Architectural Design Rewriting as an Architecture Description Language

Roberto Bruni Alberto Lluch Lafuente
Ugo Montanari
Department of Computer Science, University of Pisa
{bruni,lafuente,ugo}@di.unipi.it

Emilio Tuosto
Department of Computer Science, University of Leicester
et52@mcs.le.ac.uk

Abstract

Architectural Design Rewriting (ADR) is a declarative rule-based approach for the design of dynamic software architectures. The key features that make ADR a suitable and expressive framework are the algebraic presentation of graph-based structures and the use of conditional rewrite rules. These features enable the modelling of, e.g. hierarchical design, inductively defined reconfigurations and ordinary computation. Here, we promote ADR as an Architectural Description Language.

Categories and Subject Descriptors D.2 [*Software Engineering*]: Design Tools and Techniques, Software Architectures; G.2.2 [*Discrete Mathematics*]: Graph Theory—Graphs

Keywords Dynamic Software Architectures, Architectural Styles, Graphs, Term Rewriting

1. Introduction

Architectural Design Rewriting (ADR) (Bruni et al. 2008b) is a proposal for the design of reconfigurable software systems, conceived in the spirit of conciliating software architectures and process calculi by means of graphical methods. ADR offers a formal setting where design development, run-time execution and reconfiguration are defined on the same foot.

The key features of ADR are: (i) hierarchical and graphical design; (ii) rule-based approach; (iii) algebraic presentation; and (iv) inductively-defined reconfigurations. Architectures are modelled by *typed designs*: a kind of interfaced graphs whose inner items represent the architectural units and their interconnections and whose interface expresses the overall type and its connection capabilites. Architectures are designed hierarchically by a set of composition operators called *design productions* which enable: (i) top-down refinement, like replacing an abstract components with a possibly partial realisation, (ii) bottom-up typing, like deducting the type of and actual architecture, and (iii) well-formed composition, like composing some well-typed actual architectures together so to guarantee that the result is still well-typed.

Domains of *valid* architectures, i.e. those compliant to styles, patterns or constraints, are defined in a declarative way by means of design productions. Such productions have both a functional reading as valid architectural compositions and a grammar reading as providing an inductive definition of valid architectures.

In the functional reading, the set of productions defines an algebra of design terms, each encoding the structure of the architecture and providing a proof of style conformance. The interpretation of a design term is a design, i.e. the actual architecture.

Reconfiguration and behaviour are given as term rewrite rules acting over design terms rather than over designs. This has many advantages: (i) terms compactly and conveniently encode the hierarchical structure of the architecture; (ii) ordinary term rewriting techniques allow to specify complex reconfigurations and computations that can exploit the hierarchical structure encoded in terms; (iii) preserving properties such as style-conformance during reconfiguration can be ensured by construction.

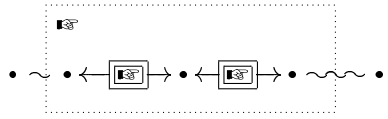

Figure 1. Design of a pipe (type ☞). The internal structure is formed by two partially specified pipes connected in sequence by binding the respective left and right ports (arrows) directly (connectors are neglected for simplicity) at the same node (bullet). Only the two ports at the extremes of the pipe are exposed (waved lines) at the interface (dotted box). The figure can also be interpreted as a design production composing two pipes in a pipe or as a refinement of a pipe as two pipes.

2. ADR as ADL

ADR was not conceived as an Architecture Description Language (ADL) but as a suitable model for style-consistent design and reconfiguration of software architectures. As a matter of fact, ADR turned out to be a general mechanism, suitable for heterogeneous models such as network topologies, architectural styles and modelling languages (Bruni et al. 2007). Despite of its generality, we think that the features of ADR are particularly tailored to ADL problematics. Indeed, we believe that ADR can be seen as an ADL itself, as a formal model of existing ADLs, possibly equipping them with extended features, like conditional reconfigurations. We promote this vision of ADR by discussing the issues that the software architectures community retains particularly relevant.

2.1 Components and Connectors

The main actors of ADR are design, which uniformly model both components and connectors. Designs are technically defined by hierarchical hyperedges whose internal structure can range from an empty graph to an arbitrarily complex graph (see Figure 1).

Interface. The interface of a design is given by the tentacles of the outerface hyperedge. Each tentacle represents a port (resp. role) of the corresponding component (resp. connector). Attaching a port to a role is done by connecting the respective tentacles to the same node. As usual in many ADLs, ports and roles are typed.

Types. Most of the ingredients of ADR such as design terms, designs, hyperedges and nodes are typed. This enables, e.g. the

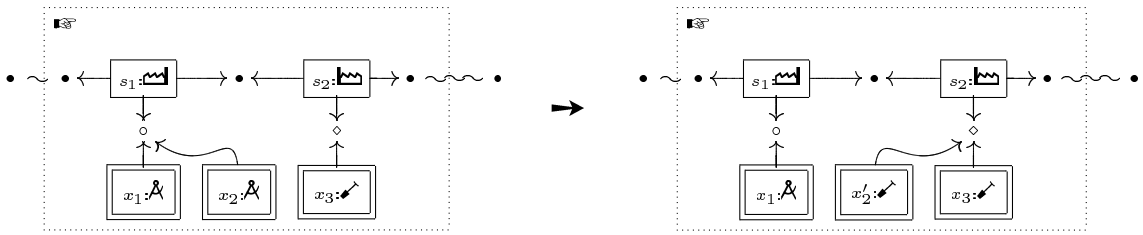

Figure 2. Reconfiguration rule for migrating tasks in a pipe. The design on the left is a pipe (type ☞) that consists of two concatenated servers of different types (and) to which a collections of tasks of appropiate type (and) are attached. The reconfiguration requires x_2 to evolve into x'_2 in order to migrate from s_1 to s_2. Types are changed consistently and the design obtained is still of type ☞.

construction of style-consistent architectures and the distinction between design classes (types) and their instances (designs).

Semantics. ADR is a formal model with a well-defined semantics. As a matter of fact, ADR builds on well-founded techniques such as algebraic approaches to graph transformation and rewrite rules in Plotkin's structural operational semantics style and Meseguer's conditional term rewriting.

Constraints. Architectural constraints are typically given in some logic-based language. Instead, ADR promotes to encode constraints as types when possible: the set of all architectures satisfying some constraint should correspond to the set of all terms of a certain type, guaranteeing constraint consistency and its preservation by construction.

Evolution. Architectural evolution can be given in terms of software modes. In each mode different behaviour, constraints or reconfigurations might apply. Modes and mode changes can be suitably modelled in ADR with types and rewrite rules, respectively.

Non-functional properties. The current version of ADR does not consider non-functional properties. However, we intend to model QoS aspects by means of constraints systems associated to designs.

2.2 Architectural Configurations

Architectural configurations are modelled by designs, allowing the uniform treatment of components, connectors and configurations. Hence, we restrict the discussion to configuration particularities.

Compositionality. Composite configurations, components and connectors are constructed hierarchically via design productions. Non admissible configurations, like connecting a client with another client instead of a server, can be ruled out by construction.

Refinement. Refinement is straightforwardly supported by a particular reading of design productions.

Traceability. ADR supports traceability aspects by means of equipping architectures (designs) with a witness of their construction (design terms). ADR promotes to carry such information over run-time to support efficient and autonomous reconfiguration.

Scalability. Adding entities to an architecture can be achieved in ADR via suitable refinements or rewrite rules. The features of ADR favour a modular approach, so that, e.g. the addition of new components can be localised in the desired sub-architecture, without affecting the rest of the system.

Dynamism. Complex behaviours and reconfigurations are expressed in ADR by flexible rewrite rules (see Figure 2). ADR does not marry any particular kind of dynamism (such as programmed or repairing); it is general enough to cover most of them.

Understandability. Architectural configurations in ADR improve understandability due to their hierarchical composition, which allows to browse complex structures inductively and to focus on the most convenient level of detail. A further support to understandability is the immediate visual representation as (typed) graphs.

2.3 Tool Support

ADR tool support is under development (Bruni et al. 2008a). At its current status prototype specifications can be simulated and some structural and behavioural properties can be analysed with verification techniques working at different levels of abstraction.

3. Conclusion

ADR suitably captures most of the features that an ADL should consider. The main advantages of ADR over similar approaches is given by the use of architectural construction information in form of design terms which enables hierarchical reconfigurations, which, most notably, are style-consistent by construction and easy to understand. Over most ADLs, ADR offers a unifying model to represent architectural design, reconfiguration, and ordinary behaviour. We believe that ADR can help in understanding and solving ADL problematics and that it can serve as the basis to formalise or extend them. For instance, the term-based feature of ADR can be exploited by ADLs that do not consider dynamism by defining architectural specifications as terms and modelling dynamism as suitable term rewrite rules. The tool-support for ADR is in a primitive but promising stage. Further information on ADR can be found at `http://www.albertolluch.com/adr.html`.

Acknowledgments

ADR is supported by the EU FETPI-GC2 project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB project TOCAI (*Knowledge Oriented Technologies for Enterprise Integration in Internet*).

References

Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. Service Oriented Architectural Design. In *Proceedings of the 3rd International Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 2007.

Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. Hierarchical Design Rewriting with Maude. In *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008a.

Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (94):161–180, February 2008b.