A service-oriented UML profile with formal support ⋆

Roberto Bruni[1], Matthias Hölzl[3], Nora Koch[2,3], Alberto Lluch Lafuente[1], Philip Mayer[3], Ugo Montanari[1], Andreas Schroeder[3], and Martin Wirsing[2]

[1]University of Pisa [2]Cirquent GmbH [3]Ludwig-Maximilians-Universität München

Abstract. We present a UML Profile for the description of service oriented applications. The profile focuses on style-based design and reconfiguration aspects at the architectural level. Moreover, it has formal support in terms of an approach called Architectural Design Rewriting, which enables formal analysis of the UML specifications. We show how our prototypical implementation can be used to analyse and verify properties of a service oriented application.

1 Introduction

Service-oriented computing is a paradigm centered around the notion of service: autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. However, services are still developed in a poorly systematic, ad-hoc way. Full fledged theoretical foundations are missing, but they are urgently needed for achieving trusted interoperability, predictable compositionality, and for guaranteeing security, correctness, and appropriate resource usage.

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 the main research activity of SENSORIA.

In this paper we present recent efforts within SENSORIA aimed to develop high-level modelling languages with strong formal support. More precisely, we present a novel extension of UML4SOA, our UML2 profile and define here its formal semantics. The presentation is illustrated on a simple example taken from the automotive domain (§ 2).

UML4SOA is an extension of UML2, the *lingua franca* of object-oriented software analysis and design. UML4SOA enhances UML2 with concepts for modelling structural and behavioural aspects of services. In § 3 we present for the first time an extension of the profile to support architectural styles [23] and dynamic reconfiguration.

The formal semantics of the extension is defined in § 4 in terms of *Architectural Design Rewriting* (ADR) [7], an approach for style-based design and reconfiguration of software architectures based on graphs and term rewriting, which supports both style-preserving and style-changing reconfiguration of service-oriented systems. In § 5 we show how the ADR formalisation and its prototypical implementation [4], can be used to analyse and specify properties of UML4SOA specifications.

⋆ This work has been supported by the EU FET-GC2 IP project SENSORIA, IST-2005-016004.

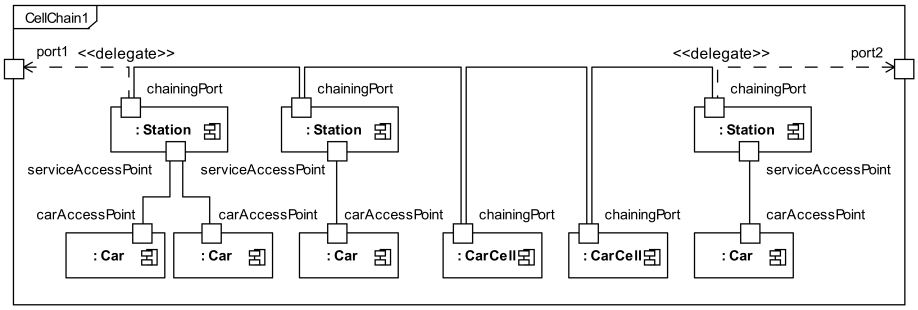

Fig. 1. Structure diagram of a configuration of the *On road connection* scenario

2 Running Example

We illustrate our approach by exploiting the *On Road Connectivity* scenario from a SENSORIA case study where a road assistance group of services support car drivers activities. In the scenario, cars access wireless services via stations that are situated along a road. We use a UML component to represent a configuration of such a system. Figure 1 shows the white-box view of a system as a component (top-left box) that contains other components (nested boxes) as parts of its internal assembly. A car is connected to the service access point of a station, which can be shared with other cars that are attached to the same station. A station and its accessing cars form a *cell*, which is dynamically reconfigurable, in the sense that cars can move away from the range of the station of their current cell and enter the range of another cell. A handover protocol permits cars to migrate to adjacent cells as in standard cellular networks.

Stations, in addition to the service access point, use two other communication ports that we call chaining ports. Such ports are used to link cells in larger *cell chains*. Stations can shut down, in which case their orphan cars are connected to other stations. This is tackled by appropriate system reconfigurations. We shall consider a shut down situation in which orphan cars switch from their normal mode of operation to a cell mode, in which case they become standalone ad-hoc stations (see the CarCells in Fig. 1).

3 UML4SOA Extension

UML4SOA [17] is a UML profile for designing service-oriented software. UML4SOA is defined as a conservative extension of the UML2 metamodel. 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 (c.f. [18]). UML4SOA uses extended internal structure and deployment diagrams. The extension for structure diagrams comprises service, service interface and service description [17]. 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 [25].

When modelling service oriented applications with our UML4SOA profile we observed the need for convenient mechanisms to model the inherent dynamic topologies of such applications: components join and leave the system, and connections are rearranged. 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, it is common practice to use *architectural styles* [23], i.e. sets of rules specifying the legal constituents of a system configuration and the permitted interconnections between them. Unfortunately, UML offers a limited and unsatisfactory support for architectural styles. We propose a novel extension of UML4SOA to remedy this. In addition, we provide a methodology for modelling dynamic changes of configurations under architectural styles.

3.1 The UML4SOA Reconfigurations Profile

We present our UML4SOA extension to draw easily understandable diagrams for architectural styles and reconfigurations. UML4SOA models services with ports. For an enhanced readability we will omit in this paper the «service» stereotype on ports, as all ports in the following represent services. Service providers are represented by components, while (one-to-one) connectors are used to model service references.

Modelling System Configurations. We model system configurations with internal structure diagrams. Using such diagrams allows us to model services as ports of service providers. Internal structure diagrams are used to depict the interconnections of components contained within an architectural entity, together their names, types, and multiplicities.

To model system configurations, we introduce the stereotype «fragment»(used in e.g. Chain), which may be applied to components and requires that all component fields (the elements pictured in the internal structure diagram) are typed with component or connector types, and interpret unspecified multiplicities as one. We constrain «fragment» components by forbidding the use of range or $*$ multiplicities. Fig. 1 shows a sample system configuration of our scenario.

In internal structure diagrams, ports of the container classifier may be drawn on the border of the diagram; they may be connected to internal elements by «delegates» edges, stating that the port of the container component is a proxy an internal port. This allows the definition of named docking points to internal components, to which other components may be glued to. For «fragment» components, we require that all external ports must be a delegate of a port of its contained components. As the name implies, a «fragment» represents a fragment of a system configuration that can be plugged together with other fragments. A complete system configuration is then modelled as a fragment without external ports.

Modelling Architectural Styles. UML internal structure diagrams provide a set of features to specify architectural styles. Indeed, such diagrams describe the static structure

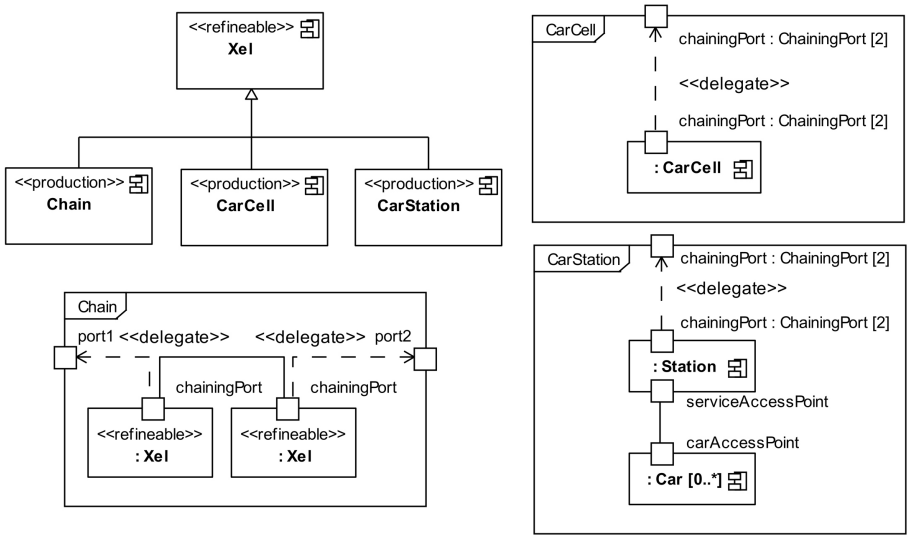

Fig. 2. Architectural style productions for the *On Road Connectivity* scenario

on the level of types, and allow to constrain multiplicities as well as interconnections between contained components. However, the service engineer is forced to model all alternatives allowed by a specific architectural style within one single diagram because of two reasons: first, there is no possibility to define abstract UML components, and second, using subtype polymorphism would introduce arbitrary many artificial composite components in the architecture. We believe that such mechanisms are not enough for a convenient specification of styles.

Our approach, instead, is based on a straightforward extension of the modelling of fragments with two modifications. First, the constraints on multiplicities are removed. Secondly, to define architectural styles in an inductive manner with composeable patterns, by using «refineable» components instead of concrete ones. Components used to define architectural style patterns are tagged with the stereotype «production». The non-terminal components marked with «refineable» may be replaced by any specializing «production» pattern (cf. Fig 2 for an example). In our scenario, for example, the «production» Chain contains two occurrences of «refineable» Xel components, which may be replaced by Chain, CarCell, and CarStation, as they all specialize the «refineable» Xel.

The «production» patterns define the legal wirings between components, and represent the basic building blocks of an architectural style. An architectural style is represented by a set of «production» patterns in the sense that every legal configuration must be produceable by applying the production patterns of the architectural style, replacing «refineable» by specializing «production» patterns.

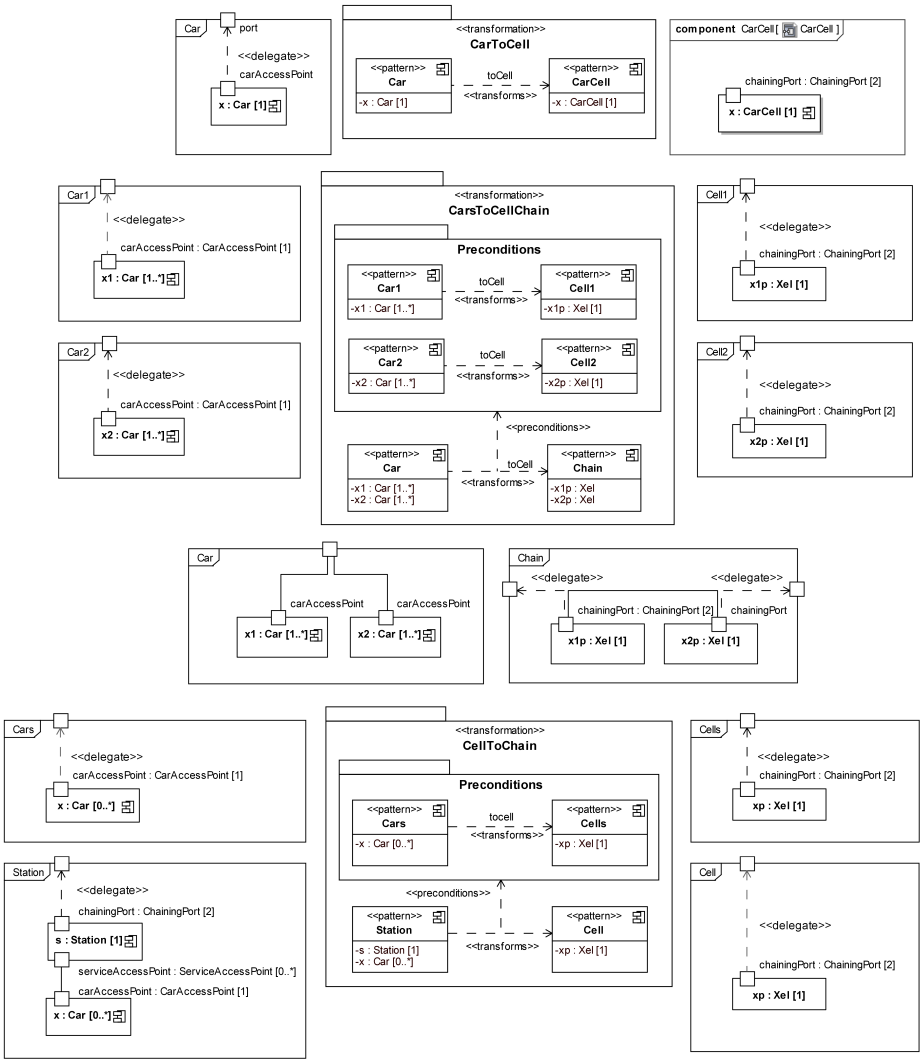

Fig. 3. Ad-hoc network reconfiguration rules for the *On Road Connectivity* scenario

Modeling Reconfigurations under Architectural Styles. Reconfiguration rules are defined as «transformation» packages having two «pattern» stereotyped components with internal structure diagrams (a left hand side and right hand side pattern), linked with a «transforms» edge. The name of fields in «pattern» components is interpreted as variable names and the enclosing «transformation» package as their scope, hence allowing to share variables among left hand side and right hand side patterns.

Often enough, reconfiguration rules depend on complex or non-local conditions. Consider for example the shut-down of a connection station: The connected cars should

form an ad-hoc network chain which will be connected to the neighbouring operating stations. Having only simple rules at hand, one would have to write one rule for each possible number of cars to be reconfigured. Using recursive rules, that is to say, using application conditions as in conditional rewrite frameworks, allows us to model the reconfiguration of arbitrarily many cars to a linear ad-hoc network, as in Fig. 3.

As can be seen from the diagram, one stereotype was introduced, «preconditions», which is attached to a dependency edge and points to a package containing the reconfiguration preconditions. The scope of variables in precondition patterns is again the enclosing «transformation» package, hence allowing to carry over transformation results from the preconditions to the actually performed reconfiguration step. In this way, a complex reconfiguration involving arbitrary many components may be modelled using simple and local reconfigurations.

One major challenge when modelling dynamic reconfigurations is to guarantee that the constraints of the architectural style are not violated. The benefit of our approach is that style preservation is ensured just by having the same types in the left- and right-hand sides of reconfiguration rules. This is of course only possible for those styles that can be inductively characterised by types. This includes common topologies such as trees or rings, but lets other, like regular grids, aside. Basically, the kind of structural constraints one model with our approach amounts to those sets of graphs characterisable with hyper-edge graph grammars [14].

4 Formal Support

ADR [7] models systems by *designs*: a kind of typed, interfaced graphs whose inner items represent the system components and their interconnections and whose interfaces express their overall types and connection capabilities. Domains of *valid* systems, (e.g. those compliant to styles) are defined in an inductive way by means of *design productions* (i.e. valid system compositions), which define an algebra of *design terms*, each encoding the structure of the system and providing a proof of validity (e.g. style conformance). Reconfiguration and behaviour are given as term rewrite rules acting over design terms rather than over designs. This enables the flexible definition of valid (e.g. style preserving) reconfigurations. A prototypical implementation is described in [4], where we also extended the approach to the treatment of hierarchical graphs and explained how to write system specifications and how to analyse them. ADR has been already validated over heterogeneous models such as network topologies, architectural styles and modelling languages. For instance, in [5] we presented a formalisation of reconfiguration aspects of SENSORIA's business-level service modelling language.

4.1 ADR Semantics for the UML4SOA Reconfiguration Profile

This section describes, in an illustrative manner, the ADR formal semantics of the above presented UML4SOA profile. The main idea behind the formalisation is that «fragment»-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.

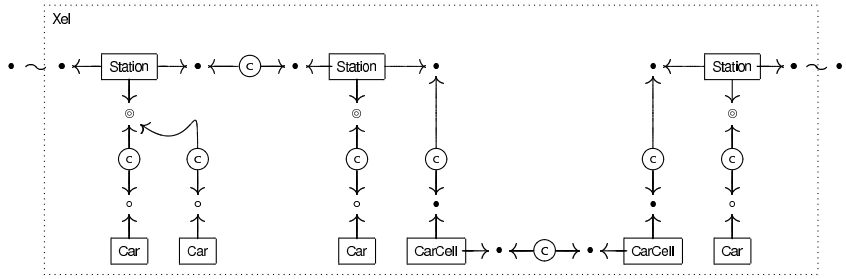

Fig. 4. The *On Road Connectivity* scenario of Fig. 1 as an ADR design

UML4SOA reconfiguration rules specified as «transformation» 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.

Modelling System Configurations in ADR. A design is a graph-based structure. 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 T-typed.

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 so-called *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 total function that maps interface nodes to body nodes.

The visual representation of a design (see Fig. 4) depicts the interface as a dotted box with its type written in its top-left corner. The body is depicted inside the dotted box. Edges are represented as boxes (possibly rounded), tentacles as arrows (their order is given by their orientation) and nodes as small circles. The nodes being exposed on the interface are denoted by waved lines.

Fig. 4 exemplifies how UML4SOA «fragment» components of Fig. 1 can be mapped to ADR designs: «service» 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 •, ∘ and ◎, respectively). Components are mapped to hyper-edges, where the component type determines the hyper-edge type. UML connectors are mapped to binary edges of a predefined type c.

The interface of the design is defined by the ports and the generalisation of the «fragment» component. The ports of the «fragment» define the set of interface nodes V_{L_d}, and each «delegates» edge defines a maplet of the mapping i_d from interface to body nodes V_{R_d}. The type of the so-produced graph, as defined by the UML4SOA model, is determined by the generalisation of each «fragment» (Xel in Fig. 1).

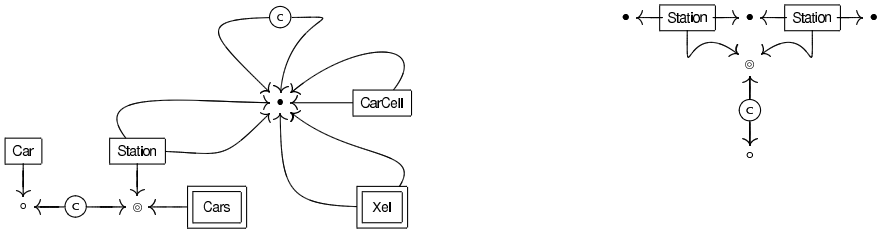

Fig. 5. Type graph for *On Road Connectivity* scenario (left) and a correctly typed graph(right)

Modelling Architectural Styles in ADR. The distinction between refinable components and non-refinable components 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).

The style definition mechanisms of UML4SOA, i.e. internal structure diagrams and productions, are modelled by ADR type graphs and by design productions. Note however that some of the architectural constraints involved in class diagrams such as multiplicities cannot be directly mimicked by type graphs. Instead, they are dealt with at the level of design productions.

Consider the type graph on the left on Fig. 5. It is easy to see that each edge corresponds to a component (Car, Station, CarCell) or connector type (the overloaded symbol c). Non-terminal edges (Cars, Xel) are distinguished by their double border. In general, the type graph is obtained from the whole UML4SOA specification: adding terminal edges for each non-refinable component type, non-terminal edges for each refinable component type, nodes for port types, tentacles for component ports and edges for the connectors.

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, the treatment of sets of cars in the UML4SOA specification via multiplicities is dealt in ADR with the design productions NoCar, Car and Cars (see Fig. 6), 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. For instance, in absence of production NoCar the multiplicity constraint would be [1..∗]. We remark that productions allow to refine the architectural constraints imposed by a type graph alone. For instance, the graph on the right of Fig. 5 is well-typed but is not generated by our productions.

Technically, a design production is very much like a design but with an order on the non-terminal edges of the 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.

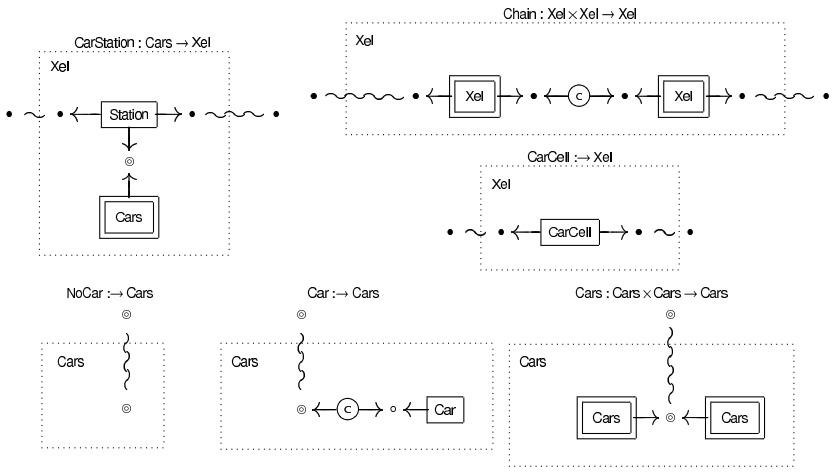

Fig. 6. Design productions for *On Road Connectivity* scenario

The functional type $A_1 \times A_2 \times \ldots \times A_{n_p} \to A_p$ associated to a production p means that p can be considered a function that when applied to a tuple $\langle d_1, d_2, \ldots, d_{n_p} \rangle$ of designs of types $A_1, A_2, \ldots A_{n_p}$, respectively, returns a design $d = p(d_1, d_2, \ldots, d_{n_p})$ of type A_p. The definition is obvious: $d = (L_p, R_d, i_p)$, where R_d is obtained from R_p by replacing edge e_k in it with graph R_{d_k} respecting the tentacle function i_{d_k}, $k = 1, \ldots, n_p$.

This view corresponds to a bottom-up design development: a design is constructed by putting together some component designs. However, the dual view is also possible: a production can be seen as a refinement of an abstract component of type A as an assembly of concrete and abstract components, the latter being of type $A_1, A_2, \ldots A_{n_p}$.

Modelling Reconfigurations under Architectural Styles. UML4SOA transformations are represented by ADR rewrite rules, which are given in different flavours: e.g. in Meseguer's rewriting logic [19] or Plotkin's structural operational semantics (SOS) [22]. 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).

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

The ad-hoc network reconfiguration is tackled by using inductive reconfiguration rules in SOS style. The base reconfiguration involves a single car (see Fig. 7):

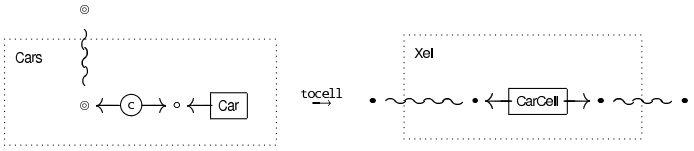

Fig. 7. Reconfiguration CarToCell

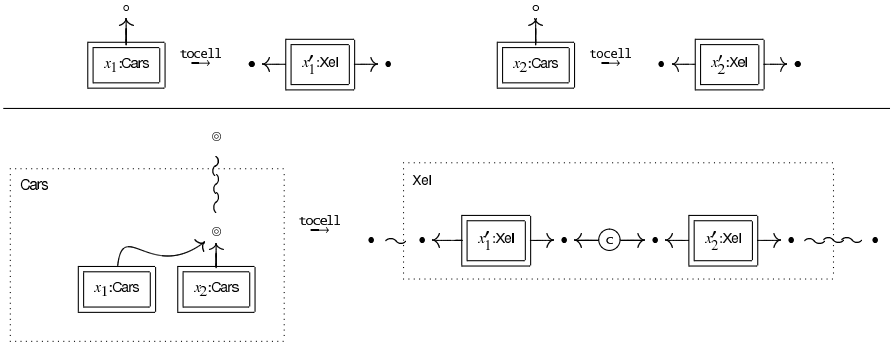

Fig. 8. Reconfiguration CarsToCellChain

$$\mathsf{CarToCell}: \ \mathsf{Car} \xrightarrow{\texttt{tocell}} \mathsf{CarCell}$$

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

$$\mathsf{CarsToCellChain}: \ \frac{x_1 \xrightarrow{\texttt{tocell}} x'_1 \qquad\qquad x_2 \xrightarrow{\texttt{tocell}} x'_2}{\mathsf{Cars}(x_1, x_2) \xrightarrow{\texttt{tocell}} \mathsf{Chain}(x'_1, x'_2)}$$

The cell with the station shutting down is reconfigured by the rule (see Fig. 9):

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

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. However, this is not what we want in this example and thus labelled rules are given in SOS style. The last rule CellToChain, instead, is given as a conditional term rewrite rule, where the premise in its turn a rewrite rulle 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`).

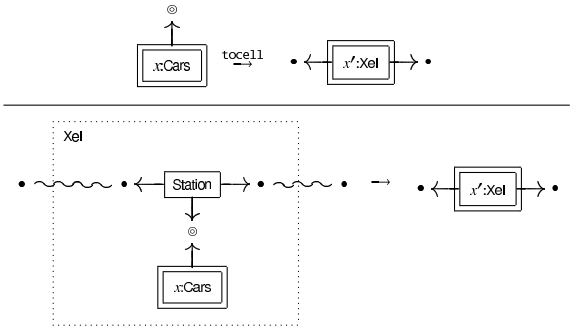

Fig. 9. Reconfiguration CellToChain

5 Analysis and Verification

This section emphasises the benefit of having a formal semantics for our UML profile by describing how the use of our implementation of ADR [4] can be used to analyse and verify properties of UML4SOA specifications. We remark that the translation of UML4SOA specifications to ADR specifications has not been implemented yet. On the other hand, a prototypical implementation of ADR is available for download [4]. Nevertheless, we offer sufficient evidence of the potential of our approach as a helpful support for software architects.

Analysing Styles. After a first development of a UML4SOA specification, a software architect might wonder whether the defined architectural styles enjoy some properties he desires. For instance, in our example scenario one could be interested in stating that no Xel production builds a configuration in which the left and right chaining ports are disconnected. It is easy to see that this property trivially holds in our example. However, as scenarios become complicated such properties become more subtle. Note that due to the inductive definition of styles it holds that if all Xel productions satisfy the property, then the property holds for any possible Xel configuration. This is indeed the case of the example property: all Xels are un-broken chains.

Our implementation includes a graph logic (Courcelle's MSO [11]) that allow us to reason about the structure of a graph. Such mechanism can be used to analyse the structure of UML4SOA productions by analysing the underlying graphs. The above example for instance is a well known property of graph connectivity which can be expressed in MSO by $\forall X.((\forall x, y(y \in X \wedge z \in R(y,z) \rightarrow z \in X \wedge \forall y.R(a,y) \rightarrow y \in X)) \rightarrow b \in X)$, where X is a set of nodes, x,y and z are nodes, R abbreviates the existence of an edge between two nodes, and a, b are shorthands for the left and right hand-side chaining ports. In words, we look for all sets of nodes X closed under the transitive closure of the relation R of direct adjacency and containing all nodes adjacent to b. If all such sets contain b too, then a and b reachable from each other. The above formula holds for all body graphs of Xel productions. We can of course, write abbreviations for such formulae to construct a sort of library of structural properties.

Checking Style Conformance. Once the software architect is confident with the style he has designed, he might be interested in re-using some of his old specifications. After manually applying some cosmetics on the types of diagrams and other entities, he might want to know whether the resulting instance is consistent with the style. Roughly, he needs a correct parse in terms of the productions. This is supported by a mechanism that roughly generates design terms and checks if the resulting designs are isomorphic to the configuration under analysis. For instance, one can show that the configuration in Fig. 4 is style conformant by finding the parsing that we mentioned in Section 3.1. A counterexample can be found in Fig. 5 (right).

Finding Configurations Automatically. Model finding is the problem of analysing the state space of all possible instances of his architectural style. Such analysis serves as a computer-aided design process or as a debugging method to find out inconsistencies in models, styles or properties. Our model finding system is based on two mechanisms: one to generate a state space of models and one to explore it. In our approach we can define a rewrite theory that simulates a design-by-refinement process, roughly consisting of the context-free graph grammar obtained by a left-to-right reading of design productions. In order to explore such state spaces, we can use various mechanisms of Maude. Typically, the space of configurations is infinite and bounds are required. For instance, we can use search strategies to find configurations with at 4 cars, 3 stations and 2 car cells and we obtain, among others, the design of Fig. 4.

Analysing Static Aspects of Configurations. Now that the software architect has adapted some of his old designs and possibly built new ones, he might want to reason about them. Returning to our example, we might wonder if a configuration has at least n cars or is free of cars in cell mode. Recall that our configurations have two levels: the more abstract level of design terms and the more detailed level of the diagrams. We can expect dual mechanisms for stating structural aspects. Indeed, we saw above that the properties of diagrams are supported by graph logics. Similarly, properties can be stated at the level of design terms. Our ADR implementation does this by means of spatial logics, the natural and structured way to reason about term-like specifications. Basically, for each design production f used to compose designs the logic incorporates a spatial operator f-so to decompose a design. For instance, formula Chain-so(ϕ_1,ϕ_2) is satisfied by all those designs of the form Chain(x1,x2), where design x1 satisfies formula ϕ_1 and design x2 satisfies formula ϕ_2.

Consider the property that states a collection of cars has at least n cars. We can inductively define it as follows: For n equal to zero the formula always holds. For $n+1$ the formula holds whenever the term is decomposable as the composition via operation Cars of one car (Car-so) and a term with at least n cars. Using such formulas we can for instance check that the design of Fig. 4 satisfies the property stating that each station has at least one car and violates the property stating that each station has at least two cars.

Analysing Dynamic Instances of Configurations. At this point the software architect might be satisfied with the structural properties enjoyed by his configurations. The modelled application, however, has a dynamic architecture with various reconfiguration

rules such as those we use in our running example. Can he express that some property is invariantly preserved or that some bad property will never happen? The standard way to reason about such properties is by means of temporal logics.

In our case temporal logics are supported by Maude's built-in LTL model checker. Properties regarding dynamic aspects of reconfigurations are expressed using the Linear-time Temporal Logic (LTL). Roughly, one is able to reason about infinite sequences of reconfigurations, by expressing properties on the ordering of state (i.e. configuration) observations. Such observations are predicates expressing structural properties as above mentioned.

As an example we can write the formula asserting that *it is always true that a collection of cars has at least 2 cars* as [] at-least-k-cars(2), where [] denotes the *always* temporal operator □. This property trivially holds for the design of figure 4. Indeed, no reconfiguration rule allows cars to leave the system so that their number remains constant.

6 Related Work

The Service Component Architecture (SCA) [9] focuses on policies and implementation aspects of services but is not based on UML. The work in [24] is based on UML models and transformations to executable descriptions of services. However, the approach lacks an appropriate UML profile preventing one from building models at the high level of abstraction; thus producing overloaded diagrams. The work of [13] proposes to use modes to address dynamic reconfiguration of service-oriented architectures and extends the UML to visualise such reconfiguration. The UML extension sticks to the mode terminology and does not include a visualisation of the transformation rules. The OMG is also working to standardise SoaML, a UML profile and metamodel for services [21]. The current version does not support styles or reconfigurations.

Structural aspects for services, modelled in UML, have also been addressed in several other works (e.g. [16]); however, as far as we know, none of them is based on a formal background like the one presented here. The only exception is the UML extension for service-oriented architectures that can be found in [2]. The approach includes refinement issues based on architectural styles and is formalised by graph transformation systems. It includes stereotypes for the structural specification of services. However, it does not introduce specific model elements for the orchestration of services, the notion of style there is less expressive (it basically amounts to our type graphs) and reconfigurations there are limited to unconditional ones.

A completely different approach to modelling architectural styles in UML would be to use constraints expressed in the OMG Object Constraint Language (OCL). To the best of our knowledge, however, there is no reconfiguration approach using solely OCL, which would have two drawbacks: OCL is a textual notation and it would introduce another language to the service engineer.

ADR has been mainly inspired by graph-based approaches to architectural styles [15,20] (see [7] for a comparison). The use of graphs and graph transformations to model architectural styles has been proposed by several authors (see [23], for instance) who based their approaches on the concept of *shapes* in programming lan-

guages. ADR shares also concepts with approaches based on process calculi with reconfigurable components (e.g. [1]). The main advantages of ADR are that the hierarchical and inductively based approach allows us to compactly represent complex reconfiguration rules, and that, style preservation is guaranteed by construction. ADR is also related to approaches that deal with reconfigurations in software architectures defined by an ADL (see [6]). The main advantages of ADR are the unified treatment of design, behaviours and reconfiguration, and the use of hierarchical, inductive reconfigurations. A comparison with a logic based architectural design methodology was given in [8].

7 Conclusion

We have presented a novel extension of UML4SOA, our approach for the modelling of service oriented architectures. The profile offers suitable ingredients to deal with architectural styles and reconfigurations. We have equipped the proposed profile extension with a formal semantics, offering support for analysis and verification from the very early stages of modeling. Thus, our approach is a comprehensive and pragmatic but 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 UML4SOA specifications, possibly by means of Maude-supported, MOF-based model transformations [3]. 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 UML4SOA or other models. In future work we would like to integrate our approach in the SENSORIA suite of tools and techniques, which already includes some development [12] and re-engineering (legacy systems as services) [10] instruments.

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. *SOSYM*, 5(2):187–207, 2006.
3. A. Boronat and J. Meseguer. An algebraic semantics for mof. In *FASE*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
4. R. Bruni, A. Lluch-Lafuente, and U. Montanari. Hierarchical design rewriting with maude. *Electr. Notes Theor. Comput. Sci.*, 238(3):45–62, 2009.
5. R. Bruni, A. Lluch Lafuente, U. Montanari, and Emilio Tuosto. Service Oriented Architectural Design. In *TGC'07*, volume 4912 of *LNCS*, pages 186–203. Springer, 2008.
6. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Architectural Design Rewriting as an Architecture Description Language. *R2D2* Microsoft Research Meeting, 2008.
7. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style Based Architectural Reconfigurations. In *EATCS Bulletin*, volume 94, pages 161–180. February 2008.
8. A. Bucchiarone, R. Bruni, S. Gnesi, and A. Lluch Lafuente. Graph-Based Design and Analysis of Dynamic Software Architectures. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*. Springer, 2008.
9. S. Consortium. Service Component Architecture Policy Framework, Version 1.0, 2007.

10. R. Correia, C. Matos, R. Heckel, and M. El-Ramly. Architecture migration driven by code categorization. In *ECSA*, volume 4758 of *LNCS*, pages 115–122. Springer, 2007.
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. H. Foster and P. Mayer. Leveraging integrated tools for model-based analysis of service compositions. In *ICIW'08*. IEEE Computer Society Press, 2008.
13. H. Foster, S. Uchitel, J. Kramer, and J. Magee. Leveraging Modes and UML2 for Service Brokering Specifications. In *MDWE'08*, volume 389 of *LNCS*, pages 76–90. CEUR, 2008.
14. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer Verlag, 1992.
15. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *ENTCS*, 109, 2004.
16. S. Johnston. UML 2.0 Profile for Software Services, 2005.
17. N. Koch, P. Mayer, R. Heckel, L. Gönczy, and C. Montangero. D1.4a: UML for Service-Oriented Systems. Specification, SENSORIA Project 016004, 2007.
18. P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *SCC'08*, IEEE, pages 1–6. IEEE, 2008.
19. J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3):213–237, 2007.
20. D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.
21. Object Management Group (OMG). Service oriented architecture Modeling Language (SoaML). `http://www.omg.org/cgi-bin/doc?ptc/09-04-01`. Last visited: July 2009.
22. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
23. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey, USA, 1996.
24. D. Skogan, R. Grønmo, and I. Solheim. Web service composition in UML. In *EDOC'04*, pages 47–57. IEEE Computer Society, 2004.
25. M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-Based Development of Service-Oriented Systems. In *FORTE'06*, volume 4229 of *LNCS*, pages 24–45. Springer, 2006.