Service Oriented Architectural Design*

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

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

Abstract. We propose *Architectural Design Rewriting* (ADR), an approach to formalise the development and reconfiguration of software architectures based on term-rewriting. An architectural style consists of a set of architectural elements and operations called *productions* which define the well-formed compositions of architectures. Roughly, a term built out of such ingredients constitutes the proof that a design was constructed according to the style, and the value of the term is the constructed software architecture. A main advantage of ADR is that it naturally supports style-preserving reconfigurations. The usefulness of our approach is shown by applying ADR to SRML, an emergent paradigm inspired by the *Service Component Architecture*. We model the complex operation that composes several SRML modules in a single one by means of suitable rewrite rules. Our approach guarantees that the resulting module respects SRML's metamodel.

1 Introduction

Service orientation is becoming a standard paradigm in the development of software applications. The paradigm is centred around the notion of *service*, i.e. a computational entity whose functional and non-functional aspects can be described in a standard document to be advertised in some service registries and made available for discovery. Contrary to traditional applications, service oriented ones are not just statically assembled. Instead, they have the potentialities for allowing dynamic assembly via publication, discovery, selection and binding.

SENSORIA [10] (*Software Engineering for Service-Oriented Overlay Computers*) is a research project that aims to develop a novel and comprehensive approach for engineering service oriented computations. Key issues of SENSORIA concern the early stage and development of service specification, like design and reconfiguration of service-based architectures. In this setting, the configuration of a system consists of the present components and interconnections (i.e. the architecture), together with their current state. Architectural styles can be

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

applied to reuse existing design patterns and thus facilitate software development. In addition, they offer a further benefit when architectural information is carried over the execution of the system, since one can control whether changes in the system imply changes in the architecture. During run-time, changes in the configuration like dynamic binding require reconfigurations of the architecture. Static reconfiguration of an architecture may also be necessary, e.g. when deploying an existing architecture on a platform it was not originally designed for. Often, the architectural style must be preserved or consistently changed.

In this paper, we propose *Architectural Design Rewriting* (ADR) [5] as a novel formal approach to tackle some of the aforementioned issues of service-oriented software development. A formal metamodel for static and dynamic aspects of the SENSORIA Reference Modelling Language (SRML) [13] is given in order to demonstrate the expressiveness and flexibility of ADR. SRML has been inspired by various formalisms: orchestration languages such as ORC [24], transactional process calculi such as Sagas [6], Web service conversation models [4] and, most notably, IBM's Service Component Architecture (SCA) which has become part of the *Open Service Oriented Application* [25] initiative involving many major industrial partners (IBM, Sun and Oracle, among others). SCA and SRML are complementary approaches. Indeed, SRML is aimed at the definition of mathematical semantics for modules while SCA focuses on implementation.

Though some aspects of architectural reconfiguration can be captured within other type-theoretic frameworks e.g., the calculus of constructions [9], we argue that ADR is very intuitive and more flexible with respect to other approaches. Indeed, ADR gives software architects the possibility to avoid style-preserving reconfigurations when necessary while usually type-theoretic frameworks impose it or require a complex machinery to get around it.

SRML Overview. When designing an architecture, it is desirable to consider the concept of *architectural style* [26], i.e. some set of rules indicating which components can be part of the architecture and how they can be legally interconnected. Traditional architectural styles include client-server and pipelines. Some of such styles have been also defined in the realm of service oriented applications, going from abstract client-server styles [21] to more concrete and complex architectures [2]. The basic ingredients of a style are architectural elements and structural constraints. For instance, the architectural elements of SRML are drawn in Fig. 1 (borrowed from [13]) and include service modules (square boxes), components (rounded boxes), wires (straight lines) and interfaces (concave and convex polygons). This graphical notation is in the line of the traditional boxes-and-lines or component-and-connectors [8] notations and much more inspired by the graphical notation of SCA. The structural constraints, in their turn, require modules to be interconnected via external wires such that one of the require interfaces (EX-R) of a module is connected to the provide interface (EX-P) of another one. Inside a module, components and interfaces are connected via internal wires (IW). An SRML architecture is given at the highest level of abstraction by an *assembly of modules* with possibly some discovered but not bound service modules interconnected via *external wires*. For instance,

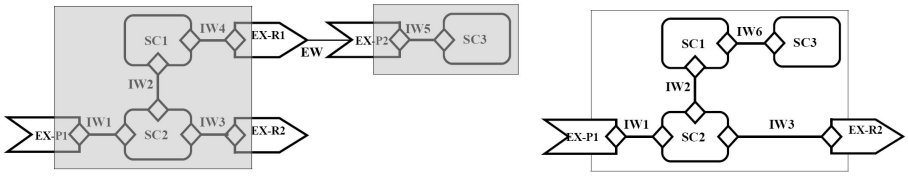

Fig. 1. An SRML diagram before (left) and after (right) composition.

Fig. 1 depicts an architecture with a service module (the leftmost square) which requires two additional services to be attached to the external interfaces EX-R1 and EX-R2. The one corresponding to EX-R1 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 [12]. SRML provides a mechanism to achieve this static reconfiguration, by means of an algorithm that manipulates SRML specifications. As an example, the assembly of Fig. 1 (left) can be composed into the service module depicted in Fig. 1 (right), where wire IW6 is derived according to certain composition rules. Such reconfigurations require a proof of correctness w.r.t. style preservation.

ADR Overview. ADR [5] is a recent proposal for the style-consistent design and reconfiguration of software architectures, conceived in the spirit of initiatives (e.g. [20]) that promote the conciliation of software architectures and process calculi by means of graphical methods. Although not discussed here, ADR can also represent the normal behaviour of systems (i.e., the evolution of components). For example, a representation of π-calculus [23] based on a graphical encoding [15] is currently under development. ADR offers a unified setting where design development, ordinary execution and reconfiguration are defined on the same foot. The key features of ADR are: (i) rule-based approach; (ii) hierarchical and graphical design; (iii) algebraic presentation; and (iii) inductively-defined reconfigurations. Architectures are suitable modelled by so-called *designs*: a kind of graphs whose items suitably represent the architectural components and their interconnections. Architectures are designed hierarchically by a set of composition operators called *design productions* which enable: (i) top-down design by refinement, (ii) bottom-up typing of actual architectures, and (iii) well-formed composition of architectures. An architectural style is defined as a set of *design productions* such that a design is style-consistent whenever it can be defined by a *design term* which makes use of the corresponding design productions. Reconfiguration and behaviour are given as rewrite rules that are defined over design terms rather than over designs. The main advantages of ADR are that: (i) instead of reasoning on flat architectures (designs), ADR specifications provide a convenient hierarchical structure (design terms), by exploiting the architectural classes introduced by the style, (ii) complex reconfiguration schemes can be de-

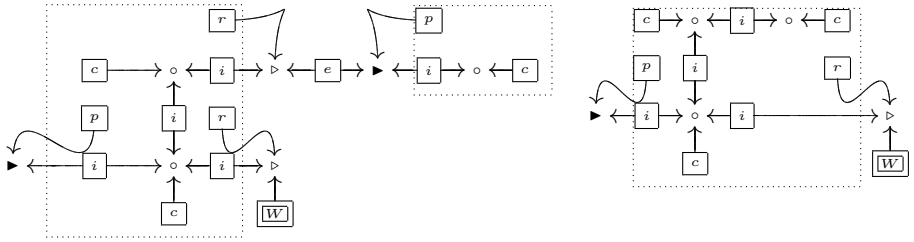

Fig. 2. Two SRML diagrams in the graphical representation of ADR.

fined inductively at any level of abstraction, and (iii) style-guarantees during reconfiguration or execution are ensured by construction.

Contribution. Our main goal is to define an ADR-based architectural style to support the development and reconfiguration of SRML diagrams according to the SRML metamodel. We shall define an architectural style given by a vocabulary of architectural elements and a set of operations for the construction of SRML diagrams. More precisely, we build an algebra where the evaluation of a design term is a design representing an SRML diagram. It is worth mentioning that any SRML diagram can be represented by a design. For instance, Fig. 2 depicts ADR designs for the SRML diagrams of Fig. 1: the correspondence is explained in § 3. SRML reconfigurations are then modelled as ADR rewrite rules over the design terms rather than over plain designs, guaranteeing style preservation and, thus, metamodel conformance.

Structure of the Paper. § 2 overviews ADR. § 3 describes an ADR style for SRML. § 4 addresses the problem of reconfiguration of SRML diagrams, focusing on module composition. § 5 summarises our work, draws conclusions and sketches interesting research avenues. For reader's convenience the graphical representation of the most complex reconfiguration rule is included in appendix A.

2 Architectural Design Rewriting

In this section we summarise the key features of ADR. We refer the reader to [5] for a detailed technical presentation. Roughly, ADR adheres to three main principles: (i) architectural designs are modelled by suitable graphs called *designs* and come equipped with their proofs of construction called *design terms*; (ii) architectures are designed hierarchically by a set of composition operations called *productions* out of which design terms are built and architectural styles are basically given by sets of such productions such that an architectural design is compliant with a style if its design term uses the corresponding productions only; (iii) reconfigurations are powerful, expressive, hierarchical and style-consistent rewrite rules defined over design terms instead of designs.

We illustrate the principles of ADR with a simple example where a local network architecture admits two styles where each network hub has respectively

two and three degrees of connectivity. Connections between hubs are also driven by the style, so that, for instance, the only legal 2-degree networks are rings.

Principle i), i.e. modelling architectural designs by suitable graphs, has been widely exploited in the literature (e.g. [2, 22]). For instance, in the well established component and connector view, software architectures are modelled by graphs of components and connectors. In ADR one can represent such graphs as follows. A component is modelled by a hyperedge whose outgoing tentacles represent the components interface, i.e. its ports. Similarly, a connector is modelled by a hyperedge whose outgoing tentacles represent the connector's interface, i.e. its roles. Attaching a port to a role is done by connecting the respective tentacles to the same node. The main actor of ADR are *designs* (see Definition 2), which are used to model components, connectors and architectural configurations.

The choice of graphs as the domain of our algebra is inherited from the previously mentioned approaches, but it is well justified by the immediate user-friendly visual representation and the expressive power of graphs and their rewritings which have been used for years as a model, not only of software architectures, but of many other things ranging from data structures to process calculi.

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

The different classes of edges used in the network example are drawn in Fig. 3 where an explicit numbering or naming of tentacles is avoided in favour of an implicit convention that assumes that the order of tentacles exiting from each edge is given by considering the leftward tentacle as the first one and the remaining tentacles as clockwise ordered.

More generally, we could consider the association of semantic information to graph items. For example, nodes can represent variables taking values over a finite domain and edges can express suitable constraints over them. Another example is the association of theories to nodes and theory morphisms to edges (e.g. a theory of interaction signatures). Then, this additional information can be exploited to drive the development and reconfiguration phases. We shall not give special emphasis to such aspects. However, we shall return to this issue along with the paper suggesting how we could capture semantical aspects of SRML in addition to the structural ones, on which we shall focus.

Principle ii), i.e the hierarchical design of architectures, is also not particularly original in itself [18], but it is here enhanced by a novel algebraic presentation. An architectural style consists of a vocabulary of architectural elements (represented by a type graph), and a set of production rules indicating how they can be legally interconnected. We distinguish two kinds of edges in the type graph: terminals $\mathcal{T}$ and non-terminals $\mathcal{NT}$. Likewise string grammars, terminal edges represent basic, non-refinable, concrete components of the architecture, while non-terminal edges, represent complex, refinable, abstract components.

In the network example we have $\mathcal{T} = \{\mathsf{2hub}, \mathsf{3hub}\}$ and $\mathcal{NT} = \{\mathsf{2N}, \mathsf{3N}, \mathsf{NET}\}$. Our graphical notation uses single-framed and double-framed boxes for terminals and non-terminals, respectively (see Fig. 3).

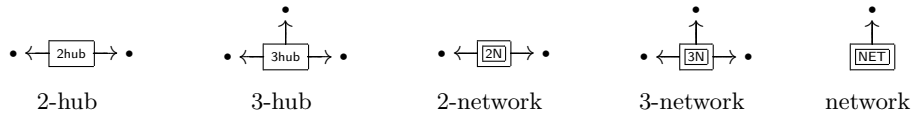

Fig. 3. Architectural elements of the network example.

In ADR, software architectures are not just specified by graphs. Instead, they are represented by *designs*. A design is a well-formed architecture with a typed interface (represented by a non-terminal edge) and an internal structure (represented by a graph). The interface is an abstract view of the design as a single component, thus hiding the internal representation except for those nodes that are exposed in the interface.

Definition 2. *A* design *is a triple* $d = \langle L_d, R_d, i_d \rangle$*, where* L_d *is the interface graph consisting a single non-terminal edge (called* interface*) whose tentacles are attached to distinct nodes;* R_d *is the body graph; and* $i_d : V_{L_d} \to V_{R_d}$ *is the total function associating body nodes to interface nodes.*

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

Designs are assembled by means of composition operations, called *design productions*.

Definition 3. *A* production *is a tuple* $p = \langle L_p,\ R_p, i_p, l_p \rangle$ *where* $\langle L_p, R_p, i_p \rangle$ *is a design with* n_p *occurrences of non-terminal edges in* R_p *that are mapped by the bijection* l_p *on segment* $[1, 2, \ldots, n_p]$.

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

Our network example uses production $\texttt{link2} = \langle L_{\texttt{link2}}, R_{\texttt{link2}}, i_{\texttt{link2}}, l_{\texttt{link2}} \rangle$ whose functionality is $\texttt{link2} : \mathsf{2N}_1 \times \mathsf{2N}_2 \to \mathsf{2N}$. Intuitively, `link2` specify an operator of the algebra that arranges two designs of type 2N into a new 2N design. In hyperedge replacement style (see [16] for details) `link2` can be written as

where the left-hand side graph can be replaced by the right-hand side one. A compact and elegant graphical representation of `link2` is drawn in Fig. 4 where

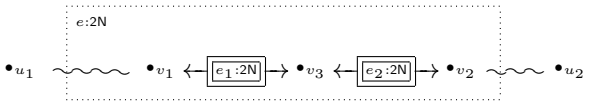

Fig. 4. Graphical representation of production `link2`

the left-hand side (i.e., the interface edge) is represented by the outermost dotted box whose nodes u_1 and u_2 are outside the dotted box. The right-hand side graph of `link2` is depicted in the dotted box and the nodes v_1 and v_2 are exposed in the interface through waved lines. Finally, the order of arguments of each production is implicit: from top to bottom, and left to right if on the same row, e.g. $l_{\texttt{link2}} = \{e_1 \mapsto 1, e_2 \mapsto 2\}$. In the rest of the paper we will neglect the textual representation of productions as well as the identities graph items in their visual representation. The rest of the productions of our example are depicted in Fig. 5. For example, a 2-network is either a network with just one 2-hub (`basic2`) or the chaining of two 2-networks (`link2`). For 3-degree networks the composition involves three arguments of type 3N. For instance, production `link3` has type $\mathsf{3N} \times \mathsf{3N} \times \mathsf{3N} \to \mathsf{3N}$. Finally, a network is either a 2-network (`net2`) or a 3-network (`net3`), whose interface nodes are merged together. To illustrate the operations associated to productions, consider the term $\texttt{net2}(\texttt{link2}(\texttt{link2}(\texttt{basic2}, \texttt{basic2}), \texttt{basic2}))$. Subterm $\texttt{link2}(\texttt{basic2}, \texttt{basic2})$ evaluates to a 2-network made of two concatenated 2-hubs. Such value is used in the subterm $\texttt{link2}(\texttt{link2}(\texttt{basic2}, \texttt{basic2}), \texttt{basic2})$ to obtain a 2-network made of three 2-hubs. Finally the whole term evaluates to the design on the right of Fig. 6. Similarly, the term $\texttt{net3}(\texttt{link3}(\texttt{basic3}, \texttt{basic3}, \texttt{basic3}))$ evaluates to the design on the left of Fig. 6. Instead an expression like $\texttt{net2}(\texttt{basic3})$ is not valid, because types mismatch.

The use of productions offers a mechanism that supports the construction of architectural designs both in a top-down way by refinement of terms and a bottom-up way by composition of terms. A typing mechanism can be used as a reverse engineering method to obtain a design term for a given design.

A crucial benefit of the use of productions regards the concept of *architectural style*, i.e. a certain set of architectural designs considered to be valid or in conformance with some design pattern. In fact, while changes in the architecture are acceptable and even necessary, the architectural style should be preserved in most cases. For instance, in a system with client-server architectural style clients connecting and disconnecting from the server are permitted, while a client connecting to a client is not. Changes in style are also interesting, take for instance, a token ring architecture configuring into a star-shaped one to achieve a most efficient communication. Typical architecture description languages define a style in terms of architectural constraints to be checked after or during the construction of a design. Instead in ADR, an architectural design is defined by a set of productions. Any design term that uses those productions defines an architectural design that is consistent with the corresponding style. As a conse-

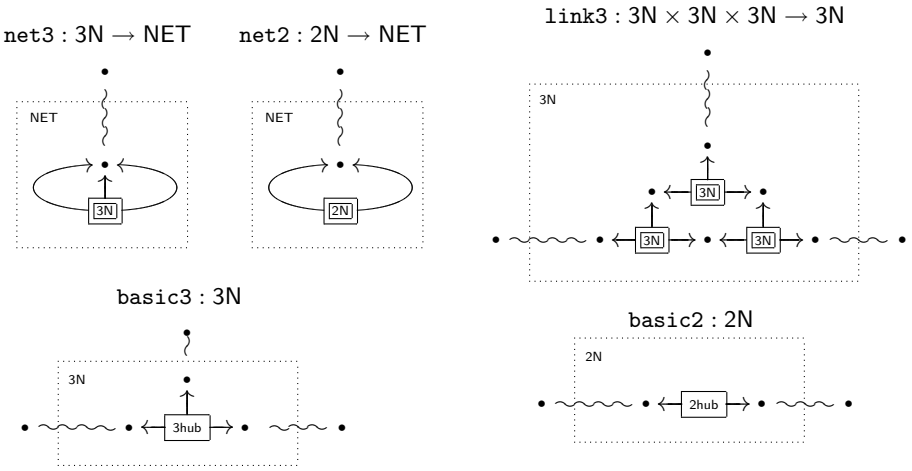

Fig. 5. Design productions of the network example.

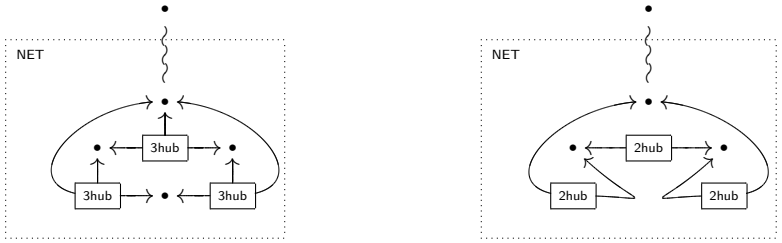

Fig. 6. Two network designs: with three-degrees (left) and two-degrees (right) hubs.

quence, no proof of style-consistency is needed. Designs are style-consistent by construction.

We say that a design d is *well-formed* if there is a well-typed design term whose value is d, while we say that d is consistent w.r.t. to a style (or *style-consistent*) if the design term uses design productions of the style. Note that style-conformance implies well-formedness, but the contrary is not true. For instance, one could construct an architecture mixing productions of two different styles. However, for the sake of simplicity we assume that mixing styles is not possible. This is achieved by requiring the set of types used in each pair of styles to be disjoint, in which case the style of a design term is automatically given by its type and thus well-formedness and style-consistency coincide.

Principle iii), i.e. reconfigurations defined over design terms instead of actual architectures, exploits the algebraic presentation of ADR. This enables a straightforward definition of hierarchical and inductive reconfigurations as ordinary term rewriting and conditional SOS rules. The main guarantee offered by ADR is that reconfigurations are style-preserving by construction.

$$\texttt{basic3to2}: \ \texttt{basic3} \xrightarrow{\texttt{3to2}} \texttt{basic2} \qquad \texttt{net3to2}: \ \frac{x \xrightarrow{\texttt{3to2}} x'}{\texttt{net3}(x) \longrightarrow \texttt{net2}(x')}$$

$$\texttt{link3to2}: \ \frac{x_1 \xrightarrow{\texttt{3to2}} x_1' \quad x_2 \xrightarrow{\texttt{3to2}} x_2' \quad x_3 \xrightarrow{\texttt{3to2}} x_3'}{\texttt{link3}(x_1, x_2, x_3) \xrightarrow{\texttt{3to2}} \texttt{link2}(\texttt{link2}(x_2', x_1'), x_3')}$$

Fig. 7. Conditional reconfigurations of the network example.

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

For example, the rule $\texttt{link2}(x_1, x_2) \to \texttt{link2}(x_2, x_1)$ where x_1 and x_2 have type 2N, reconfigures any 2N chain by switching the order of its two components.

In case certain local changes in the architecture are subordinated to the corresponding adaptation of the adjacent environment, we can use conditional reconfiguration rules, expressing that a composed architecture can be rewritten only if its sub-components are suitably transformed first. This step makes the formalism very powerful. Simple conditional rewrites take the form:

$$\frac{t_1 \to t_1' \quad \ldots \quad t_n \to t_n'}{L \to R}$$

meaning that, given an assignment η, the architecture $L\eta$ can be reconfigured according to $R\eta$ only if each $t_i\eta$ can be reconfigured to $t_i'\eta$.

The reconfiguration rules needed to downgrade the hubs of any 3-network are defined in Fig. 7. Note that types are not preserved by rewrites labelled 3to2, which change the type from 3N to 2N. But this is not a problem because rules are intended to be applied in appropriate (inductively defined) contexts. This is particularly clear in the rule net3to2 where the conclusion actually transforms a network into a network: the silent label makes it applicable in any larger context. The rule link3to2 is graphically represented in Fig. 8. By applying net3to2 (once), link3to2 (once) and basic3to2 (three times), we obtain a style-preserving rewrite from the leftmost design in Fig. 6 to the rightmost one.

For another simple but illustrative example of the ADR modelling of a road assistance scenario we refer the reader to [5].

3 Design of SRML diagrams

The metamodel of SRML is defined in terms of some class diagrams. Roughly, a *module* is an abstraction of a business entity that can either perform a task (in which case it is called an *activity module*) or provide a service (called a *service module*). Modules consist of *components* and *external interfaces*, possibly

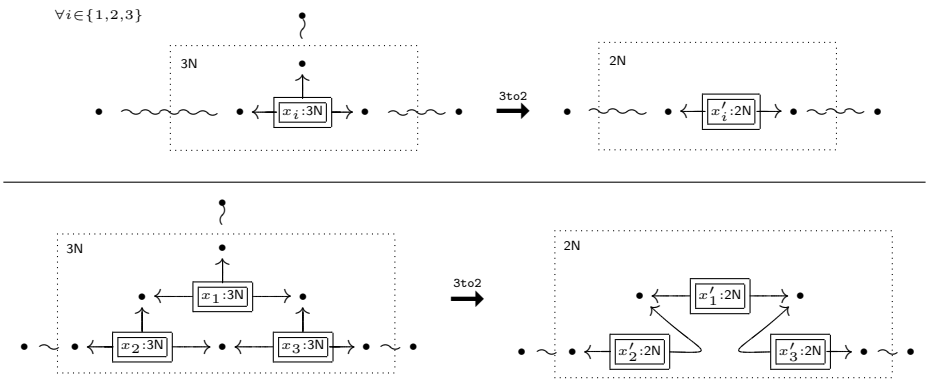

Fig. 8. Graphical representation of rule `link3to2`.

linked via *internal wires*. Components abstract the computational aspects of modules, while interfaces model the interaction with the external world. There are two kinds of external interfaces: *provide* and *require*. The former are present in service modules and indeed they specify the service provided by modules. Require interfaces, instead, specify the services needed. A needed service can be identified during static- or run-time. The latter being the open-frontier of the service-oriented paradigm, but static binding is interesting too as we shall see.

In this section we define an ADR architectural style that is compliant with the SRML metamodel. The encoding of SRML composition as ADR reconfiguration is deferred until § 4.

3.1 Architectural Elements of SRML

Service components, wires and interfaces are concrete architectural elements that we represent as terminal edges (see Fig. 2, for instance). A service component is represented by an edge of type c with a unique tentacle representing its interaction port attached to a node of type $\circ$ (a *component port*). Require and provide interfaces are edges of type r and p, respectively, whose tentacles are attached to nodes of type $\triangleright$ (*required port*) and $\blacktriangleright$ (*provided port*), respectively. Internal and external wires are represented with edges respectively typed by i and e. Internal wires must be attached to a node of type $\circ$ or $\blacktriangleright$ and another node of type $\circ$ or $\triangleright$. This means that the left (resp. right) tentacle of an internal wire cannot be of type $\triangleright$ (resp. $\blacktriangleright$). External wires are attached to one node of type $\triangleright$ via its left tentacle and another one of type $\blacktriangleright$ via its right tentacle.

Typing imposes syntactical restrictions not present in the (less-accurate) UML metamodel (e.g., it does not make sense to connect two require interfaces via an internal wire). Further syntactical and semantic aspects are enforced by suitable mechanisms that impose restrictions on the actual use of wires in a diagram. For instance, the ports of components and interfaces and the roles of wires

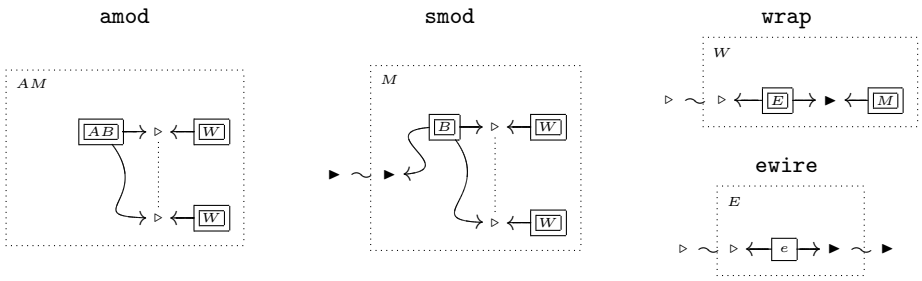

Fig. 9. An activity (left), a service (center) and a wrapped service (right).

have associated suitable interaction signatures. Then, a component or interface can share a node with an internal wire only if their respective ports and roles have the same signature. External wires, instead require both attached signatures to be *compatible* and the behaviour of the required service to be entailed from the provided one. We deal only with the most abstract structural aspects of SRML; insights on aforementioned restrictions are in [13] where suitable models of interaction signatures and service behaviour entailment are fully detailed.

The non-terminal architectural elements of the SRML style are present in the various figures of this section where families of architectural elements are actually represented. In fact, designs and productions are parametrised by the rank of their constituents. For simplicity, our graphical notation abstracts away from ranks and a dotted line between two nodes for representing any number of them. In other words, we overload the name of (ranked) productions and assume the application of production exploits polymorphism in a suitable way.

The presented architectural elements are the basic ingredients to build graphs that represent SRML diagrams such as those in Fig. 2.

3.2 Design Productions for SRML

We follow a top-down presentation of the ADR productions for SRML (from modules to wires and components).

Service and activity Modules. An SRML module consists of a body (to be identified during development) and some wrapped services (to be refined at run-time). Productions `smod` and `amod` in Fig. 9 model this structure. For instance, we could have $\texttt{amod}(t, x_1, x_2)$, t being a concrete design of type AB (the complete specification of the body of the activity), while x_1 and x_2 are variables of type W.

Remarkably, the ADR modelling of SRML binding is performed in two steps. First, the selected service is wrapped in the module via an external wire and then the internal wires of both the activity and the service are rearranged to internalise the connection (cf. Section 4). Here, the wrapping step is modelled by production $\texttt{wrap} : E \times M \to W$ in Fig. 9, that wraps a service module by

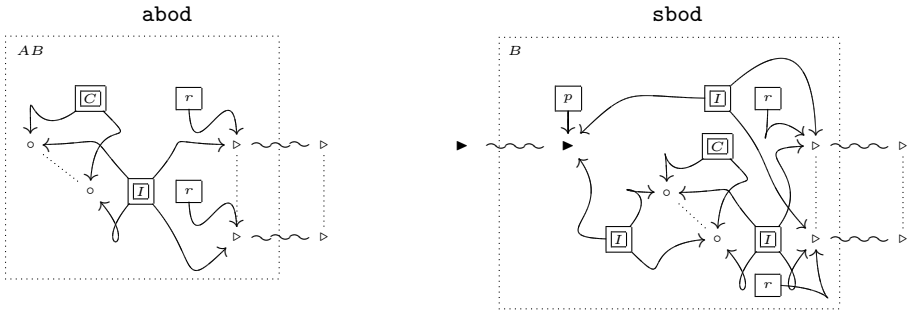

Fig. 10. The body of an activity module (left) and a service module (right).

means of a binding wrapper that connects the require interface port with the interface port provided by the service. The only binding considered here is a single external wire (see production `ewire`).

Module Bodies. The body of a module consists of a collection of service components and interfaces connected via internal wires. The difference between the body of an activity and a service module is that the former does not have a provide interface. In the body of a service module it is convenient to distinguish three collections of internal wires connecting, respectively, the provide interface with the require interfaces, the provide interface with the service components and the service components with the require interfaces. This distinction leads to three arguments of type I, that partition internal wires depending on the types of the ports their tentacles are attached to. Correspondingly, production `sbod` has type $I \times C \times I \times I \to B$ (see Fig. 10). Production `abod` is very similar, but requires only a collection of internal wires. Its type is $C \times I \to AB$.

Again, suitable restrictions should be imposed on connecting wires when further aspects are in order. For instance, assume/guarantee relations between a require and a provide interfaces can be given as an entailment of the provide interface from the require one.

Service Components and Internal Wires. Service components are the main computational entities of SRML modules. We define two design productions to construct collections of such, possibly interconnected, service components: `comp` (of type $\to C$) to create a single component, and `comps` (of type $C \times C \times I \to C$) to compose two collections of components via internal wires (see Fig. 11).

Productions `iwire` $:\to I$ and `wires` $: I \times I \to I$ respectively build a single wire and a collection of wires (out of two collections of wires). Regarding `iwire`, observe that Fig. 12 actually represents all the productions obtained by attaching the leftward and rightward tentacles of the edge of type i to any of the exposed nodes. Production `nowire` accounts for empty wire collections.

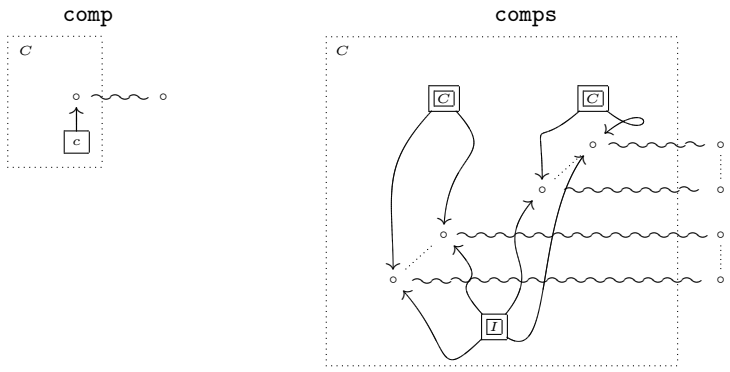

Fig. 11. An interconnection of service components.

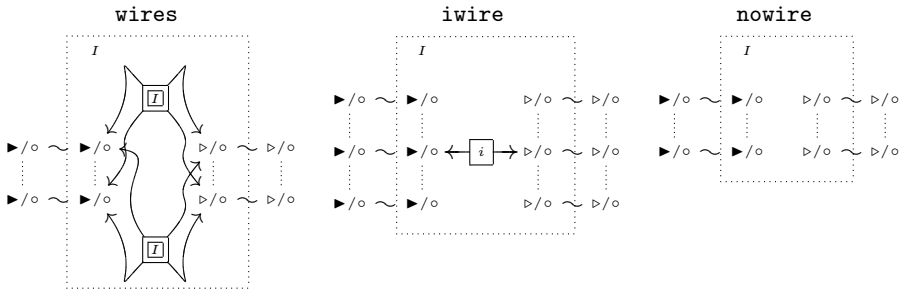

Fig. 12. Internal wires.

The ADR designs in Fig. 2 that encode the SRML diagrams in Fig. 1 are well-formed by the design terms $\texttt{m1} = \texttt{smod}(\texttt{b1}, \texttt{w1}, x)$ and $\texttt{m2} = \texttt{smod}(\texttt{b2}, x)$, where x is a variable of type W that models the non-discovered service, and:

$\texttt{b1} = \texttt{sbod}(\ \texttt{nowire},\ \texttt{comps}(\texttt{comp}, \texttt{iwire}, \texttt{comp}),\ \texttt{iwire},\ \texttt{wires}(\texttt{iwire}, \texttt{iwire})\ )$
$\texttt{w1} = \texttt{wrap}(\ \texttt{ewire},\ \texttt{smod}(\texttt{sbod}(\texttt{nowire}, \texttt{comp}, \texttt{iwire}, \texttt{nowire}))\ )$
$\texttt{b2} = \texttt{sbod}(\ \texttt{nowire},\ \texttt{comps}(\texttt{comps}(\texttt{comp}, \texttt{iwire}, \texttt{comp}), \texttt{iwire}, \texttt{comp}),\ \texttt{iwire},\ \texttt{iwire}\ ).$

4 Reconfiguration of SRML Diagrams

Rewriting of architectural designs can be used to define interesting reconfiguration mechanisms in SRML, like turning an assembly of modules into a composite module. The reconfiguration of an assembly into a module is called *composition*. It is typically applied during run-time while needed services are discovered and bound, but it can be applied at static time too, in order to avoid the run-time computational effort required for service discovery, selection and binding in dynamic composition.

The composition operation has already been sketched in § 1, where the assembly of the two modules on the left of Fig. 1 yields the single module on the

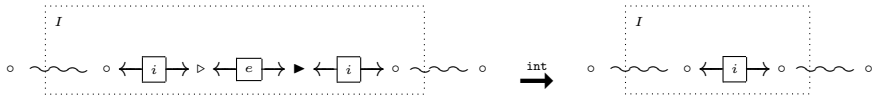

Fig. 13. Base rule that internalises a wires.

right. The only external wire EW has been *internalised*: the linked interfaces disappear and the components of both modules that were previously connected via those interfaces and the external wire are now directly connected via internal wires. SRML defines an algorithm that performs the composition by manipulating the involved SRML diagrams. The main idea is that each pair of internal wires connected via an external wire becomes an internal wire. However, no formal proof of compliance w.r.t. SRML's metamodel is provided.

Here, we encode internalisation of wrapped modules as proper ADR reconfigurations. The corresponding ADR rewrite rules transform a term representing any SRML diagram with a wrapped service into a term representing the diagram where the wrapped service has been internalised. We exploit an auxiliary design production `link`, which is very simple: it connects two collections of wires via an external wire (see, e.g. its use in Figure 14). Production `link` was not presented in § 3 because it is not really used to construct SRML diagrams and modules, but just needed in some rule premises to compute the internal wires to be inserted in the module.

The basic rule transforms the indirect connection of two ports into a direct, internal connection (see Fig. 13): $\texttt{link}(\texttt{iwire},\texttt{iwire}) \xrightarrow{\texttt{int}} \texttt{iwire}$.

Recall that we are dealing with the most abstract structural aspects, but SRML imposes further syntactical restriction on these rule as well semantic restrictions in binding services. Indeed, the interaction signature of the internal wire obtained by the rule should be the result of properly combining the interaction signatures of the wires appearing in the left-hand side of the rewrite.

The premises of the second rule (see Fig. 14) require each possible combination of internal wires to be properly internalised. If this is possible separately, then the overall internalisation is performed.

$$\frac{\texttt{link}(u_1,v_1) \xrightarrow{\texttt{int}} w_1^1 \;\; \texttt{link}(u_1,v_2) \xrightarrow{\texttt{int}} w_1^2 \;\; \texttt{link}(u_2,v_1) \xrightarrow{\texttt{int}} w_2^1 \;\; \texttt{link}(u_2,v_2) \xrightarrow{\texttt{int}} w_2^2}{\texttt{link}(\texttt{wires}(u_1,u_2),\texttt{wires}(v_1,v_2)) \xrightarrow{\texttt{int}} \texttt{wires}(\texttt{wires}(w_1^1,w_2^1),\texttt{wires}(w_1^2,w_2^2))}$$

Once we have presented the rule for internalising wires, we are ready to give the general rule for internalising a wrapped service (see Appendix A for the graphical representation). The rule takes into account the more general form of a design term with a wrapped service to be internalised:

$$\begin{array}{l}\dfrac{\texttt{link}(w_3,w_4) \xrightarrow{\texttt{int}} w_3^4 \;\; \texttt{link}(w_2,w_4) \xrightarrow{\texttt{int}} w_2^4 \;\; \texttt{link}(w_3,w_5) \xrightarrow{\texttt{int}} w_3^5 \;\; \texttt{link}(w_2,w_4) \xrightarrow{\texttt{int}} w_2^5}{\begin{array}{l}\texttt{smod}(\texttt{sbod}(c_1,w_1,\texttt{wires}(w_2,w_2'),\texttt{wires}(w_3,w_3')),\\ \quad\texttt{wrap}(\texttt{smod}(\texttt{sbod}(c_2,w_5,w_6,w_4),s_{n+1},\ldots,s_m),s_1,\ldots,s_n))\end{array}}\\ \longrightarrow \texttt{smod}(\texttt{sbod}(\texttt{comps}(c_1,w_2^5,c_2),\texttt{wires}(w_1,w_2^4),\texttt{wires}(\texttt{wires}(w_2',w_6),w_3^5),\\ \quad\texttt{wires}(w_3',w_3^4),s_1,\ldots,s_n,s_{n+1},\ldots,s_m)\end{array}$$

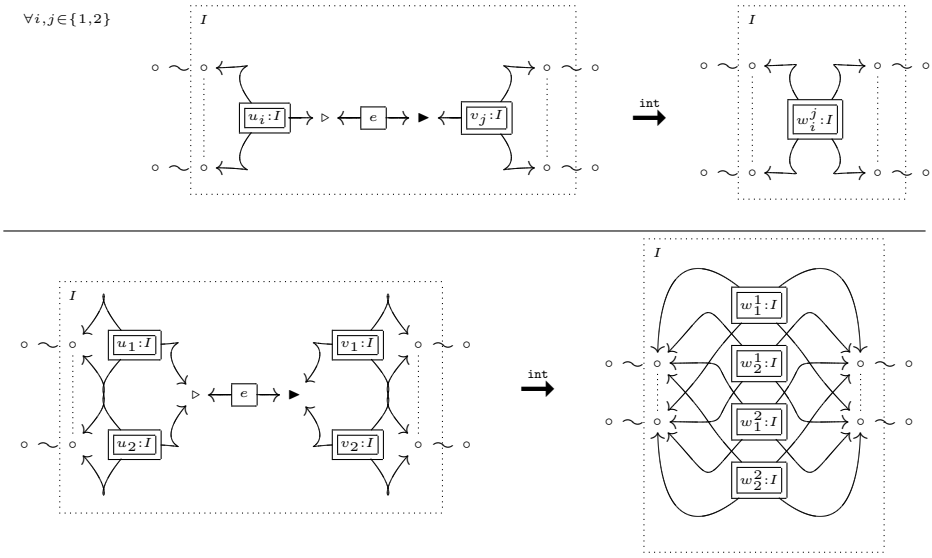

Fig. 14. Rule that internalises a wires: recursive case.

As an example of reconfiguration, it can be verified that `m1` is reconfigured into `m2` (cf. end of § 3 and Fig. 2) in one rewrite step by applying the above rule, where the only required premise is $\texttt{link}(\texttt{iwire},\texttt{iwire}) \xrightarrow{\texttt{int}} \texttt{iwire}$ (which is trivially satisfied).

It is worth noting that the composition rule can be applied in any context thus ensuring well-typedness and style-preservation.

5 Conclusion

We have proposed *Architectural Design Rewriting* as a framework for hierarchical style-based reconfigurations of software architectures. The approach is based on algebras of typed graphs with interfaces, yielding a unifying treatment of style-based design and reconfiguration. Its hierarchical and inductive features allows us to compactly represent complex reconfiguration rules. While in other approaches (e.g. [22]) correctness of each reconfiguration rule must be formally proved, in ADR correctness is automatically given by the fact that rewrites act on design terms, rather than on designs. Comparing ADR to architectural description languages, ADR offers a unifying model to represent architectural design, reconfiguration, and ordinary behaviour too. A deeper comparison of ADR against similar approaches can be found in [5].

In this paper we have defined an ADR style to support the design and reconfiguration of service-oriented specifications given in SRML whose choice is justified by the fact that it aims at providing a formal approach to service-oriented modelling that is close to SCA [25]. The model of SRML consists of an

ADR-based architectural style that is compliant with the SRML metamodel so that it can suitably define SRML complex reconfigurations with the main benefit they are compliant with the metamodel by construction.

We plan to analyse and eventually enrich our approach to support further issues inherent to the design and management of service-oriented architectures, like the treatment of modes [17] or the semantical information of SRML [14]. In addition we plan to perform a deeper comparative analysis of ADR against similar approaches like process calculi to deal with reconfigurable component based architectures [1], architectural metaprogramming initiatives [3] that promote the unifying treatment of software refactoring, synthesis and development as algebras over programs, and graphical representation of concurrent systems such as those based on process calculi encodings [15], Synchronized Hyperedge Replacement [11], and bigraphs [19]. An implementation of ADR in Maude [7] is also under development.

References

1. N. Aguirre and T. S. E. Maibaum. Hierarchical temporal specifications of dynamically reconfigurable component based systems. *Electr. Notes Theor. Comput. Sci.*, 108:69–81, 2004.
2. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. *Software and Systems Modeling*, 5(2):187–207, June 2006.
3. D. S. Batory. Program refactoring, program synthesis, and model-driven development. In S. Krishnamurthi and M. Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.
4. B. Benatallah, F. Casati, and F. Toumani. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54, 2004.
5. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style based reconfigurations of software architectures. Technical Report TR-07-17, Dipartimento di Informatica, Università di Pisa, 2007.
6. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In J. Palsberg and M. Abadi, editors, *POPL*, pages 209–220. ACM, 2005.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer Verlag, 2007.
8. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
9. T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
10. FETPI Global Computing project IST-2005-016004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers). `http://sensoria.fast.de`.
11. 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, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2005.

12. J. L. Fiadeiro, A. Lopes, and L. Bocchi. Algebraic semantics of service component modules. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *WADT*, volume 4409 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2006.
13. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *Proceedins of WS-FM'06, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.
14. J. L. Fiadeiro and V. Schmitt. Structured co-spans: An algebra of interaction protocols. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2007.
15. F. Gadducci. Graph rewriting for the π-calculus. *Mathematical Structures in Computer Scence*, 17(3):407–437, 2007.
16. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
17. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *Proceedings of EWSA'06, 3rd European Workshop on Software Architectures*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.
18. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *Electronic Notes on Theoretical Computer Science*, 109:97–109, 2004.
19. O. H. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.
20. B. König, U. Montanari, and P. Gardner, editors. *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems, 6.-11. June 2004*, volume 04241 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005.
21. I. Loulou, A. H. Kacem, and M. Jmaiel. Consistent reconfiguration for publish/subscribe architecture styles. In *Proc. of the First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*, 2007.
22. D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.
23. R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1992.
24. J. Misra and W. Cook. Orchestration computation: A basis for wide area computing. *Software and Systems Modeling*, 6(1):83–110, 2006.
25. Service Component Architecture. `http://osoa.org`.
26. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.

A Reconfiguration rule for module composition

The figure below depicts the left- (top) and right-hand sides of the rule for composing modules. The premises $\texttt{link}(w_3, w_4) \xrightarrow{\texttt{int}} w_3^4$, $\texttt{link}(w_2, w_4) \xrightarrow{\texttt{int}} w_2^4$, $\texttt{link}(w_3, w_5) \xrightarrow{\texttt{int}} w_3^5$ and $\texttt{link}(w_2, w_4) \xrightarrow{\texttt{int}} w_2^5$ are neglected from the figure due to space constraints (they are similar to those in Fig. 14).