, model conformance, behavioural analysis, assessing sound reconfiguration and refactoring transformations) and last but not least, sound and complete visual encoding of computational systems.

Different kinds of structures can be super-imposed on graphs. The simplest one is enclosing a graph G in some sort of box whose label B implicitly defines some properties of the enclosed graph, i.e., its *style*. For example, Fig. 1 (left) shows some basic examples of "topologically" labelled graphs, that can be written, e.g., $Seq[G]$ and $Star[G']$ (for obvious graphs G, G' derivable from Fig. 1) or, equivalently, as membership annotations $G : Seq$ and $G' : Star$, where Seq can be read as the set of all sequentially-linked graphs, and similarly for $Star$.

The natural extension of taking a "graphs within boxes" view is then the "graphs within edges" (or "nested graphs") view, where boxing can be iterated by allowing style labels that are edges themselves (see Fig. 1 (right)). Note that the boxed interfaces are now equipped with tentacles and that dotted lines make explicit the correspondence between inner nodes exposed by interfaces and actual nodes where the module is linked to. When the nodes attached to the tentacles of its (outermost) interface are read as formal parameters, we call it a *design*. This way, boxes can be read as enhanced interfaces allowing for more sophisticated forms of containment, strctured composition, modular specification, logical hierarchies, and node sharing, among others, making such features easily understandable also to non-specialists. For example, Fig. 2 shows that sequential composition of (nested) sequential graphs still yields a sequential graph. Note that if we remove all enclosing boxes from a nested graph then we are left with an *underlying* plain graph, but we loose any information about the conceptual organization of its elements. Thus in a sense, nested graphs can be read as "blueprints" of their underlying graphs.

3 Design Foundations

This section overviews the principles of *Architectural Design Rewriting* (ADR) [5].

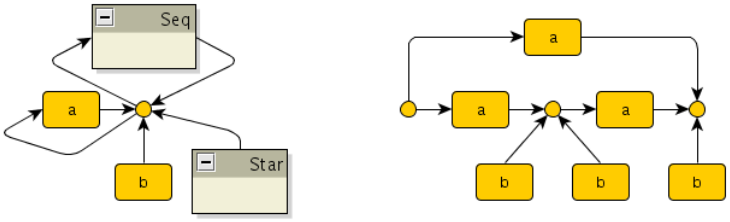


Fig. 3. A type graph (left) and a configuration (right)

3.1 System Configurations

A system configuration in ADR is the underlying graph of a design, representing the architectural units and their interconnections. Recall that a *graph* is a 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. From now on we assume that graphs are always typed over a suitable type graph T, even if sometimes it is not described explicitly. Intuitively, a type graph plays the role of an architectural vocabulary and enforces certain tentacles to be connected to nodes of a given type, but note that a type graph T itself cannot impose any sophisticated topological structure on T-typed graphs.

The distinction between refinable components and non-refinable components in software architectures amounts to the distinction between non-terminal and terminal edges in ADR. The underlying idea is the same: a non-terminal edge is an edge intended to be refined (i.e., replaced by an arbitrarily complex graph). Non-terminal edges can appear in designs, representing unspecified parts of a configuration (a refinable component) or in design productions (see later). Terminal edges instead represent parts of a graph that cannot be further refined (non-refinable components).

Fig. 3 shows a type graph (left) and a configuration typed over it (right), where the typing is made explicit by the shapes and labels of nodes and edges. Refinable components are represented as group-boxes, while non-refinable components as plain boxes. The type graph in Fig. 3 includes both kinds of edges, while the configuration is *ground*, in the sense that it consists of terminal edges only.

3.2 Architectural Designs

An architectural *design* is a nested graph representing a structured system configuration. Technically, 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 (the *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 (injective) function that maps interface nodes to body nodes.

For example, Fig. 4 shows three designs: the design on the left has a ground body graph that matches the intuition of its interface edge *Star*; the design on

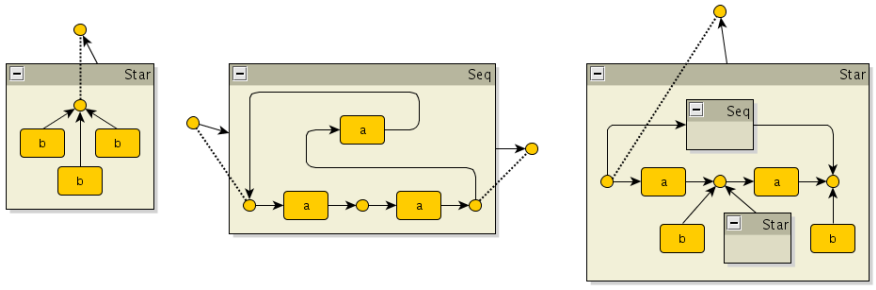


Fig. 4. Three designs

the center has a ground body graph shaped as a ring, hence not exactly matching the intended meaning associated with the interface edge *Seq*; the rightmost design has a body graph involving all different kinds of edges and exhibiting little correspondence with the intuitive meaning of its interface edge *Star*.

3.3 Architectural Styles

To avoid the above raised problems, the shape of graphs embedded in a design must be constrained. To this aim architectures are designed inductively by a set of composition operators called *design productions* which enable: (i) top-down refinement, like replacing a refinable components with its (possibly non-ground) realisation, (ii) bottom-up typing, like inferring the "style" of a system configuration, and (iii) well-formed composition, like composing some well-typed architectures together so to guarantee that the result is still well-typed.

Design productions take inspiration from Graph Grammars [7], where hyper-edge replacement rules allow to substitute, in a graph G, a (refinable) edge L with a particular graph R, suitably connected to the nodes of G where L was connected to. Technically, a design production $p = \langle L_p, R_p, i_p \rangle$ is very much like a design but with an order on the non-terminal edges $\{e_1, ..., e_{n_p}\}$ appearing in its body graph (intuitively, the order of the arguments they represent). 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 non-terminal symbol labelling the k-th non-terminal edge e_k of the body of the production. The functional type means that p can be considered as a function that when applied to a tuple $\langle d_1, d_2, \ldots, d_{n_p} \rangle$ of designs such that $d_i : A_i$, 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 each non-terminal edge e_k in it with body graph R_{d_k} of d_k respecting its tentacle function i_{d_k}, for $k = 1, \ldots, n_p$.

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 a refinement of an abstract component of type A_p as an assembly of concrete and abstract components, the latter being of type $A_1, A_2, \ldots A_{n_p}$.

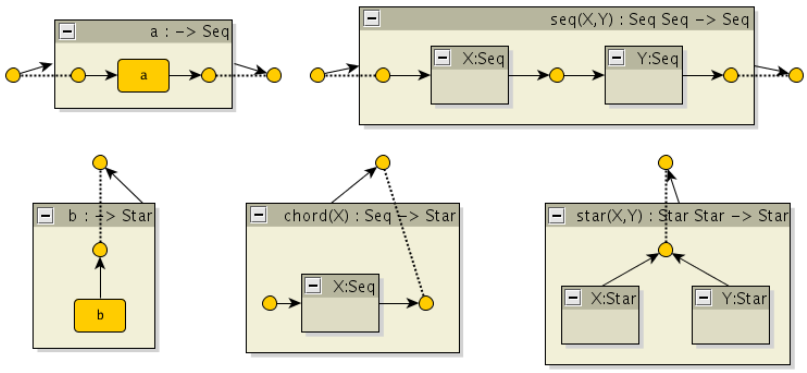


Fig. 5. Architectural styles for sequences and stars

For example, Fig. 5 shows simple design productions for configurations shaped as sequences (*Seq*) and stars (*Star*). Note that, according to the functional flavor described above, the labels of enclosing boxes are enriched with the information about the name of the production, the names of its "arguments", their types and the type of the result. For example we have constant building blocks a and b (respectively for *Seq* and *Star*) and operation $seq(X, Y)$ that takes two arguments X, Y of style *Seq* and returns a graph of style *Seq* obtained by concatenating the two ends of X and Y to form a sequence and exposing a suitable interface. Another possible reading for the rules is the following: a graph has style *Seq* if it is either a single component a or the sequential composition of two other graphs of styles *Seq*; a graph has style *Star* if it is a single component b or a sequence seen as a chord, or the joint composition of two other graphs of styles *Star*.

3.4 Design Algebra

One key feature of architectural styles is that design productions provide us with a signature for defining graphs. Furthermore, the terms over such a signature do not even need to mention node names or edge names, because the way in which components are connected is entirely embedded in each operation (i.e., in each design production). For example, a term like $star(b, chord(seq(seq(a, a), a)))$ describes a ground configuration, that is conformant to style *Star*: a component b joined with a chord embedding three components a.

In general, it can be the case that different terms denote the same underlying configuration, like $star(b, star(b, b))$ and $star(star(b, b), b)$: they essentially correspond to the graph in Fig. 4 (left). In some cases this distinction can be even desirable, to mark significant design choices no longer recoverable from the configuration itself. In other cases, the distinction can be annoying, because the order in which certain refinement steps are applied is not essential. Often the latter situation can be dealt with at the level of design algebra by imposing suitable structural congruence axioms. All such axioms must be *sound*, in the

sense that terms denoting non-isomorphic ground configurations must be kept distinct. However the axiomatization is not required to be *complete*, i.e. terms that are not structurally congruent may still denote the same graph (up to isomorphism). For example, the associativity and commutativity of $star(\cdot,\cdot)$ and the associativity of $seq(\cdot,\cdot)$ are natural axioms for our running example.

3.5 Design Reconfiguration

Software architectures might evolve in different dimensions. First, they might change statically when components are refined or architectures are assembled together. At run-time instead, architectures might evolve due to actions of normal behaviour or reconfigurations. Components leaving or joining the system can require correcting actions that lead the system into a proper state. Sometimes a reconfiguration rule can be described as a direct manipulation of a design or its corresponding term (without variables). However, reconfigurations arise more naturally and in a well-disciplined way at the abstract level of the architecture, i.e., as manipulations of designs. 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.

Reconfiguration as Graph Rewrites Since our configurations are represented by graphs, reconfigurations can be defined as graph transformations [7], e.g. based on the single-pushout and double-pushout approaches. Basically the rules come with left- and right-hand side graphs G_L, 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. There can be items shared by G_L and G_R that are required to trigger the rewrite, but are preserved by the transformation.

This view operates on flat, unstructured graphs, thus disregarding the architectural information and, for instance, not guaranteeing style preservation in the general case. When style preservation is a requirement, we need either ad hoc proofs or a rule format that ensures that any reconfiguration of a well-styled configuration leads to another well-styled configuration. This is not obvious to set-up for graph rewriting techniques as one has to consider all possible contexts where rules are applied.

Synchronised Hyperedge Replacement (SHR) [8] is a graph-based framework for modelling the operational semantics of systems with mobility and multiple synchronisation. Several flavours of SHR semantics exist, but here we focus on a variant without mobility and with Milner dyadic synchronisation, where pairs of complementary transition actions (e.g. read and write) can be synchronised. Since each transition may carry more than one action the synchronisation might involve the whole system. In § 4.2 we shall see that this variant is suitable for defining SRML run-time semantics with ADR. In particular, rewrite rules in that style do not change the interface of components and this is a sufficient condition for the style preservation.

Reconfiguration as Term Rewrites. 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 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 terms of our design algebra with the same type. Typically, both l and r may contain (typed) variables, but they are linear and all the variables in r appear in l. These variables can be instantiated in any way consistent with the types, and both r and l can be freely contextualized in larger contexts. Then, it is possible to apply the rule in any larger architecture $t(l\sigma)$, where σ assigns proof terms to variables and where the type of the hole in $t(\cdot)$ is at the same as the type of l. After the reconfiguration, the architecture $t(r\sigma)$ is obtained.

There is a 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. For example, the rewrite rule $\mathit{chord}(\mathit{seq}(a, x)) \to \mathit{star}(b, \mathit{chord}(x))$ can be applied under $\mathit{star}(b, \cdot)$ with substitution $x \mapsto a$ for rewriting $\mathit{star}(b, \mathit{chord}(\mathit{seq}(a, a)))$ to $\mathit{star}(b, \mathit{star}(b, \mathit{chord}(a)))$.

However, it is often the case that a structured 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 is increased by considering conditional labelled rewrite rules, defined inductively over the terms encoding style proofs in SOS style:

$$\frac{x_1 : S_1 \xrightarrow{a_1} x'_1 : S'_1 \quad \ldots \quad x_n : S_n \xrightarrow{a_n} x'_n : S'_n}{l(x_1, \ldots, x_n) \xrightarrow{a} r(x'_1, \ldots, x'_n)}$$

The labels $a, a_1, ..., a_n$ tag the kind of rewrite under consideration. The meaning of such a rule is that, given any assignment σ of concrete architectures to the parameters of l and r, the architecture $l\sigma$ can be reconfigured according to $r\sigma$ only if each $x_i\sigma$ (conformant to style S_i) can be reconfigured to $x'_i\sigma$ (conformant to style S'_i). 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. But this is not a problem because rules are intended to be applied in appropriate (inductively defined) contexts. When no tag labels a rewrite step, then we tacitly assume that its source and target have the same style and that the rewrite step can be applied in any larger context. For example the three SOS rules below account for sequence to star transformation:

$$\frac{}{a \xrightarrow{\star} b} \qquad \frac{x : \mathit{Seq} \xrightarrow{\star} x' : \mathit{Star} \qquad y : \mathit{Seq} \xrightarrow{\star} y' : \mathit{Star}}{\mathit{seq}(x, y) \xrightarrow{\star} \mathit{star}(x', y')} \qquad \frac{x : \mathit{Seq} \xrightarrow{\star} x' : \mathit{Star}}{\mathit{chord}(x) \to x'}$$

Using them it is possible, e.g., to infer the one-step, unlabelled rewrite leading from $star(b, chord(seq(a, a)))$ to $star(b, star(b, b))$.

We shall exploit SOS style rules in § 4.1 and § 4.3 for modelling reconfigurations in UML4SOA and in software Modes.

4 Formal Support to Business and Architectural Design

We provide in this section a brief overview on the ADR-based formalisation of business and architectural design issues of UML4SOA and SRML.

4.1 UML4SOA Reconfiguration Profile

UML4SOA is a UML profile for designing service-oriented software, defined as a conservative extension of the UML2 metamodel, see Chapter 1-1 (UML Extensions for Service-Oriented Systems). Such a UML profile is the basis for the specification of a model-driven approach for the automated generation of service-oriented software through model transformations. UML4SOA uses extended internal structure and deployment diagrams. The extension for structure diagrams comprises service, service interface and service description. A component may publish several services specified as ports, which are described by service descriptions. Each service may contain a required and a provided interface containing operations. The orchestration of these services defines a new service. The extension for deployment diagrams is restricted to different types of communication paths between the nodes of a distributed system: permanent, temporary and on-the-fly.

UML4SOA profile aims at providing convenient mechanisms to model the inherent dynamic topologies of service-oriented systems: components join and leave the system, and connections are re-arranged. Such dynamic reconfigurations exhibit a number of beneficial features, but require a suitable mechanism to constrain the possible evolutions of system configurations and to avoid ill-formed configurations. In order to express such constraints on topologies, UML4SOA provides ingredients to specify architectural styles, and a methodology for modelling dynamic changes of configurations under architectural styles.

The main idea behind the ADR formalisation is that $\langle\langle$fragment$\rangle\rangle$-stereotyped components, i.e. configurations, are represented by ADR designs, while the architectural constraints imposed by UML4SOA concepts such as multiplicity or productions are captured by appropriate ADR types and design productions. UML4SOA reconfiguration rules specified as $\langle\langle$transformation$\rangle\rangle$ packages are represented by ADR rewrite rules. It is worth to recall that the main novel principles of the profile, i.e. style-consistent design-by-refinement and style-preserving, conditional reconfigurations are indeed the quintessence of ADR.

Figure 6 exemplifies how UML4SOA $\langle\langle$fragment$\rangle\rangle$ components can be mapped to ADR designs: $\langle\langle$service$\rangle\rangle$ ports are mapped to ADR nodes, while the port type determines the node type (e.g. UML types ChainingPort, CarAcccessPort and StationAcccessPort are represented by node types $\bullet$, $\circ$ and $\circledcirc$, respectively).

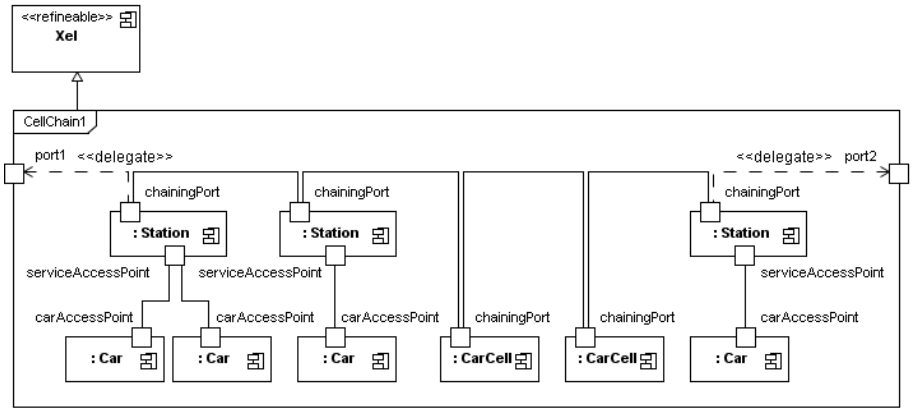


Fig. 6. A configuration of the *On Road Connectivity* scenario

Components are mapped to hyper-edges, where the component type determines the hyper-edge type.

The interface of the design is defined by the ports and the generalisation of the $\langle\langle$fragment$\rangle\rangle$ component. The ports of the $\langle\langle$fragment$\rangle\rangle$ define the set of interface nodes V_{L_d}, and each $\langle\langle$delegates$\rangle\rangle$ edge defines a maplet of the mapping i_d from interface to body nodes V_{R_d}. The type of the graph, as defined by the UML4SOA model, is determined by the generalisation of each $\langle\langle$fragment$\rangle\rangle$.

Modelling Architectural Styles in ADR. Refinable components and non-refinable components of UML4SOA specifications are respectively modelled by non-terminal and terminal edges in ADR. Internal structure diagrams and productions are the style definition mechanisms of UML4SOA; they are modelled by ADR design productions. Note however that some of the architectural constraints involved in class diagrams, such as multiplicities, cannot be expressed by type graphs directly. Indeed, type graphs do not impose any multiplicity constraint, i.e. they would amount to a UML $[0..*]$ multiplicity constraint. A suitable way to impose a multiplicity constraint in ADR is by means of design productions. For instance, in ADR the treatment of sets of cars in the UML4SOA specification via multiplicities is dealt with the design productions Car and Cars (see Fig. 7), which respectively allow to refine a generic set of cars as an empty set, a single car or the union of two other sets. In this way, UML4SOA productions are directly mapped into ADR design productions.

Modelling Reconfigurations under Architectural Styles. We just recall here that one of the advantages of ADR reconfigurations over other graph-based approaches is style-preservation, which is guaranteed by rewrites that do not change the overall type (they can actually change the type of certain sub-parts in the rule derivation of the overall reconfiguration).

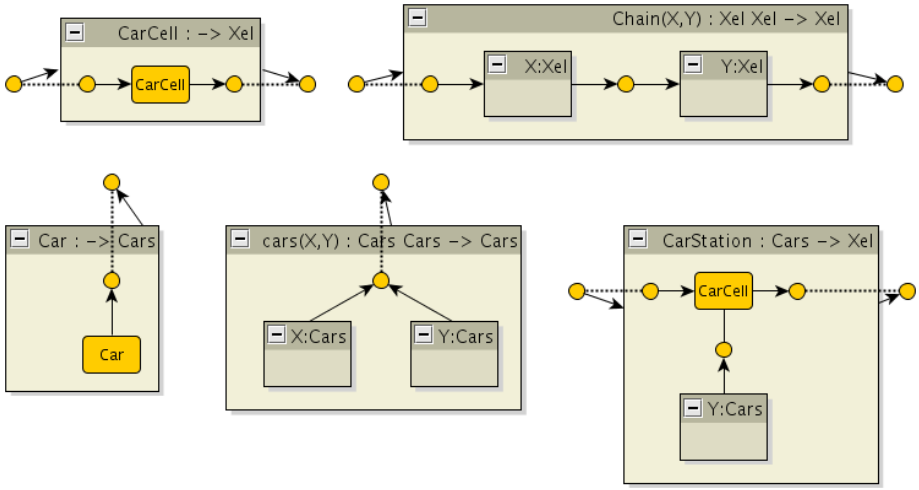


Fig. 7. Design productions for *On Road Connectivity* scenario

Translating UML4SOA reconfiguration rules to ADR in the general case is done by translating the precondition rules, the $\langle\langle \mathsf{transforms} \rangle\rangle$ left- and right-hand sides of the rule conclusion, and translating transformation labels into their respective counterparts in ADR. In this process, $\langle\langle \mathsf{pattern} \rangle\rangle$ components are translated to ADR designs by first producing ADR design graphs (replacing components with $[0..*]$ multiplicities by the corresponding non-terminal hyperedge, as done in the example with Cars) and then parsing the result using the ADR productions generated from the UML4SOA productions.

We show now a simple example of an ad-hoc network reconfiguration, which is modelled with inductive reconfiguration rules in SOS style. The base reconfiguration involves a single car:

$$\mathsf{CarToCell}: \ \mathsf{Car} \overset{\texttt{tocell}}{\longrightarrow} \mathsf{CarCell}$$

The inductive case we consider is when the union of two collections of cars is reconfigured as the concatenation of the respective reconfigured cells:

$$\mathsf{CarsToCellChain}: \ \frac{x_1 \overset{\texttt{tocell}}{\longrightarrow} x_1' \qquad\qquad x_2 \overset{\texttt{tocell}}{\longrightarrow} x_2'}{\mathsf{Cars}(x_1, x_2) \overset{\texttt{tocell}}{\longrightarrow} \mathsf{Chain}(x_1', x_2')}$$

Finally, the cell with the station shutting down is reconfigured by:

$$\mathsf{CellToChain}: \ \frac{x \overset{\texttt{tocell}}{\longrightarrow} x'}{\mathsf{CarStation}(x) \longrightarrow x'}$$

Obviously, types are not preserved by CarToCell and CarsToCellChain and thus the right- and left-hand sides of the rewriting rule cannot be applied in the same

contexts. Type changing allows for the modelling of reconfigurations that lead from one architectural style to another. The last rule CellToChain, instead, is given as a conditional term rewrite rule, where the premise is in its turn a rewrite rule requiring a collection of cars to become a chain cell, while the conclusion actually transforms a chain of cells into a chain of cells. The type is preserved and the silent label makes it applicable in any larger context (unlike style-changing rewrites labelled `tocell`).

4.2 SRML

In this section we provide a formalisation of some aspects of the SENSORIA Reference Modelling Language (SRML), see Chapter 1-2 (The SENSORIA Reference Modelling Language). SRML is inspired by the Service Component Architecture (SCA [11]). Roughly, it provides primitives for modelling composite services and activities whose business logic involves the orchestration of interactions among more elementary components and the invocation of services provided by external parties.

In [4] we presented a formalisation of the design and reconfiguration aspects of SRML based on ADR. The main idea was to define an ADR architectural style of *correct* SRML diagrams and set of ADR reconfiguration rules correctly modelling the internalisation of services that occurs in SRML, both at design-time (static module composition) and at run-time (dynamic service binding). After recalling the work in [4], this section mainly outlines a formal semantics of the behaviour of SRML specifications based on ADR rules in the form of Synchronised Hyperedge Replacement (SHR).

More precisely, given a SRML specification, first we exploit the translation given in [4] to derive a corresponding ADR design term that evaluates to a particular design, and then we consider the application of SHR rules directly over the design and not over the design term as in the case of reconfiguration.

Binding-time reconfigurations in SRML. We consider a scenario that involves an activity OnRoadRepair that takes place in a software system embedded in a vehicle to handle engine failures detected by a sensor. When the activity is triggered, the system determines the current location of the car by using a GPS device, searches for the closest garage that can ensure minimal levels of repair and call a tow truck, and contacts a car rental service near the garage.

Some architectural elements of SRML are drawn in Fig. 8 and include service modules, service components, wires and interfaces. A module is specified in terms of a number of entities and the way they are interconnected. For example, the activity module shown in Fig. 8 (top-left) involves the following software entities: OR (the orchestrator that coordinates the interactions with the external services) and IM (the component that manages the interactions with the driver). These entities are interconnected through wires, each of which defines an interaction protocol between two entities. Typically, wires deal with the heterogeneity of partners involved in the activity by performing data integration. The activity OnRoadRepair relies on a number of external services that will be discovered on

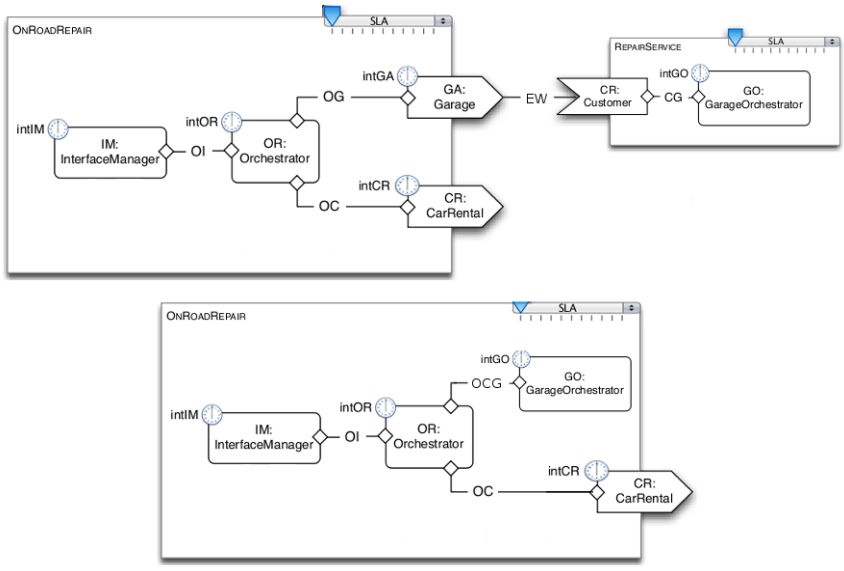


Fig. 8. An SRML diagram before (top) and after (bottom) composition.

the fly: (1) the service for booking a garage and calling a tow-truck, and (2) the service for booking a rental car. This dependency is made explicit through the *requires-interfaces* GA and CR, respectively. As illustrated, every activity module declares interfaces of various kind: one and only one *provides-interface* that binds the activity to the application that triggered its execution (e.g., CR in module RepairService), and a number of *requires-interfaces* (possibly none) that bind the activity to services that are procured externally when certain conditions become true (e.g., GA in module OnRoadRepair). Service modules such as RepairService in Fig. 8 provide a service to the external environment and can be dynamically discovered and invoked (instead of being launched directly by users).

The graphical notation of SRML is inspired by the traditional boxes-and-lines or component-and-connectors notations and elements are shaped as in SCA. The structural constraints, in turn, require modules to be interconnected via external wires such that one of the require interfaces of a module is connected to the provide interface of another one. Inside a module, components and interfaces are connected via internal wires. An SRML architecture is given at the highest level of abstraction by an *assembly of modules* with possibly some discovered but not yet bound service modules (i.e., they are still connected via external wires). Figure 8 (top) shows the architecture of our scenario with the service module OnRoadRepair, where one of the two required services (namely RepairService, corresponding to the interface GA) has been discovered and connected via an external wire (EW).

An example of a reconfiguration in SRML is the *composition* of (already discovered) interconnected modules into a single module. SRML provides a

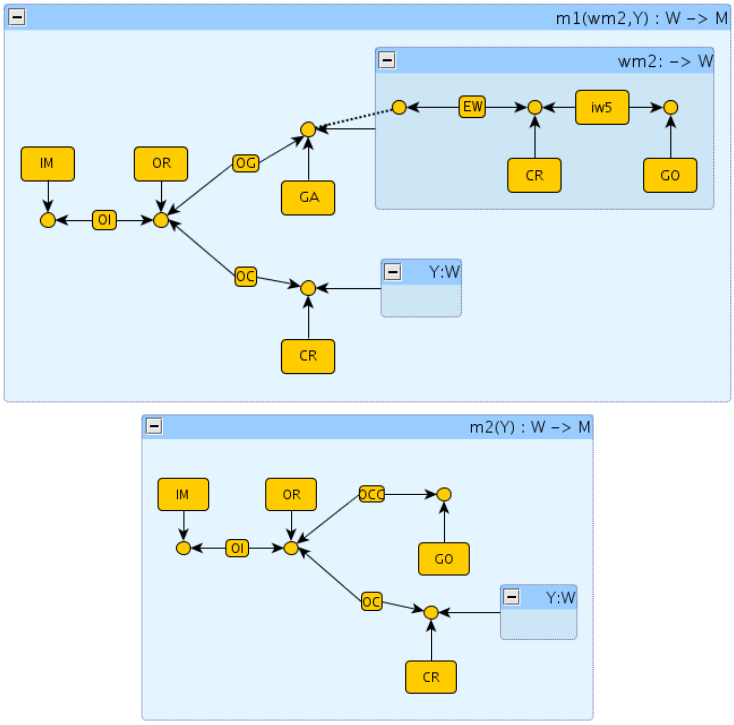


Fig. 9. ADR-view of SRML binding (top) and composition (bottom)

mechanism to achieve this static reconfiguration, by means of an algorithm that manipulates SRML specifications. As an example, the assembly of Fig. 8 (top) can be composed into the service module depicted in Fig. 8 (bottom), where the wire OCG is derived according to certain composition rules. Such reconfigurations require a proof of correctness w.r.t. style preservation.

The formalization of SRML in ADR given in [4] introduces suitable architectural elements for representing service components, internal wires, external wires, provide interface specifications, require interface specifications as terminal edges and activity modules, service modules, wrapped modules and their bodies as non-terminal edges. While the interested reader is referred to [4] for full details, we sketch here the basic idea of the modelling. Figure 9 (top) shows the ADR service module and wrapped module corresponding to the SRML diagram in Fig. 8 (top). When a binding is performed, then the wrapped module is plugged-in the service module. SRML composition is then realised via conditional rewrite rules that synthesize a suitable wiring out of the specifications of the require interface and provide interface of the composed modules, according to the internal and external wiring connecting them. The result is shown in Fig. 9 (bottom), where the internal wiring OCG is synthesized out of OG, GA, EW, CR, and CG.

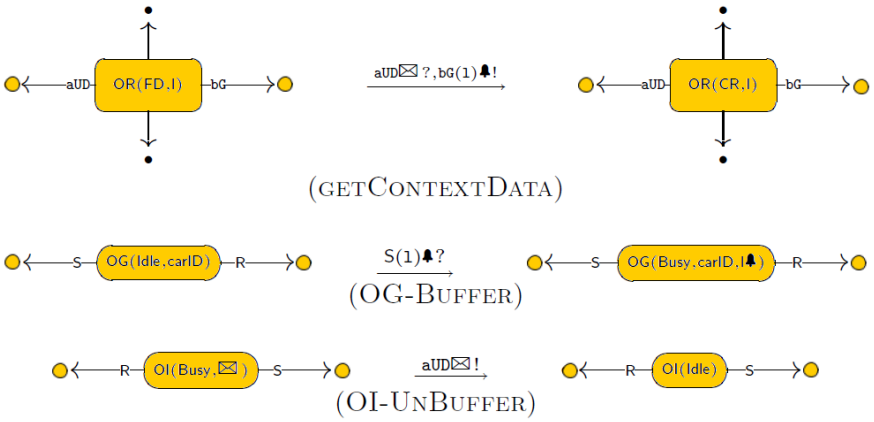


Fig. 10. Some operational rules modelling SRML behaviour

Operational Semantics for SRML. To illustrate the SHR modelling of ordinary computational aspects of SRML, let us consider the automotive scenario where the service execution is at the point in which a garage service has been discovered and bound (see Fig. 9 (bottom). The new configuration includes the components and the top/bottom layer interfaces of OnRoadRepair and those of RepairService. The representation of a configuration does not include the external provide/require interfaces because external interfaces do not describe an executable process. Note that Fig. 9 presents a simplified form of the actual graphs, where we just decorate the edges with the component and connector names, while additional information such as the type and state of components or the name of interactions are abstracted away. We shall explicitly represent some of those details when needed.

We continue our illustration with the representation of a transition getContextData of the business role Orchestrator (to retrieves the data of the driver, for example from InterfaceManager) as a rule in SHR style (see Fig. 10): the transition label witnesses that the component is ready to receive (?) a reply event (✉) on interaction askUsrDetails (aUD) and simultaneously send (!) an initiation event (♟) on interaction bookGarage (bG), while changing state from FD (FAILURE_DETECTED) to CR (CONTEXT_RECEIVED).

Note that the graphical representation of the rules is simplified for the sake of readability. For instance, not all interactions (represented here as labelled tentacles) are drawn. We put state information (local state, variables) as edge labels in tuples, and we neglect some of the parameters of the interactions. The type of interactions is drawn using different node types, so to forbid any mismatched connection: e.g. we use • for `s&r` and ◦ for `r&s`.

Observe that the various events involved in the transition (i.e. the trigger and the events to be sent) are put on the rule label. In this particular case the transition guard is implicit in the edge label (containing the state of the

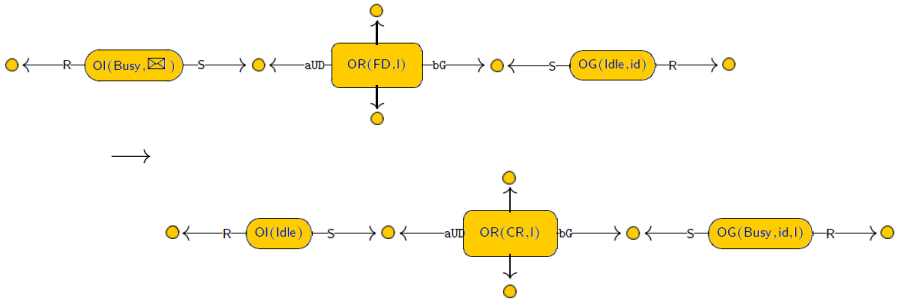
component). In general, guards are modelled as side conditions. Effects, instead, are just the resulting edge label. Transitions are thus given an atomic semantics: trigger, effects and sends are executed simultaneously.

SHR rules can be synchronised together using different styles. In our case, we follow the Milner style, where rule synchronisation requires an action being sent on a node to be synchronised with a corresponding co-action on the same node. Note that since a rule can involve actions of more than one node among those attached to an edge, multiple synchronisations are possible. In our example, rule getContextData can only be fired if the surrounding connectors are able to perform the corresponding co-actions.

For instance, in the surroundings of the orchestrator component we find two connectors involved in transition getContextData, namely OG and OI. The binding is such that the bookGarage and askUserDetails interactions of the orchestrator are assigned to roles S of connectors OG and IM, respectively.

All the rules that we need are such that the left-hand side and the right-hand side differ only in the label of the edge, meaning that only the state is changed, but neither the interface nor its bindings, i.e. the type is preserved. This condition is enough to guarantee that the overall design term is not affected by the application of operational rules. Part of the behaviour of the connectors include rules to buffer and unbuffer events (see Fig. 10). For instance, consider the rule to buffer the init event: it changes the state of the connector from Idle to Busy (buffered), and the received parameter I is enqueued (it is just part of the edge label).

A straightforward ternary synchronisation allow us to derive the rewrite



out from rules getContextData, IO-UnBuffer and OG-Buffer. Such derivation synchronises a component with two of its attached connectors. In some cases, connectors might synchronise with both parties such that complex synchronisations involving multiple components and connectors are possible.

4.3 Software Modes

In this section, we consider the ADR modelling of Software Architecture Modes, as presented in [9]. There, a mode abstracts a specific set of services that must interact for the completion of a specific subsystem task. Modes are first-class architectural ingredients that govern the architecural constraints and reconfiguration mechanisms of a software system.

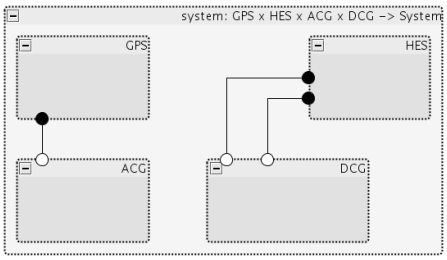


Fig. 11. The RPS subsytem production

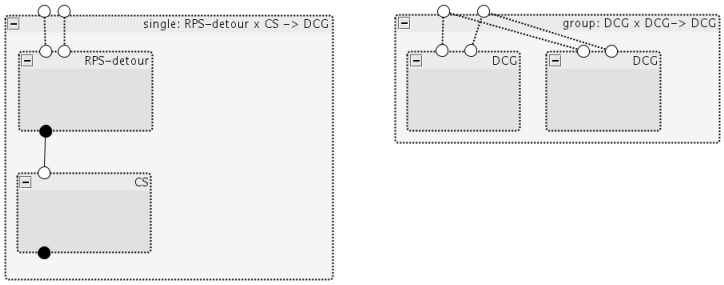


Fig. 12. Building detour convoy groups

Service Modes extend the concept of Software Architecture Modes with that of behaviour and policy specifications for service adaptation and dynamic reconfiguration. Service Modes are specifically aimed at specifying "operational adaptation" for a service-oriented system. They are based upon an evolving set of scenarios describing service component architecture, behaviour and events which trigger reconfiguration, whilst upholding quiescence in service operation.

We illustrate the ADR formalisation of modes with a road assistance scenario of the automotive case study. In the scenario, cars are equipped with navigation systems connected to a road assistance service platform. The focus of our example is on the Route Planning Subsystem (RPS) which is in charge of providing guiding indications to the driver. The RPS has three modes of operation: *Autonomous*, i.e. connected to the GPS to establish the route, *Convoy*, i.e. following another car, *Detour*, i.e. following indications from the Highway Emergency system (HES).

The components that form the RPS subsystem are the Global Positioning System (GPS) the Highway Emergency System (HES), the Route Planning Subsystem (RPS), the Planner (P), the User Interface (UI), and the User Prompt (UP). Some of these (sub)components can be in a different mode. For instance, P can be in modes P-*Master* or P-*Slave*, while the user interface UI and prompt UP can be in modes *Enable* or *Disable*.

Informally, the main architectural constraints require that an RPS must be composed of a P, an UI and an UP. The mode of the RPS depends on the modes

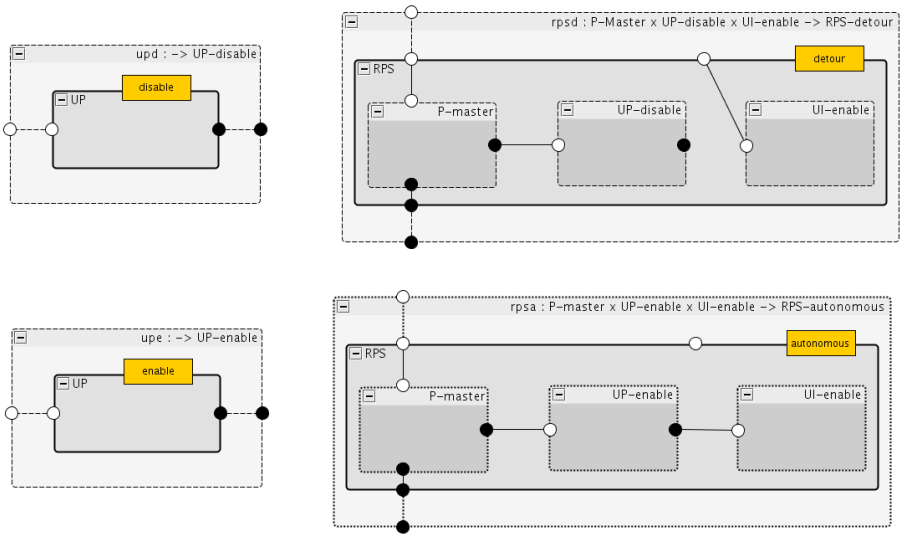


Fig. 13. Building RPSs in various modes

of its constituents. Convoys are formed by a leader RPS followed by RPSs in convoy mode. Leaders can be either in autonomous or detour mode. Autonomous RPSs are connected to the GPS. Detour RPSs are connected to the HES.

The main idea of the formalisation with ADR is to encode a software class as a type T and its various modes as subsorts T-M of T. Additional types can stand for complex constructions (shapes, styles, patterns) such as sequences, sets or trees. Such types can be used to define composition operations that determine the valid configurations. Mode types play a relevant role in reconfigurations.

Structural constraints are captured by a set of ADR design productions that build conformant configurations. For instance, we use productions to build systems (see Fig. 11) and Detour Convoy Groups (DCG) either as a single RPS in detour mode followed by a sequence of RPSs in convoy mode (Fig. 12, left) or two DCGs (Fig. 12, right). Dotted lines between ports denote interface exposure and not binding of actual ports (denoted by straight lines). The figures are drawn according to Darwin notation, but the correspondence with our designs is immediate (ports are nodes, boxes are edges and the flattening axioms apply if the contour of a box is dotted), except for the fact that each port binding actually corresponds to a terminal edge b. Whenever necessary (i.e., not implied by flattening axioms), we assume the expected properties for the operations to hold (commutativity, associativity, etc.). Similar operations are used to build Autonomous Convoy Groups (ACG): they can be built from an RPS in autonomous mode (`RPS-autonomous`) followed by a sequence of RPSs in convoy mode (`CS`) or from two ACGs. Productions are also used to build a CS as single RPS in convoy mode or as the concatenation of CSs.

There are also productions to build RPSs in various modes (see Fig. 13 for the productions needed in the examples that concludes this section). An RPS in autonomous mode is a composite component with a P in master mode, an UP in enable mode and an UI in enabled mode. An RPS in detour mode is a composite component with a P in master mode, an UP in disable mode and an UI in enabled mode. An RPS in convoy mode is a composite component with a P in slave mode, an UP in enable mode and an UI in enabled mode. Basically, each pair (component type,mode) has a corresponding constructor and ADR type. Constructors for HES and GPS are similar.

The allowed RPS reconfigurations are from Detour to Autonomous mode (and back) and from Convoy to Autonomous and Detour mode (and back). However, RPS components should not reconfigure independently: their constituents and contexts should reconfigure such that architectural constraints are respected.

For instance, a single RPS in detour mode is moved from the group of RPSs in detour mode to the group of convoys if that RPS reconfigures itself from mode detour to mode autonomous, i.e. if its UP reconfigures from disable to enable mode. Exploiting our design algebra, the corresponding rules can be written as the following (labelled) conditional term rewrite rules:

$$\frac{}{\mathrm{upd} \xrightarrow{d2e} \mathrm{upe}} \qquad \frac{Xupd \xrightarrow{d2e} Xupe}{\mathrm{rpsd}(Xpm, Xupd, Xuie) \xrightarrow{d2a} \mathrm{rpsa}(Xpm, Xupe, Xuie)}$$

$$\frac{Xrpsd \xrightarrow{d2a} Xrpsa}{\begin{array}{l}\mathrm{system}(Xgps, Xhes, Xacg, \mathrm{group}(\mathrm{single}(Xrpsd, Xcs), Xdcg)) \\ \qquad \longrightarrow \mathrm{system}(Xgps, Xhes, \mathrm{group}(Xacg, \mathrm{single}(Xrpsd, Xcs)), Xdcg)\end{array}}$$

5 Conclusion

This chapter collects results from [1, 3–6]. In particular, we have provided an overview of main ADR features and the ADR representation of UML4SOA, SRML and Software Modes; the latter being original to this contribution. The formal semantics prepares the ground towards tool support for analysis and verification from the very early stages of modeling. Thus, ADR offers a comprehensive and pragmatic yet theoretically well founded approach to software engineering for service-oriented systems. Our current efforts are aimed at completing our tool support. First, by automatising the translation of high-level specifications in the considered languages, possibly by means of Maude-supported, MOF-based model transformations. Second, by upgrading the prototypical implementation of ADR into a tool that can be used to formally analyse ADR models either specified directly or transformed from other models.

In fact, ADR specifications can be exploited to perform formal specification and verification based on techniques developed, e.g. for term rewrite systems and graph transformation systems. For instance, ADR specifications can be encoded into Rewriting Logic [10] and benefit from Maude's built-in tools.

References

1. R. Bruni, M. Hölzl, N. Koch, A. Lluch Lafuente, P. Mayer, U. Montanari, A. Schroeder, and M. Wirsing. A service-oriented UML profile with formal support. In *ICSOC/ServiceWave'09*, volume 5900 of *Lecture Notes in Computer Science*, pages 455–469. Springer Verlag, 2009.
2. R. Bruni and A. Lluch Lafuente. Ten virtues of structured graphs. In *Proceedings of the 8th International Workhshop on Graph Transformation and Visual Modeling Technique (GT-VMT'09)*, volume 18 of *Electronic Communications of the EASST*. EASST, 2009.
3. R. Bruni, A. Lluch Lafuente, and U. Montanari. Hierarchical design rewriting with Maude. In G. Rosu, editor, *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, volume 238 (3) of *Electronic Notes in Theoretical Computer Science*, pages 45–62. Elsevier, 2009.
4. R. Bruni, A. Lluch Lafuente, U. Montanari, and Emilio Tuosto. Service-oriented architectural design. In *Proceedings of the 2nd International Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *Lecture Notes in Computer Science*, pages 186–203. Springer Verlag, 2007.
5. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 94:161–180, February 2008.
6. A. Bucchiarone, R. Bruni, S. Gnesi, and A. Lluch Lafuente. Graph-Based Design and Analysis of Dynamic Software Architectures. In *Concurrency, Graph and Models. Festschrift in honor of Ugo Montanari*, volume 5065 of *Lecture Notes in Computer Science*, pages 37–56. Springer Verlag, 2008.
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 163–246. World Scientific, 1997.
8. 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, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 22–43. Springer, 2006.
9. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *Proceedings of the 3rd European Workshop on Software Architecture (EWSA'06)*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer Verlag, 2006.
10. J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
11. Service Component Architecture. `http://osoa.org`.
12. M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. In *Prentice Hall, NJ. USA*, 1996.