A perspective on architectural re-engineering

Alejandro Sanchez[a,b], Nuno Oliveira[b], Luis S. Barbosa[b], Pedro Henriques[c]

[a] *Universidad Nacional de San Luis, San Luis, Argentina*
[b] *HASLab - INESC TEC & Universidade do Minho, Braga, Portugal*
[c] *CCTC, Universidade do Minho, Braga, Portugal*

Abstract

Continuous evolution towards very large, heterogeneous, highly dynamic computing systems entails the need for sound and flexible approaches to deal with system modification and re-engineering. The approach proposed in this paper combines an analysis stage, to identify concrete patterns of interaction in legacy code, with an iterative re-engineering process at a higher level of abstraction. Both stages are supported by the tools CoordPat and Archery, respectively. Bi-directional model transformations connecting code level and design level architectural models are ask defined. The approach is demonstrated in a (fragment of a) case study.

Keywords: software architecture, coordination patterns, re-engineering

1. Introduction

Legacy software has to be maintained, improved, replaced, adapted and regularly assessed for quality, which brings their re-engineering to the top of concerns of the working software architect. This is not, however, an easy task. On the one hand a systems' architecture relies more and more on non trivial coordination logic for combining autonomous services and components, often running on different platforms. On the other hand, often such a coordination layer is strongly weaved within the application at the source code level.

The CoordInspector tool [1, 2] was a first attempt to address this problem by systematically inspecting code in order to isolate the coordination threads from the computational layer. This is done in a semi-automatic way through the combination of generalised slicing techniques and graph manipulation.

Such a stage of *architectural discovery* constitutes a necessary, but not sufficient step in a re-engineering process. Actually, experience has shown that

- recovering an architectural model from code would be much more effective if driven by some notion of *pattern* encoding typical interactions;
- in any case, the low level model produced through slicing and code analysis, needs to be mapped to a more structural one, to precisely abstract and identify components and connectors and enable their re-engineering.

This paper, combining previous research on both *program understanding* and *software architecture*, addresses the challenge as follows:

- First of all it introduces a notion of a *coordination pattern* directly extracted from the program dependency graph of the legacy system, as well as a language, CoordL, to describe such patterns. A collection of coordination patterns constitutes a low level architectural description in terms of execution threads and interaction points. Its main purpose is to act as a template to inspect code and represent its coordination layer. CoordPat, a pattern search facility based on this idea, was combined with CoordInspector to enhance the tool support for the technique.
- Then a systematic method is proposed to translate such patterns into a high-level architectural model in Archery [3] which provides a proper setting for studying and simulating architectural changes. This iterative process is illustrated in 1 through the loop arrow from the Archery model.
- Finally, a reverse translation method is proposed to transform the new architectural model back to a collection of coordination patterns which guides the re-implementation process.

Figure 1 sums up the proposed approach for architectural re-engineering. The combination of CoordPat and Archery equips the architect with suitable tool-support for recovering architectural decisions, reconstructing an architectural model, and analysing the impact of different possible modifications. Since the two frameworks work at different abstraction levels, (the first providing abstractions over dependency graphs; the second entailing a *components-and-connectors* view of architectural organisation), 'translations' $\mathcal{A}$ and $\mathcal{C}$ in Figure 1 are central to the proposed method. Their application is illustrated in detail through an example, extracted from a real case study.

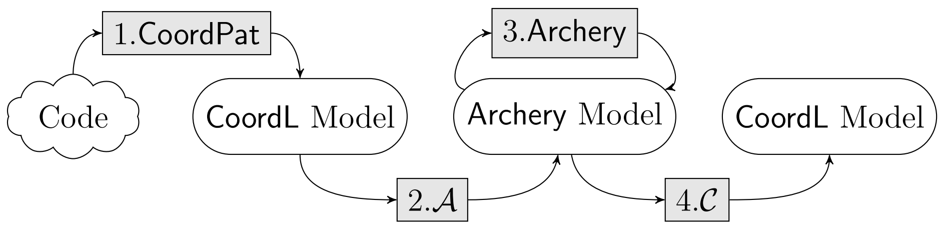


Figure 1: An approach to architectural re-engineering.

A main motivation for this work is the problem of quality assessment and re-engineering of Open Source Software (OSS) as discussed in [4]. Availability of code makes OSS particularly suited to application of backward analysis and program understanding techniques [5]. Often architectural decisions are only partially documented in OSS due to the *pay-as-you-go* documentation style and the distributed and heterogeneous nature of its development. Architectural re-engineering plays nevertheless a main role in OSS maintenance and evolution: it is particularly critical to endow OSS communities with techniques and tools to identify and to control architectural drift, *i.e.*, the accumulation of architectural inconsistencies resulting from successive code modifications, that may affect different quality attributes of the system.

The paper is organised as follows: section 2 describes the approach and the example we use to illustrate it; sections 3 and 4 introduce, respectively, CoordPat and Archery, the two main methods/tools in this process; section 5 describes the systematic translations of CoordL to Archery models and back; section 6 illustrates the approach through a detailed example; finally, section 7 reports on related work and concludes.

2. An approach to architectural re-engineering

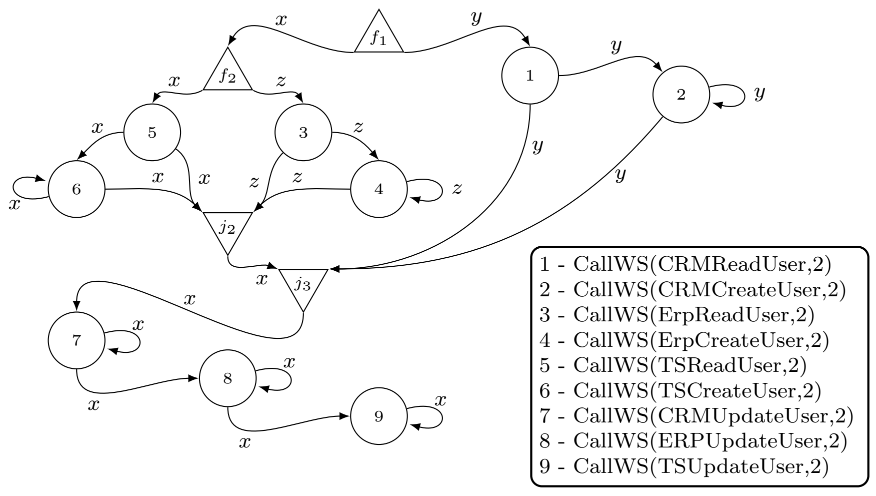
The approach proposed in this paper for architectural re-engineering of legacy code is depicted in Figure 1. As explained above, it resorts to the combination of a tool for extracting coordination patterns from source code (CoordPat) and a high level architectural description language (Archery) plus a guide to map patterns back and forth between these two levels.

The example chosen to illustrate the approach is part of a real case study on software integration described in [2]. It concerns a service to control the updating of user profiles and information common to a number of components of a company's information system. In its original formulation the context is that of a company offering professional training through e-learning courses. The information system comprises the following three main components: an Enterprise Resource Planner (ERP) for controlling expenses and profits; a Customer Relationship Management (CRM) for managing both general and customer-focused course campaigns; and a Training Server (TS) for managing the courses. These components worked almost independently, all information being shared by a set of scripts executed manually, which gave rise to frequent synchronisation problems. The decision to perform a global architectural analysis and reconstruction was pushed by a sudden growth in the company market share and the need for introducing a web portal for on-line sales.

CoordPat is first applied in the re-engineering process. The tool aims at uncovering, registering and classifying architectural decisions often left undocumented and hardwired in the source code. It implements a rigorous method [6] to extract the architectural layer which captures the system behaviour with respect to its network of interactions. This is often referred to as the *coordination layer*, a term borrowed from research on *coordination* models and languages [7] which emerged in the nineties to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

The extraction stage combines suitable slicing techniques to build a family of *dependency graphs* by pruning a *system dependency graph* [8] first derived from source code. After the extraction stage, the tool exploits such graphs to identify and combine instances of *coordination patterns* and then reconstruct the original specification of the system coordination layer. CoordPat maintains an incrementally-built repository of patterns to guide the analysis process.

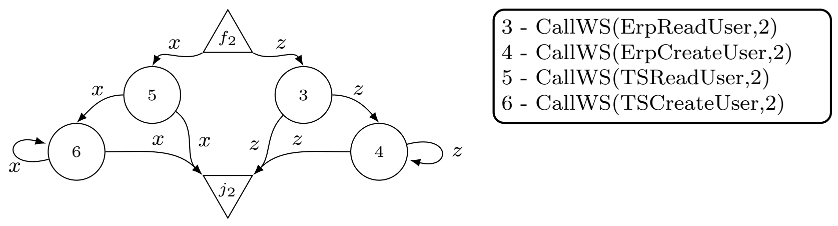
CoordPat coordination patterns are described in CoordL, a graph-based language with both a textual and a graphical representation. The later is almost self-explanatory. A base node is represented by a circle, a fork is represented by a triangle, a join by an inverted triangle, while a thread trigger is represented by a square. A greyed square is used for pattern instances. Edges are depicted by labelled, full arrows. Dashed arrows are used for connecting failed-synchronization nodes that come out of thread triggers. Consider, for example, the specification in Figure 2. It reproduces one of the coordination patterns recovered from the original system, which describes the user-updating service, that originates calls to user-read, -create and -update operations, offered by each of the three main components (CRM, ERP and



f_1
f_2
j_2
j_3
x
y
z
1 - CallWS(CRMReadUser,2)
2 - CallWS(CRMCreateUser,2)
3 - CallWS(ErpReadUser,2)
4 - CallWS(ErpCreateUser,2)
5 - CallWS(TSReadUser,2)
6 - CallWS(TSCreateUser,2)
7 - CallWS(CRMUpdateUser,2)
8 - CallWS(ERPUpdateUser,2)
9 - CallWS(TSUpdateUser,2)

Figure 2: A CoordL coordination pattern extracted from the original system.

TS). The fragment which includes nodes labeled as f_2, 3, 4, 5, 6 and j_2, is depicted in Figure 3 and will be identified as P_1 in the sequel. It executes the same task in two different components: node f_2 launches threads x and z that work with component ERP and with component TS. Each thread checks whether the information exists and creates it otherwise. Loops in nodes 4 and 6 represent iterations of creation attempts in case of failures. Node j_2 joins back both threads into a single one. A detailed description of CoordL is provided in section 3.



f_2
j_2
x
z
3 - CallWS(ErpReadUser,2)
4 - CallWS(ErpCreateUser,2)
5 - CallWS(TSReadUser,2)
6 - CallWS(TSCreateUser,2)

Figure 3: The P_1 fragment.

Archery [9, 3] is an architecture description language that emphasises systems' behavioural features and the relevant interaction protocols. The basic specification concept is that of an *architectural pattern*, which comprises a set of architectural elements (namely, connectors and components) specified by their behaviours and interfaces (set of ports). An architecture describes a particular configuration that instances of elements may assume and a set of attachments linking their ports or as a set of renamings making such ports externally visible. Both patterns and elements act as types for behaviours expected from instances, which are kept and referenced through typed variables. The language supports hierarchical composition.

The semantics of Archery is given by translation to a process algebra – mCRL2 [10]. The language also supports a reconfiguration layer whose semantics is given by bigraphical reactive systems [11]; that layer, however, falls out of the scope of this paper. Process algebra [12], broadly defined as *the study of the behaviour of parallel or distributed systems by algebraic means* [13], provides a suitable conceptual framework not only to describe software architectures, but also to *reason* about them either equationally (on top of well studied notions of behavioural equivalence), or through formulation and verification of behavioural requirements expressed in an associated modal logic. Moreover, it supports compositional reasoning and abstraction with respect to internal activity. The use of process algebra as an architectural description language is further explored in reference [14]. The mCRL2 notation is supported by a toolset [15] enabling simulation, visualisation, behavioural reduction and verification.

```
pattern WSPattern()
element WService()
  act rec, snd;
  proc Do() = rec.snd.Do();
  interface in xor rec; out xor snd;
element WSCaller()
  act rs, nt, snd, rec;
  proc Do() = rs.rec.snd.nt.Do();
  interface in xor rs,rec; out xor nt,snd;
end
ws : WSPattern = architecture  WSPattern()
  instances
    c1 : WSCaller = WSCaller(); c2 : WSCaller = WSCaller();
    s : WService = WService();
  attachments
    from c1.snd to s.rec; from c2.snd to s.rec;
    from s.snd to c1.rec; from s.snd to c2.rec;
  interface
    c1.rs as rs1; c1.nt as nt1; c2.rs as rs2; c2.nt as nt2;
end;
```

Listing 1: A pattern and an architecture example in Archery.

The language has an algebraic and a textual notation. While the former allows for the manipulation of models in a more succinct way, the latter includes common keywords from the software architecture domain, and aims at resulting more familiar to software engineers. Listing 1 shows an example architectural pattern expressed in the textual notation. It prescribes configurations arranged by instances of web services and their callers. Archery is described in section 4.

Equipped with these two tools, the re-engineering process proposed here, and illustrated in detail in section 6, comprises the following stages, as shown in Figure 1:

1. The architect uses CoordPat to extract the coordination model. This, expressed in the form of dependency graphs, allows the architect to study the network of interactions and to detect *coordination patterns*. It also provides enough information to spot problems and improvement opportunities, and to locate the source code associated to them.
2. The architect translates the model into Archery. He obtains a specification closer to the components-and-connectors view typically found in classical descriptions of software architecture.
3. Archery enables a richer description of the underlying system, and the architect exploits it by adding detail to the model. In particular, this provides a flexible setting to address problems and improvement opportunities, by inspecting the associated source code and modifying the model accordingly. The architect modifies the model to address detected issues, and studies the impact of the changes. He performs this study assisted by tools and records the relationships holding between the original and the modified models.
4. The architect translates the model back to CoordPat which guides the system re-implementation. Note that CoordPat can be used again on the re-engineered implementation and the whole process iterated if needed.

3. Architectural reconstruction with CoordPat

CoordPat is a tool for reverse architectural analysis, providing a systematic way to identify coordination patterns in source code. As explained in the Introduction, it is built on top of CoordInspector [1]. The latter provides mechanisms for building and slicing over several sorts of program graphs to identify in the source code all threads concerned with inter-component interaction and coordination. CoordPat adds an engine to define, store, update and identify specific patterns in source code relevant to the coordination layer of the system under analysis. It uses the CoordL language to specify coordination patterns and provides utilities for pattern discovery, editing and visual rendering. A pattern repository is integrated in the tool to support all these features and to be dynamically populated by the users. The next sub-section introduces the notion of a coordination pattern used in the tool. It underlies the CoordL language which is discussed afterwards.

3.1. Specifying coordination patterns

A coordination pattern is an abstraction over a program dependency graph $\mathcal{G}$ [6]. Conceptually, it is a graph defined over a set N of nodes, which abstracts program statements or activities, and a set of thread identifiers Thr which label the flow connections between nodes. Edges, on their turn, represent possible paths in the original, underlying graph $\mathcal{G}$.

Nodes are divided into base nodes, and control nodes which are specific to patterns to abstract control of execution threads. The latter arise from combinators *fork* and *join* discussed below. Each base node n is associated to a unique control thread $t(n)$. Similarly, edges in a pattern, which abstract paths in the underlying graph, are labelled with thread names, making explicit the control thread to which the path belongs.

Attached to this graph structure is an *interface* composed of two, not necessarily disjoint sets of nodes. One set represents *input points* in the pattern, to where external connections, with origin in other pattern instances, may be plugged in. The other represents *output points*, from where new edges may be defined to other pattern instances. This is known in the literature as a *graph with interface* [16]. Formally,

Definition 1. *A pattern is a tuple*

$$p \;=\; \langle N, in, out, T\rangle$$

where N is the set of nodes, subsets $in, out \subseteq N$ correspond to the pattern's input and output interfaces, respectively, and $T \subseteq N \times Thr \times N$ is a direct graph, often given as a family of binary relations indexed by thread references Thr.

A pattern is subject to an invariant that prevents any edge departing or arriving at a base node to be labelled with different atomic thread identifiers; the node cannot be part of two different threads. Thus,

$$n \textit{ is a } \text{base } \textit{node} \;\Rightarrow\; (n \xrightarrow{x} n' \vee n' \xrightarrow{x} n) \Rightarrow t(n) = x \tag{1}$$

This invariant means that if two base nodes m, n are connected then $t(m) = t(n)$. The simplest pattern is the single node

$$p = \langle \{n\}, \{n\}, \{n\}, \varnothing \rangle \tag{2}$$

Patterns can be aggregated by juxtaposition and connected by drawing new edges from a node in a pattern's output interface into a node in the input interface of the another one. Moreover, two input nodes of a pattern can be forced to join into a new node so as to provide a common entry point to two different paths in the pattern. Alternatively, this operation can be regarded as the fork of an input thread. Dually, nodes in the output interface can also be joined together, capturing the join of different threads coming out of the pattern. In the sequel we give a formal definition to each of these operations.

The aggregated pattern $p_1 \otimes p_2$, with $p_i = \langle N_i, in_i, out_i, T_i \rangle$ for $i \in \{1, 2\}$, is given by

$$p_1 \otimes p_2 = \langle N_1 \cup N_2, in_1 \cup in_2, out_1 \cup out_2, T_1 \cup T_2 \rangle. \tag{3}$$

In the sequel let $p = \langle N, in, out, T \rangle$. The link operator establishes a connection between two nodes (with the same thread reference) in a pattern interface. Formally,

$$(p) \leftarrow\!\urcorner^{i}_{j} = \begin{cases} \langle N, in \backslash \{i\}, out \backslash \{j\}, T \cup \{j \xrightarrow{t(i)} i\} \rangle & \Leftarrow \quad i \in in, j \in out, t(i) = t(j) \\ \qquad\qquad\qquad\qquad\qquad p & \Leftarrow \quad \textit{otherwise} \end{cases} \tag{4}$$

The remaining pattern combinators are intended to glue together two nodes in the input (respectively, output) interface. The first combinator is *fork*: two input nodes are made internal and replaced in the input interface by a single node, say f, which acquires the thread reference from the combinators first (or upper) argument. Because f must be a base node itself, a control node $\overline{f}$ is added as depicted in Figure 4. Formally,

$$\begin{aligned} f \prec\!\!\!\lhd\, {}^{a}_{b}(p) &= \langle N \cup \{f, \overline{f}\}, \{f\} \cup in \backslash \{a, b\}, out, \\ &\qquad T \cup \{f \xrightarrow{t(a)} \overline{f}, \overline{f} \xrightarrow{t(a)} a, \overline{f} \xrightarrow{t(b)} b\} \rangle \end{aligned} \tag{5}$$

$$\begin{tikzcd}
 & & a \\
f \arrow[r, "t(a)"] & \overline{f} \arrow[ur, "t(a)"] \arrow[dr, "t(b)"'] & \\
 & & b
\end{tikzcd}$$

(fork)

$$\begin{tikzcd}
a \arrow[dr, "t(a)"] & & \\
 & \overline{f} \arrow[r, "t(a)"] & f \\
b \arrow[ur, "t(b)"'] & &
\end{tikzcd}$$

(join)

$$\begin{tikzcd}
a \arrow[dr, "t(a)"] & & f \\
 & \overline{f} \arrow[ur, "t(a)"] \arrow[dr, "t(a)"'] & \\
b \arrow[ur, "t(b)"'] & & g
\end{tikzcd}$$

(trigger)

Figure 4: *Fork*, *join* and *trigger*.

where f is a fresh node identifier. Let x be the thread associated to a and f, and y the thread associated to b. Note that, at a later stage, a node n can be linked to f to represent the forking of thread x into itself and y.

The combinator dual to *fork* is *join*: two output nodes are made internal and replaced in the output interface by a special node which, again, acquires the thread reference from the combinator (or upper) argument, as depicted in Figure 4. Formally,

$$\begin{aligned}(p)\,{}^{a}_{b}\!\rightarrowtail f = \langle N \cup \{f, \overline{f}\}, in, \{f\} \cup out \backslash \{a, b\},& \\ T \cup \{a \xrightarrow{t(a)} \overline{f}, b \xrightarrow{t(b)} \overline{f}, \overline{f} \xrightarrow{t(a)} f\}\rangle& \qquad (6)\end{aligned}$$

The last pattern combinator is the *thread trigger* $(p)\,{}^{a}_{b}\!\rightarrowtail\!\!\!\leftarrowtail{}^{i}_{j}$. It acts like a join, joining two output nodes a and b and acquiring its thread reference from the 'upper' argument. Unlike join, however, it provides two new nodes, both labelled with the same thread reference. The 'upper' node, f, represents a synchronization of both threads (after execution of the statements abstracted in a and b), just as a normal join node. The 'lower' node, g, however, represents absence of synchronization: control goes from a to j without previous synchronization with b in its thread. Graphically, any connection from this node is depicted as a dashed line. Formally, the combinator effect on the pattern design is given by

$$\begin{aligned}(p)\,{}^{a}_{b}\!\rightarrowtail\!\!\!\leftarrowtail{}^{f}_{g} = \langle N \cup \{f, g, \overline{f}\}, in, \{f, g\} \cup out \backslash \{a, b\},& \\ T \cup \{a \xrightarrow{t(a)} \overline{f}, b \xrightarrow{t(b)} \overline{f}, \overline{f} \xrightarrow{t(a)} f, \overline{f} \xrightarrow{t(a)} g\}\rangle& \qquad (7)\end{aligned}$$

Without loss of generality, combinators *link*, *join* and *thread trigger* may have, instead of a single node, a list L of nodes as possible sources for their incoming connections.

With these combinators the fragment P_1 of the coordination pattern shown in Figure 3, and described in section 2, can be specified as follows:

$$P_1 \;= f_2 \prec\!\!\!\lessdot\, {}^{3}_{5}\ \big((3 \otimes 4) \looparrowleft^{4}_{\{3,4\}} \otimes (5 \otimes 6) \looparrowleft^{6}_{\{5,6\}}\big)\ {}^{\{3,4\}}_{\{5,6\}} \rightarrowtail j_2 \tag{8}$$

A number of structural properties of these combinators are trivial to prove. For example, for $\cong$ denoting graph isomorphism,

Lemma 1.

$$p_1 \otimes p_2 \;\cong\; p_2 \otimes p_1 \tag{9}$$
$$(p_1 \otimes p_2) \otimes p_3 \;\cong\; p_1 \otimes (p_2 \otimes p_3) \tag{10}$$
$$(p \looparrowleft^{i}_{j}) \looparrowleft^{l}_{k} \cong (p \looparrowleft^{l}_{k}) \looparrowleft^{i}_{j} \tag{11}$$

Proof. All results are immediate by unfolding the definition of both combinators. The first two come from commutativity and associativity of set union. For the last one, note that in both cases the input and output interfaces are the same and so is the transition structure $(T \cup \{j \xrightarrow{t(i)} i, k \xrightarrow{t(l)} l\})$. □

Patterns can also be ordered by the existence of a *simulation* relating the underlying transition structures. Formally,

Definition 2. *Two patterns p_1 and p_2 are* similar*, denoted by $p_1 \lessapprox p_2$ iif $in_1 \subseteq in_2$, $out_1 \subseteq out_2$ and there is a* simulation *of T_1 into T_2 relating each node in in_1 to the equally named node in in_2. A relation $R \subseteq T_1 \times T_2$ is a* simulation *iff, whenever $\langle n, m\rangle \in R$,*

$$n \xrightarrow{x}_{T_1} n' \;\Rightarrow\; \exists_{m' \in N_2} \,.\, m \xrightarrow{x}_{T_2} m' \;\wedge\; \langle n', m'\rangle \in R$$

A simulation R whose converse is also a simulation is called a bisimulation*. Patterns p_1 and p_2 are said to be* bisimilar*, denoted by $p_1 \approx p_2$ iff $in_1 = in_2$, $out_1 = out_2$, and there is a* bisimulation *over T_1 and T_2 relating each node in in_1 to the equally named node in in_2.*

The existence of a bisimulation between two patterns means they have identical interfaces and exhibit the same transitional behaviour. In general, similarity and bisimilarity provide a guide for comparing and classifying patterns.

Lemma 2. *Let $p_1 \approx p_2$. Then,*

$$\begin{aligned}
p_1 \otimes p &\approx p_2 \otimes p && (12)\\
f \prec {}^{a}_{b}(p_1) &\approx f \prec {}^{a}_{b}(p_2) && (13)\\
(p_1)\,{}^{a}_{b} \rightarrowtail f &\approx (p_2)\,{}^{a}_{b} \rightarrowtail f && (14)\\
(p_1)\,{}^{a}_{b} \rightarrowtail\!\!\prec {}^{f}_{g} &\approx (p_2)\,{}^{a}_{b} \rightarrowtail\!\!\prec {}^{f}_{g} && (15)
\end{aligned}$$

Proof. The proof of (12) is trivial and in all cases equality of interfaces is easy to check. For (13), let R be a bisimulation witnessing $p_1 \approx p_2$. Consider $R' = R \cup \{(f,f),(\overline{f},\overline{f})\}$. The unique transition from f in the first system, labelled by $t(a)$, is matched by a unique, equally labelled transition in the second, both to $\overline{f}$. Form there in both cases there is a unique transition to a, labelled by $t(a)$ and to b, labelled by $t(b)$. By assumption pairs (a,a) and (b,b) are already in R, which makes R' a bisimulation as well. For (14), let again R be a bisimulation witnessing $p_1 \approx p_2$. If $\{(a,a),(b,b)\} \nsubseteq R$, R is still a bisimulation for joined pattern. If not, add pairs (f,f) and $(\overline{f},\overline{f})$. Transitions from a or b will match in both patterns, as well as the unique transition from $\overline{f}$ to f. In both patterns, f is an output node with no further transitions, which concludes the proof. The prove of (14) follows a similar argument. □

Bisimilarity is not, however, a congruence because it is not preserved by the link operator.

3.2. CoordL – the language

Based on the notion of a coordination pattern, introduced above, the CoordL language admits a textual and graphical notation as a concrete interface to both the CoordPat tool and the working software architect.

A pattern definition in CoordL follows the template in Listing 2. Note that each pattern has an identifier (`pattern_id`) and a set of input/output ports as its interface. The later is declared inside a ()-block, with a bar '|' separating the *in* ports and, in the right-hand-side, the *out* ports. The union of these sets must not be empty, but are not required to be disjoint.

```
pattern_id (p1, p2 | p1, p3) { [DECLARATIONS] [PATTERN GRAPH] }
```

Listing 2: Pattern declaration.

The {}-block has two parts. The first one is reserved for declaring nodes, basically selecting information (such as the corresponding code fragment, type of interaction or calling discipline) from the underlying dependency graph $\mathcal{G}$. The second part specifies the pattern graph structure. Before proceeding, the reader may want to inspect, in Listing 3, the CoordL code corresponding to coordination pattern P_1 (see Figure 2).

```
P1_pattern(f2 | j2){
  node 3,4,5,6 = { st==”...” && ct==webservice &&
    cm==async && cr==consummer };
  fork f2;  join j2;  root f2;
  { f2 -(x,z)-> (5, 3) } @[ | 5,3]
  { 5-x->6, 5-x->j2, 6-x->6, 6-x->j2,
    3-z->4, 3-z->j2, 4-z->4, 4-z->j2 } @ [5,3 | 6,4] }
  { (6,4) -(x,z)-> j2 } @[ | 6,4]
```

Listing 3: CoordL code for coordination pattern P_1.

Nodes declaration. As discussed in section 3.1 there are two possible types of nodes in a coordination pattern: *base* nodes and *control* nodes. The former come directly from the underlying dependency graph and are supposed to describe fragments of coordination code. Each one is declared with four *attributes* (combined in conjunction (&&) or disjunction (||)).

- *Statement* (`st`): reference to the coordination code fragment abstracted in the node. This is typically described by a regular expression acting as filter over source code during the construction of the dependency graph $\mathcal{G}$ (see [6] for details).
- *Type* (`ct`): defines the type of the coordination primitive the code fragment implements. Such types are assigned during the construction of $\mathcal{G}$: typical examples are `webservice` (for a web service call) or `rmi` (for a remote method invocation).
- *Method* (`cm`): defines the mode in which the request is made. It can be either `sync`(hronous) or `async`(hronous).
- *Role* (`cr`): describes the role of the component that is requesting the service. It can be either a `consumer` or a `producer`.

Example node, fork, and join declarations are shown in Listing 3. Control nodes are introduced by the different variants of *fork* and *join* and *tread trigger* combinators.

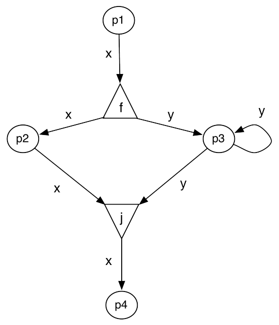
Besides basic and control nodes, CoordL introduces a third type of node which abbreviates a *pattern instance.* This is created as in the example shown in Listing 4, assuming a previous declaration of the corresponding pattern. The pattern name is used to identify the type of its instances. Notice that, in creating a pattern instance both node and thread identifiers can be instantiated to actual values.

```
pattern1(p1 | p2, p3){ ...}
pattern2(p1, p2 | p3, p4){ ...}
...
pattern1 instance1(pi1 | po1, po2) ;
pattern2 instance2(pi2, pi3 | po3, po4) ;
```

Listing 4: Instance declaration.

This abbreviation allows for the hierarchical construction of patterns. All connections, however, are always made with explicit reference to the relevant interface nodes, as detailed below.

```
my_patt(p1 | p4){
  node p1, p2, p3, p4 = {
    st == "Invoke(*)" &&
    ct == webservice &&
    cr == consumer &&
    cm == sync };
  fork f;
  join j;
  root p1;
  {f -(x,y)-> (p2,p3)}@[p3|]
  {p1 -> f, p3 -y-> p3}@[|p3]
  {(p2,p3) -(x,y)-> j}
  {j -x-> p4}
}
```



```
new_patt(p1 | p2, p3){
  node p1, p2, p3 = {
    st == "Get(*)" &&
    ct == webservice &&
    cr == consumer &&
    cm == sync };
  my_patt inst(pi | po);
  root p1;
  {p1 -x-> inst(pi | po)}
  {po -x-> p2}@[|po]
  {po -x-> p3}
}
```

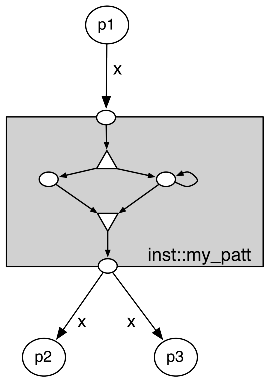


Figure 5: A hierarchical pattern description.

Edges specification. Links between input and output nodes in a pattern are specified with a direct syntax in which an arrow, suitably labelled, is explicitly written. This corresponds to a simple way of implementing combinator $\hookleftarrow$ in section 3.1. As mentioned there, an edge has a very precise meaning with reference to the underlying dependency graph $\mathcal{G}$. For instance, connection: `p1 -x-> p2` means that information flows from node `p1` to node `p2` in a thread identified by `x` through an unspecified number of mediating edges in $\mathcal{G}$. Connections to or from *control* nodes are illustrated, for *fork* and *join* in Listing 3, and for *thread trigger* in Listing 5. They implement, as expected, combinators $\prec\!\!\lhd$, $\rightarrowtail$ and $\rightarrowtail\!\!\prec\!\!\lhd$. Notice all new node identifiers, such as `f`, `j` or `m` have to be previously declared.

```
{(pa, pb) -(x,y)-> m,  m.sync -x-> pc, m.fail -x-> pd}
```

Listing 5: Usage of control nodes.

Moreover, symbol '@' followed by a list of nodes separated by symbol '|' is used to make both input and output ports of nodes or patterns alive (i.e., open), once they were used in the previous list of edges definition. This constructor is used to keep consistent nodes and pattern ports.

Figure 5 shows two coordination patterns in CoordL, and the corresponding graphical representation. The first pattern, identified by `my_patt`, illustrates the composition of base nodes, forks and joins. The second one, identified by `new_patt`, introduces the declaration of pattern instances and their (hierarchical) composition. Note that in the graphical rendering of the second pattern a grey square that represents the inclusion of a pattern instance.

4. Architectural modelling with Archery

Archery [9, 3] is a high-level architectural description language. This section describes the language, briefly explains its behavioural semantics the the relations used for architectural analysis. The architectural pattern for web services, shown in Listing 1, is used for illustration purposes.

4.1. Modelling architectural patterns

4.1.1. Patterns and elements

A specification of an Archery model (*Spec*) comprises one or more patterns P, a main architecture referenced by a variable Var and global data specifications D. A description of the latter is omitted here because they coincide with mCRL2 data types.

$$\begin{aligned}
Spec &= \mathcal{P}(P) \times Var \times D \\
P &= IdP \times Fp^* \times \mathcal{P}(E) \\
Fp &= IdPar \times DataType \\
E &= IdE \times Fp^* \times MPrc \times \mathcal{P}(Prc) \times \mathcal{P}(Prt) \\
MPrc &= IdPrc \times Fp^* \times Val^* \times \mathcal{P}(Act) \times Body \\
Prc &= IdPrc \times Fp^* \times \mathcal{P}(Act) \times Body \\
Act &= IdAct \times Domain^* \\
Prt &= IdPrt \times Dir \times PrtType, \text{ with } IdPrt \subseteq IdAct
\end{aligned}$$

where $\mathcal{P}(X)$ denotes the powerser of X. An *architectural pattern* P defines the basic building blocks of a family of architectures. It includes a unique identifier, an optional list of formal parameters Fp and one or more architectural elements E. In turn, each Fp has an identifier and a data type. For instance, tuple wsp below corresponds to pattern $WSPattern$ between lines 1 and 10 in Listing 1.

$$\begin{aligned}
wsp &= (WSPattern, [\,], \{ce, se\}) && (16)\\
ce &= (WSCaller, [\,], && (17)\\
&\quad (Do, [\,], [\,], \{rs, snd, rec, nt\}, \text{``}rs.snd.rec.nt.Do\text{''}),\\
&\quad \{\}, \{rs, in, xor), (snd, out, xor), (rec, in, xor), (nt, out, xor)\})\\
se &= (WService, [\,], (Do, [\,], [\,], \{rec, snd\}, \text{``}rec.snd.Do\text{''}), && (18)\\
&\quad \{\}, \{(rec, in, xor), (snd, out, xor)\})
\end{aligned}$$

An *architectural element* E models either a component or a connector. It is described by an identifier, an optional list of formal parameters, a description of its *behaviour* and an *interface*. The behaviour consists of a *main process* $MPrc$, which describes the initial behaviour, and a set of processes Prc referenced from it. Tuple $MPrc$ comprises an identifier, a list of formal parameters, a list of initial expressions, matching in order and type the formal parameters, a set of actions Act, and a process expression $Body$ specified in a slightly modified subset of mCRL2. An action Act has an identifier and an optional list of mCRL2 domains. For instance, the main process of element *WSCaller* is identified by Do, with no arguments, and defines actions rs, *snd*, *rec*, *nt* that respectively represent the events of receiving a signal, sending a request, receiving a response, and notifying termination. Its process expression specifies an iteration of the sequence of these four actions.

The interface, on the other hand, contains one or more ports Prt. Each *port* defines an identifier, which must match the identifier of an action in any of the element processes, a *direction* Dir, and a *port type* $PrtType$. The direction can be either *in* or *out* and indicates how data along the attached ports flows. Ports are synchronous; however, a suitable process algebra expression can be used to emulate any other port behaviour. The *port type* indicates how many participants are necessary for a communication to take place, and can be either *and*, *xor*, or *or*. While an *and* port requires all attached participants to synchronise, a *xor* port requires exactly one, and an *or* port at least one.

4.1.2. Pattern and element instances

A *variable* Var is a placeholder for instances. It has an identifier, a type that must match an element or pattern identifier, and an *instance* $Inst$ as a value. $Inst$ is either a distinguished value (inactive process) represented by singleton 1, an element instance $EInst$, or a pattern instance $PInst$. Instances may not match the variable's type but they must match the interface defined by such a type.

$$\begin{aligned}
Var &= IdVar \times IdType \times Inst \text{ with } IdType = IdP + IdE\\
Inst &= 1 + EInst + PInst\\
Einst &= IdE \times Val^*\\
PInst &= IdP \times Val^* \times \mathcal{P}(Var) \times \mathcal{P}(Att) \times \mathcal{P}(Ren)\\
Att &= PR \times PR\\
Ren &= IdPrt \times PR\\
PR &= IdVar \times IdPrt
\end{aligned}$$

For example, variable ws in the tuple wsv contains a configuration of pattern $WSPattern$. The corresponding textual notation for this tuple is shown between lines 11 and 20 in Listing 1.

$$\begin{aligned}
wsv =&(ws, CoordL, (CoordL, [], \{(s, WService, (WService, [\,])),\\
&(c1, WSCaller, (WSCaller, [\,])),\\
&(c2, WSCaller, (WSCaller, [\,]))\}, As, Rs))
\end{aligned} \tag{19}$$

$$\begin{aligned}As =&\{((c1, snd), (s, rec)), ((c2, snd), (s, rec)), ((s, snd), (c1, rec)),\\&((s, snd), (c2, rec))\},\\Rs =&\{((c1, rs), rs1), ((c1, nt), nt1), ((c2, rs), rs2), ((c2, nt), nt2)\}\end{aligned}$$

An *element instance EInst* has an identifier that matches an element name and a list of actual parameters matching in order and type the formal parameters.

An architecture, or *pattern instance PInst*, defines a set of variables and describes the configuration adopted by their instances. It contains a token that must match a pattern name; an optional list of actual arguments; a set of variables; an optional set of attachments; and an interface. The actual arguments must match in type and order those of the pattern acting as its type. The type of each variable in the set must is an element defined in the pattern of which the architecture is an instance.

An *attachment Att* includes a *port reference* to an *out* port and another one to an *in* port. Each *port reference PR* is an ordered pair of identifiers corresponding to the variable and its instance, respectively. Thus it specifies which *out* port communicates with which *in* port — see lines 16 and 17 in Listing 1.

The architecture interface is a set of *port renamings Ren*. Each *port renaming* contains a port reference and a token with the external name of the port. Ports not included in this set are not visible from the outside. Note that ncluding the same port in an attachment and the interface is incorrect.

4.2. Combinators

Architectural patterns and their instances' (re)configurations are described by combinators, which include tuple constructors, update operations, script application and an architectural product. Tables 1 and 2 provide an (informal) overview of the language combinators. See [17] for the formal definitions.

Each tuple has an associated set of constructors with identical name, but written in lowercase letters. There is always a default constructor restricted to the mandatory components. For instance, the constructors for an architectural pattern (P), directed by the corresponding signature, are as follows

$$\begin{aligned}&p : IdP \to P && p : IdP \times Fp^* \to P\\&p : IdP \times \mathcal{P}(E) \to P && p : IdP \times Fp^* \times \mathcal{P}(E) \to P.\end{aligned}$$

Pattern construction

Constructors with signature $t : idT \rightarrow T$ receive an identifier and return a tuple with default values in each component, where t/T stands for either p/P (pattern), e/E (element), $mprc/MPrc$ (main process), prc/Prc (process), act/Act (action), or prt/Prt (port)

$\hookleftarrow_E$: $P \times E \rightarrow P$	adds an element to a pattern
$\not\hookleftarrow_E$: $P \times IdE \rightarrow P$	removes an element from a pattern
$\hookleftarrow_E$: $P \times \mathcal{P}(E) \rightarrow P$	adds a set of elements to a pattern
$\not\hookleftarrow_E$: $P \times \mathcal{P}(IdE) \rightarrow P$	removes a set of elements from a pattern
$\ast : E \times MPrc \rightarrow E$	replaces the main process of an element
$\circledast : E \times Prc \rightarrow E$	adds a process to an element
$\not\circledast$: $E \times IdPrc \rightarrow E$	removes a process from an element
$\circ$, $\square$, $\diamond : E \times IdPrt \rightarrow E$	adds an And (Or, Xor, resp.) In port to an element
$\bullet$, $\blacksquare$, $\blacklozenge : E \times IdPrt \rightarrow E$	adds an And (Or, Xor, resp.) Out port to an element
$\circ$, $\square$, $\diamond : E \times \mathcal{P}(IdPrt) \rightarrow E$	adds And (Or, Xor, resp.) In ports to an element
$\bullet$, $\blacksquare$, $\blacklozenge : E \times \mathcal{P}(IdPrt) \rightarrow E$	adds And (Or, Xor, resp.) Out ports to an element
$\not\circ$: $E \times IdPrt \rightarrow E$	removes a port from an element

Table 1: Archery's algebraic notation for patterns.

Update operations are available for each component of each tuple. They all receive the original tuple and the component to modify, returning the updated tuple. The sort of updates available depend on the component's type. Three variations are provided according to the modified tuple component. If it is a *set* or a *list*, *add* and *remove* operations are considered in which the second argument contains, respectively, the element to add, or the identifier of the element to remove. If, on the contrary, the component is of a *non-collection* type, only a *replace* operation is available. The basic combinators required for the case study discussed in this paper, are described below. Unless explicitly stated, all operators are infix, and their type and effect is assumed to fall in one of the aforementioned variations according to the component type.

Combinators related to patterns include addition ($\hookleftarrow_E$) and removal ($\not\hookleftarrow_E$) of an element. A distributed (and overloaded) version for these operators operators upon a set $\mathcal{P}(E)$, and iteratively calls the original operation for each element in it.

Combinators for elements modify both their behaviour and interface. The

Instance construction	
$var : IdVar \times IdType$	creates a variable
$einst, pinst : IdType$	creates an element (resp. pattern) instance
$pr : IdVar \times IdPrt$	creates a port reference
$ren : IdPrt \times PR$	creates a renaming
$att : PR \times PR$	creates an attachment
$\boxed{\square} : PInst \times Var \rightarrow PInst$	adds an instance to a pattern instance
$\boxed{\not\square} : PInst \times IdVar \rightarrow PInst$	removes an instance from a pattern instance
$\boxed{\square} : PInst \times \mathcal{P}(Var) \rightarrow PInst$	adds instances to a pattern instance
$\boxed{\not\square} : PInst \times \mathcal{P}(IdVar) \rightarrow PInst$	removes instances from a pattern instance
$\boxed{\div} : PInst \times Att \rightarrow PInst$	adds an attachment to a pattern instance
$\boxed{\not\div} : PInst \times Att \rightarrow PInst$	removes an attachment from a pattern instance
$\boxed{\div} : PInst \times \mathcal{P}(Att) \rightarrow PInst$	adds attachments to a pattern instance
$\boxed{\not\div} : PInst \times \mathcal{P}(Att) \rightarrow PInst$	removes attachments from a pattern instance
$\boxed{\triangle} : PInst \times Ren \rightarrow PInst$	adds a port renaming to a pattern instance
$\boxed{\not\triangle} : PInst \times IdPrt \rightarrow PInst$	removes a port renaming from a pattern instance
$\boxed{\triangle} : PInst \times \mathcal{P}(Ren) \rightarrow PInst$	adds a set of port renamings to a pattern instance
$\boxed{\not\triangle} : PInst \times \mathcal{P}(IdPrt) \rightarrow PInst$	removs port renamings from a pattern instance
$\boxed{\circ} : Var \times Inst \rightarrow Var$	sets the value of a variable
$\boxed{\circ}_{Id} : Var \times Inst \rightarrow Var$	sets the value of a variable Id in a pattern inst.
$\boxtimes : PInst \times PInst \rightarrow PInst$	architectural product
$\boxtimes : Var \times Var \rightarrow Var$	architectural product of the variable's values

Table 2: Archery's algebraic notation for instance construction.

former group includes the replacement of the main process ($\ast$) and the addition ($\circledast$) and removal ($\not\circledast$) of processes. For the latter, different symbols are used to distinguish combinators for adding ports according to type and direction: $\circ$, $\bullet$, $\square$, $\blacksquare$, $\diamond$, $\blacklozenge$, where circles, squares, and diamonds correspond to *and*, to *or* and to *xor* types, and filled (respectively, hollow) symbols indicate the *out* (respectively, *in*) direction. The port removal combinator is indicated with symbol $\not\circ$. Distributed versions of these operators are also defined.

Using these combinators pattern $WSPattern$, described as a tuple in expression (16), and textually in Listing 1, is written

$$wsp = p(WSPattern) \hookleftarrow_E \{ce, se\} \tag{20}$$

$$\begin{aligned} ce = {} & e(WSCaller) \blacklozenge \{snd, nt\} \diamond \{rec, rs\} \\ & \ast\, mprc(Do, \{rs, snd, rec, nt\}, \text{“}rs.snd.rec.nt.Do\text{”}) \end{aligned} \tag{21}$$

$$se = (WService) \diamond rec \blacklozenge snd \circledast mprc(Do, \{rec, snd\}, \text{``}rec.snd.Do\text{''}) \quad (22)$$

Pattern instances are updated by adding ($\boxed{\square}$) or removing ($\boxed{\not\square}$) instances, both admitting distributed (and overloaded) versions. And, similarly, to add ($\boxed{\div}$) and remove ($\boxed{\not\div}$) attachments and renamings ($\boxed{\vartriangle}$ and $\boxed{\not\vartriangle}$). There is also an operation to replace ($\boxed{\circ}$) the value in a variable. When the first argument is a pattern instance, this can also be used with an identifier *id* (as in $\boxed{\circ}_{id}$) that indicates the value of the inner variable *id* to be replaced. For example, the expression below describes a $WSPattern$ configuration in which a web service is connected to two callers. Notice that, in variable tuples a component is missing; 1 is the assumed value, as in $(s, WService) = var(s, WService) = (s, WService, 1)$.

$$\begin{aligned}
wsv &= var(ws, WSPattern) \boxed{\circ}\, pinst(WSPattern) \\
&\quad \boxed{\square} \{si, ci_1, ci_2\} \boxed{\div}\, atts \boxed{\vartriangle}\, rs \qquad (23) \\
si &= (s, WService) \boxed{\circ}\, einst(WService) \\
ci_1 &= (c1, WSCaller) \boxed{\circ}\, einst(WSCaller) \\
ci_2 &= (c2, WSCaller) \boxed{\circ}\, einst(WSCaller) \\
atts &= \{((c1, snd), (s, rec)), ((c2, snd), (s, rec)), \\
&\quad ((s, snd), (c1, rec)), ((s, snd), (c2, rec))\} \\
rs &= \{((c1, rs), rs1), ((c1, nt), nt1), ((c2, rs), rs2), ((c2, nt), nt2)\}
\end{aligned}$$

Architectural product, $\boxtimes : PInst \times PInst \to PInst$, combines two architectures of the same type into a single one, putting them side by side. For convenience, an operator ($\boxtimes : Var \times Var \to Var$) that returns and takes variables as arguments is defined. The types and values of the variables must coincide. Argument variables are discarded: the identifier of the returned one derives from the argument variables.

Scripts take a list of arguments and can be applied to a specific configuration. Expression (24) defines a script *node* that takes an argument n and can be applied to architecture x. It creates an instance of pattern $WSPattern$ in variable $vpId_n$, creates variable id_n and assigns an instance of $WSCaller$ to it. It also renames its ports and makes them externally visible, according to R. This allows the resulting architecture to initiate action, notify termination, send a request, and receive a response, respectively. Script application is denoted by $\rhd$. Then, applying script *node* to an empty architecture is written as $node(n) \rhd 1$, or simply as $node(n)$.

$$\begin{aligned}
node(n)(x) \triangleq\ & var(vpId_n, WSPattern) \boxed{\circ}\, pinst(WSPattern) \\
& \boxed{\square}\, v_n \boxed{\triangle}\, R \qquad (24) \\
v_n =\ & (id_n, WSCaller) \boxed{\circ}\, einst(WSCaller) \\
R =\ & \{((id_n, rs), rs_n), ((id_n, nt), nt_n), \\
& ((id_n, snd), snd_n), ((id_n, rec), rec_n)\}
\end{aligned}$$

4.3. Behavioural semantics

The behavioural semantics of Archery is given through a translation $\mathcal{T}$ into a mCRL2 specification (see [9] for details). Let us illustrate this with our running example. Each element instance is translated to (at least) two processes, one calling the other, defined by unique identifiers. For example, the translation of the web service instance, referenced by variable s in our example (see expression 19, or expression 23, or line 14 in Listing 1), results in the two processes specified between lines 1 and 3 of Listing 6 (for brevity, action declarations are omitted). If the instance as an initial value given by an expression, the caller uses it as an actual parameter. Other processes defined within the element are recursively translated.

```
1  proc WService_s_init = Do_s;
2  proc Do_s = (rec_s_snd_c1 + rec_s_snd_c2).cal_s.
3    (snd_s_rec_c1 + snd_s_rec_c2).Do_s;
4  proc WSCaller_c1_init = Do_c1;
5  proc Do_c1 = rs_c1.rec_c1_snd_s.snd_c1_rec_s.nt_c1.Do_c1;
6  proc WSCaller_c2_init = Do_c2;
7  proc Do_c2 = rs_c2.rec_c2_snd_s.snd_c2_rec_s.nt_c2.Do_c2;
8  init
9  hide ({cal_s,synch_snd_c1_rec_s,synch_snd_c2_rec_s,
10   synch_snd_s_rec_c1,synch_snd_s_rec_c2},
11   rename ({nt_c2->nt2_ws,rs_c2->rs2_ws,
12     nt_c1->nt1_ws,rs_c1->rs1_ws},
13     allow ({nt_c1,nt_c2,rs_c1,rs_c2,
14       synch_snd_c1_rec_s,synch_snd_c2_rec_s,
15       synch_snd_s_rec_c1,synch_snd_s_rec_c2},
16       comm ({snd_c2_rec_s|rec_s_snd_c2->synch_snd_c2_rec_s,
17         snd_c1_rec_s|rec_s_snd_c1->synch_snd_c1_rec_s,
18         snd_s_rec_c2|rec_c2_snd_s->synch_snd_s_rec_c2,
19         snd_s_rec_c1|rec_c1_snd_s->synch_snd_s_rec_c1},
20       WSCaller_c2_init||WSCaller_c1_init||WService_s_init
21 )));
```

Listing 6: Translation of example $WSPattern$ configuration to mCRL2.

A process expression may include sequence composition, alternative choice, conditionals, actions, process calls and ports. The translation of the first three results in the same operation applied to the translated operands. Each action and process is given a unique identifier by combining its name with the variable's one, *e.g.*, process Do becomes Do_s (see line 1 of Listing 6).

The translation of a port depends on its type and the attachments to which it belongs. For each attachment, an action is defined using the variable and port identifiers of the original port reference. For instance, the two attachments of port rec give rise to actions $rec\_s\_snd\_c1$ and $rec\_s\_snd\_c2$. The generated actions are combined into a process expression that represents the expected behaviour according to the port's type. This can be illustrated varying the type of rec to be *and*, *or* and *xor*. The resulting expressions are shown in lines 1, 2 and 3 of Listing 7. The direction of the port influences the resulting process expression when the ports involved have parameters and there is a flow of data.

```
1 rec_s_snd_c1|rec_s_snd_c2
2 rec_s_snd_c1+rec_s_snd_c2+rec_s_snd_c1|rec_s_snd_c2
3 rec_s_snd_c1+rec_s_snd_c2
```

Listing 7: Example process expressions for ports.

The translation of a pattern instance is summarised in expression (25) which represents the parallel composition of the processes resulting from translating instances referenced by the pattern inner variables $vars$. Listing 6 shows the parallel composition in line 20. The communication among such processes is established by an operator Γ (`comm`) according to a set C of communication rules calculated from the attachments. Each communication rule references two actions of processes that synchronise, and a third one resulting from the synchronisation. The corresponding rules for our example are shown between lines 16 and 19. Then, operator ∇ (`allow`) allows a set A of synchronisations and actions (not ports) and blocks all the others, preventing in this way the individual activation of a port (see lines 13 and 15 in our example translation). Operator ρ (`rename`) renames actions according to a set R calculated from the renamings of the pattern instance (see lines 11 to 12). The last operator, τ (`hide`), turns actions in set H invisible (between lines 9-10). Finally, set H is calculated from the actions that occur in the processes conforming the pattern instance, excluding port interactions.

$$\tau_H(\rho_R(\nabla_A(\Gamma_C(\prod_{v\,\in\,vars} \mathcal{T}(v))))) \tag{25}$$

4.4. Architectural analysis

Architectural models in Archery can be compared through the behavioural equivalences and preorders defined in mCRL2. Actually, rooted branching bisimilarity, $\approx_{RB}$, provides a basis for establishing architectural interchangeability with respect to the interface behaviour. Branching bisimilarity [12] relates behaviours differing in the amount of internal activity but exhibiting similar branching structure. Rooted branching bisimilarity adds a rootedness condition: initial internal transitions are never inert. Formally, architectural equivalence and refinement are defined as follows

$$a \equiv b \Leftrightarrow \mathcal{T}(a) \approx_{RB} \mathcal{T}(b) \;\text{ and }\; a \sqsubseteq b \Leftrightarrow \mathcal{T}(a) \sqsubseteq_{RB} \mathcal{T}(b) \tag{26}$$

Coarser relations are sometimes necessary to compare Archery models. Weak trace equivalence and refinement, which abstract from the internal branching structure, can also be used to define coarser architectural relations.

5. Translating CoordL to Archery and back

5.1. From CoordL to Archery

CoordL models can be represented in Archery as instances of specific elements of a pattern. For this let us define a translation $\mathcal{A}(\cdot)$ to represent a CoordL model as an Archery specification. The result is an instance of pattern $WSPattern$, first defined in section 2 (see Listing 1), extended with elements standing for each of CoordL main combinators. To illustrate the translation process we concentrate on coordination pattern P_1, given in expression 8 and depicetd in Figure 3), to illustrate the translation.

5.1.1. Base node

Base nodes represent interactions. In this work we focus on synchronous web service calls. Then, a base node (2) is represented in Archery as an instance of element $WSCaller$, which stands for a web service synchronous caller (see alternatively (17) or (21)). Translation $\mathcal{A}(\cdot)$ for a node n is defined in (27) in terms of script $node$, previously specified in (24). As an example, the translation of the first base node in P_1 is $\mathcal{A}(3) = node(3)$.

$$\mathcal{A}(n) = \; node(n) \rhd 1 = node(n) \tag{27}$$

5.1.2. Juxtaposition

The juxtaposition of two CoordL patterns, formally defined in (3), is the architectural product of the translation of each of them. In the example, translating the juxtaposition of the two first base nodes yields expression $\mathcal{A}(3 \otimes 4) = \ node(3) \boxtimes node(4)$.

$$\mathcal{A}(p \otimes q) = \mathcal{A}(p) \boxtimes \mathcal{A}(q) \tag{28}$$

5.1.3. Link

The representation of a CoordL link (4) in Archery depends of whether it is a loop or not. The latter case resorts attachments, but for the former, a different behaviour for the instance that represents the web service caller must be specified. Element *RLink* stands for such a reflexive link, that includes the possibility of a calling loop.

$$\begin{aligned} R = {} & e(RLink) \diamond \{rs, rec\} \blacklozenge \{se, snd\} \\ & \ast\, mprc(Do, \{rs, nt\}, \text{``}rs.Loop.nt.Do\text{''}) \\ & \circledast\, prc(Loop, \{rec, snd\}, \text{``}snd.rec.(\tau + \tau.Loop)\text{''}) \end{aligned}$$

A link between nodes i and j in pattern p is translated as the application of reconfiguration $link(i,j)$ to the translation of p. Then, the specific reconfiguration depends on whether the link is a self-reference or not (30). In the former case, the value of variable id_i is replaced by an instance of element *RLink*. In the latter the corresponding attachment is made.

$$\mathcal{A}((p) \leftarrow\!\urcorner_i^j) = link(i,j) \rhd \mathcal{A}(p) \tag{29}$$

$$link(i,j)(x) \triangleq \begin{cases} x \boxdot_{id_i} einst(RLink) & \text{if } i = j \\ x \boxdiv ((id_i, nt), (id_j, rs)) & \text{if } i \neq j \end{cases} \tag{30}$$

The translation of a link among a list of nodes L and node j is the successive application of *link* to the translation of p as follows.

$$\mathcal{A}((p) \leftarrow\!\urcorner_L^j) = link(L,j) \rhd \mathcal{A}(p) = \ link(i,j) \underset{i \in L}{\rhd} \mathcal{A}(p) \tag{31}$$

The first two links of our example are then translated as follows.

$$\mathcal{A}((3 \otimes 4) \leftarrow\!\urcorner_{\{3,4\}}^4) = link(\{3,4\},4) \rhd \mathcal{A}(3 \otimes 4)$$

5.1.4. Fork

The representation of fork nodes (5) requires an element $Fork$ in the pattern $WSPattern$. Such an element defines instances with a *in* port rs for receiving start notifications, and two *out* ports (ssa and ssb) to send start notifications to the two instances.

$$\begin{aligned} F = {} & e(Fork) \diamond \{rs\} \blacklozenge \{ssa, ssb\} \\ & \circledast\, mprc(Do, \{rs, ssa, ssb\}, \text{“}rs.(ssa.ssb + ssb.ssa).Do\text{”}) \end{aligned}$$

We translate a fork node f that starts threads in nodes a and b as an instance of element $Fork$. The corresponding external visible ports of a and b are removed. Subsequently the *out* port of f is attached to the respective rs ports in nodes a and b.

$$\mathcal{A}(f \prec\!\!\!\lhd\,{}^{a}_{b}\,(p)) = fork(f, a, b) \rhd \mathcal{A}(p) \tag{32}$$

$$\begin{aligned} fork(f, a, b)(x) \triangleq {} & x \boxed{\square} ((id_f, Fork) \boxed{\circ}\, einst(Fork)) \\ & \boxed{\not\vartriangle} \{rs_a, rs_b\} \boxed{\vartriangle} ((id_f, rs), rs_f) \\ & \boxed{\div} \{((id_f, ss_a), (id_a, rs)), ((id_f, ss_b), (id_b, rs))\} \end{aligned} \tag{33}$$

The fork in the example CoordL pattern is translated as $\mathcal{A}(f_2 \prec\!\!\!\lhd\,{}^{3}_{5}\, Q) = fork(f_2, 3, 5) \rhd \mathcal{A}(Q)$ where Q stands for the rest of the expression.

5.1.5. Join

In a similar way the instances of element $Join$ of pattern $WSPattern$ represent join nodes (6). Instances of such an element have two *in* ports to receive termination notifications from either node a or node b and an *out* port to notify its termination.

$$\begin{aligned} J = {} & e(Join) \diamond \{rta, rtb\} \blacklozenge \{nt\} \\ & \circledast\, mprc(Do, \{rs, rta, rtb\}, \text{“}(rta.rtb + rtb.rta).nt.Do\text{”}) \end{aligned}$$

Then, a join between nodes a and b to node j in pattern p is translated as the application of reconfiguration $join(a, b, j)$ to the architecture resulting from translating p. The script includes an instance of element $Join$ into the architecture, removes renamings of ports which notify termination of instances representing a and b, and reconnect such ports to the corresponding ones in the $Join$ instance.

$$\begin{aligned}
\mathcal{A}((p) \,{}^{a}_{b}\!\rightarrowtail j) &= join(j,a,b) \rhd \mathcal{A}(p) && (34)\\
join(j,a,b)(x) &\triangleq x \boxed{\square} ((id_j, Join) \boxed{\circ}\, einst(Join)) && (35)\\
&\qquad \boxed{\not\triangle}\, \{nt_a, nt_b\} \boxed{\triangle} ((id_j, nt), nt_j)\\
&\qquad \boxed{\div}\, \{((id_a, nt), (id_j, rta)), ((id_b, nt), (id_j, rtb))\}
\end{aligned}$$

The translation of a join node among lists A and B requires iterating the sets to remove external ports and to create appropriate attachments.

$$\begin{aligned}
\mathcal{A}((p) \,{}^{A}_{B}\!\rightarrowtail j) &= join(j,A,B) \rhd \mathcal{A}(p) && (36)\\
join(j,A,B)(x) &\triangleq x \boxed{\square} ((id_j, Join) \boxed{\circ}\, einst(Join)) \boxed{\triangle} ((id_j, nt), nt_j) && (37)\\
&\qquad \underset{i \in A \cup B}{\boxed{\not\triangle}}\, nt_i \underset{a \in A}{\boxed{\div}} ((id_a, nt), (id_j, rta)) \underset{b \in B}{\boxed{\div}} ((id_b, nt), (id_j, rtb))
\end{aligned}$$

We are now ready to completely translate the coordination pattern fragment P_1 used as an example. Thus,

$$\begin{aligned}
\mathcal{A}(P_1) = {} & fork(f_2, 3, 5) \rhd join(j_2, \{3,4\}, \{5,6\}) \rhd && (38)\\
& (\ link(\{3,4\}, 4) \rhd node(3) \boxtimes node(4)\ )\\
& \boxtimes (\ link(\{5,6\}, 6) \rhd node(5) \boxtimes node(6)\ )
\end{aligned}$$

5.1.6. Thread trigger

CoordL thread trigger (7) nodes are represented by instances of $TTrigger$ shown below. It provides *in* ports rta and rtb to receive notifications of termination from nodes a and b and *out* ports ssc and ssd to send start signals to nodes c and d.

$$\begin{aligned}
T = {} & e(TTrigger) \diamond \{rta, rtb\} \blacklozenge \{ssc, ssd\}\\
& \circledast\, mprc(Do, \{rta, rtb, ssc, ssd\},\\
& \text{“}(rta.rtb + rtb.rta).ssc.Do + rta.ssd.Do + rtb.ssd.Do\text{”})
\end{aligned}$$

A thread trigger on p connecting nodes a, b, c and d is translated by applying reconfiguration $ttrigger(a,b,c,d)$ to the architecture resulting from translating p. The script creates an instance of $TTrigger$ and adds it into the architecture. Subsequently removes renamings of ports which notify termination of instances representing a and b, and ports receiving start signals of instances representing c and d. Then, it connects the corresponding ports to the ones in the $TTrigger$ instance.

$$\mathcal{A}((p)\,{}^{a}_{b}\!\rightarrowtail\!\!\!\prec{}^{c}_{d}) = ttrigger(a,b,c,d) \rhd \mathcal{A}(p) \tag{39}$$

$$\begin{aligned} ttrigger(a,b,c,d)(x) \triangleq\ & (x \boxtimes node(c) \boxtimes node(d)) \boxed{\not\triangleleft} \{nt_a, nt_b, rs_c, rs_d\} \\ & \boxed{\scriptstyle\square}\ ((id_t, TTrigger) \boxed{\scriptstyle\circ}\ einst(TTrigger)) \\ & \boxed{\div}\ \{((id_a, nt), (id_t, rta)), ((id_b, nt), (id_t, rtb)), \\ & ((id_t, ssc), (id_c, rs)), ((id_t, ssd), (id_d, rs))\} \end{aligned} \tag{40}$$

In the case that lists A and B of nodes are given as arguments to the thread trigger node, the reconfiguration requires iterating on such lists to remove renamings and create attachments accordingly.

$$\mathcal{A}((p)\,{}^{A}_{B}\!\rightarrowtail\!\!\!\prec{}^{c}_{d}) = ttrigger(A,B,c,d) \rhd \mathcal{A}(p) \tag{41}$$

$$\begin{aligned} ttrigger(A,B,c,d)(x) \triangleq\ & (x \boxtimes node(c) \boxtimes node(d)) \\ & \boxed{\scriptstyle\square}\ ((id_t, TTrigger) \boxed{\scriptstyle\circ}\ einst(TTrigger)) \\ & \underset{a\in A}{\boxed{\not\triangleleft}}\ nt_a \underset{b\in B}{\boxed{\not\triangleleft}}\ nt_b \boxed{\not\triangleleft} \{rs_c, rs_d\} \\ & \underset{a\in A}{\boxed{\div}}\ ((id_a, nt), (id_t, rta)) \underset{b\in B}{\boxed{\div}}\ ((id_b, nt), (id_t, rtb)) \\ & \boxed{\div}\ \{((id_t, ssc), (id_c, rs)), ((id_t, ssd), (id_d, rs))\} \end{aligned} \tag{42}$$

5.2. From Archery to CoordL

An Archery specification representing a coordination pattern can be translated back to CoordL. The model is assumed to be an instance of the architectural pattern $WSPattern$and to be structured in terms of architectural products and the application of the following scripts: $node$ (24), $link$ (30), $fork$ (33), $join$ (35), and $ttrigger$(40)). Translation $\mathcal{C}(\cdot)$ receives such a specification S and returns a CoordL model. It is defined inductively as follows,

$$\begin{aligned} \mathcal{C}(node(a)) &= \langle \{a\}, \{a\}, \{a\}, \emptyset \rangle \\ \mathcal{C}(S_1 \boxtimes S_2) &= \mathcal{C}(S_1) \otimes\ \mathcal{C}(S_2) \\ \mathcal{C}(link(a,b) \rhd S) &= \mathcal{C}(S) \Lsh^{b}_{a} \\ \mathcal{C}(fork(f,a,b) \rhd S) &= f \prec^{a}_{b} \mathcal{C}(S) \\ \mathcal{C}(join(j,a,b) \rhd S) &= \mathcal{C}(S)\ {}^{a}_{b}\!\rightarrowtail j \\ \mathcal{C}(ttrigger(a,b,c,d) \rhd S) &= (\mathcal{C}(S))\,{}^{a}_{b}\!\rightarrowtail\!\!\!\prec{}^{c}_{d} \end{aligned}$$

The translation for scripts with set or list arguments, namely for variants of *link*, *join* and *ttriger*, becomes

$$\begin{aligned}
\mathcal{C}(link(L,b) \rhd S) &= \mathcal{C}(S) \mathrel{\reflectbox{$\lrcorner$}}{}^{b}_{L} \\
\mathcal{C}(join(j,A,B) \rhd S) &= \mathcal{C}(S) {}^{A}_{B} \rightarrowtail j \\
\mathcal{C}(ttrigger(a,b,c,d) \rhd S) &= (\mathcal{C}(S)) {}^{a}_{b} \rightarrowtail\!\!\!\leftarrowtail {}^{c}_{d}
\end{aligned}$$

Applying the translation to the result of expression (38), *i.e.*, $\mathcal{C}(\mathcal{A}(P_1))$, yields a CoordL model which is equivalent, modulo commutativity/associativity and thread renaming, to P_1.

6. Architectural re-engineering at work

The re-engineering process starts by inspecting the code of the legacy system under consideration, using CoordPat to extract the original system coordination layer. As discussed in section 2, the coordination pattern shown in Figure 2 describes the coordination logic of the module responsible for updating user profiles in components CRM, ERP and TS. Two undesired behaviours and an improvement opportunity were detected upon an informal analysis:

- *Duplicate user creation* – The service issues a user existence check and, in case of a negative answer, calls repeatedly the user creation operation until it answers. This loop copes with the possibility of a creation failure, but does not distinguish this from the case in which although the user is effectively created, but the response to the corresponding service fails. If this happens, duplicated users are inserted into the system. The problem can be avoided by inserting user-existence checks before every call to the creation operation.

- *Parallel user updates* – Upon checking and creating users, the service updates them by sequentially calling the corresponding operations in the CRM, ERP and TS components. Since there is no dependence among these operations and the call order is arbitrary, it is possible to call them in parallel.

- *Deadlocks* – If one of the create or update operations is off-line the service will loop indefinitely. Also this can be easily fixed by suitably limiting the number of retries to a natural number.

Once the coordination model of the original system is extracted, the next phase translates it to Archery. This produces a sequence of models: at each step the new model is compared with the previous one to detect whether they are (behaviourally) equivalent, or one is a refinement of the other. Such relationships can be automatically established in Archery using the underlying mCRL2 tools. Note, however, that as one might have expected, along a process aiming at improving the functionality of an architecture most design steps lead to non bisimilar models. The situation is completely different from a specification refinement process. Finally, once a satisfactory model has been obtained, it is translated back to CoordL to guide a re-implementation phase.

The rest of this section is organised as follows: subsection 6.1 provides a translation of our example CoordL model to an Archery specification. Such model is then modified by identified common patterns (subsection 6.2), to include user-existence checks (subsection 6.3), to replace sequential calls to updatings by parallel ones (subsection 6.4), and, finally, to avoid deadlocks (subsection 6.5). The reverse translation from Archery to CoordL is shown in subsection 6.6.

6.1. From CoordL to Archery

The translation from CoordPat to Archery is a direct application of $\mathcal{A}(\cdot)$ (introduced in section 5.1). Note that the fragment P_1 of the example was already translated (see expression (38)). Therefore, we end up with

$$\begin{aligned}\mathcal{A}(P) = \mathcal{A}_0 = {}& link(j_2, 7) \rhd (\ fork(f_1, f_2, 1) \rhd join(j_1, \{j_2\}, \{1, 2\}) \rhd \\ & \quad (\ link(\{1, 2\}, 2) \rhd node(1) \boxtimes node(2)\ ) \\ & \quad \boxtimes (\ fork(\{f_2\}, 3, 5) \rhd join(j_2, \{3, 4\}, \{5, 6\}) \rhd \\ & \quad\quad (\ link(\{3, 4\}, 4) \rhd node(3) \boxtimes node(4)\ ) \boxtimes \\ & \quad\quad (\ link(\{5, 6\}, 6) \rhd node(5) \boxtimes node(6)\ )\ )\ ) \\ & \boxtimes (\ link(7, 7) \rhd link(\{7, 8\}, 8) \rhd link(\{8, 9\}, 9) \rhd \\ & \quad node(7) \boxtimes node(8) \boxtimes node(9)\ )\end{aligned}$$

6.2. Detecting patterns

The re-engineering process starts once an Archery specification is obtained. In the case study discussed in this paper, the architect's attention was first driven towards detecting common, repeating patterns. Let us describe in some detail the steps taken in this phase.

First of all a behavioural pattern was detected in the interaction of three pair of instances, and each pair was factored out into a single one. Specifically, it was observed that the configuration among instances generated by nodes 1 and 2, 3 and 4, and 5 and 6, can be generalised taking the pair of nodes a and b as parameters into a reconfiguration $link(\{a,b\},b) \rhd node(a) \boxtimes node(b)$. Moreover, such configuration can be replaced by an instance of a new element, *ReadLoopCreate*, shown in expression (43). Then, the reconfiguration script in expression (44) is defined to instantiate it. Note that $rlc(a,b) \rhd 1 \equiv link(\{a,b\},b) \rhd node(a) \boxtimes node(b) \rhd 1$, *i.e.*, they are interchangeable. This equivalence holds provided that roots activate configurations only once. Node identifier $c(a,b)$ allows us to trace original node identifiers, and thus to recover the CoordL model.

$$\begin{aligned}
RLC = {} & e(ReadLoopCreate) \diamond \{rs, rec_r, rec_c\} \blacklozenge \{nt, snd_r, snd_c\} && (43)\\
& \ast\, mprc(Do, \{rs, snd_r, rec_r, nt\}, \\
& \qquad \text{“}rs.snd_r.rec_r.(\tau + \tau.Loop).nt\text{”}) \\
& \circledast\, prc(Loop, \{rec_c, snd_c\}, \text{“}snd_c.rec_c.(\tau + \tau.Loop)\text{”}) \\
rlc(a,b)(x) \triangleq {} & x \boxed{\square} (&& (44)\\
& \quad (id_{c(a,b)}, ReadLoopCreate) \boxed{\circ}\, einst(ReadLoopCreate)) \\
& \boxed{\triangle} \{((id_{c(a,b)}, rs), rs_{c(a,b)}), ((id_{c(a,b)}, nt), nt_{c(a,b)}), \\
& ((id_{c(a,b)}, snd_r), snd_a), ((id_{c(a,b)}, rec_r), rec_a), \\
& ((id_{c(a,b)}, snd_c), snd_b), ((id_{c(a,b)}, rec_c), rec_b)\}
\end{aligned}$$

The nodes that invoke update operations, namely 7,8 and 9, follow a specific pattern as well. In this case, it is enough to define a script that captures such a pattern as it is done in (45). Then, the example configuration is reformulated as in (46), It can be shown that $\mathcal{A}_0 \equiv \mathcal{A}_1$.

$$\begin{aligned}
upd(a)(x) \triangleq {} & link(a,a) \rhd node(a) && (45)\\
\mathcal{A}_1 = {} & link(j_2, 7) \rhd && (46)\\
& (\, fork(f_1, f_2, 1) \rhd join(j_1, \{j_2\}, c(1,2)) \rhd rlc(1,2) \\
& \quad \boxtimes (\, fork(f_2, c(3,4), c(5,6)) \\
& \qquad \rhd join(j_2, c(3,4), c(5,6)) \rhd rlc(3,4) \boxtimes rlc(5,6)\ )\ ) \\
& \boxtimes (\ link(7,8) \rhd link(8,9) \rhd upd(7) \boxtimes upd(8) \boxtimes upd(9)\ )
\end{aligned}$$

6.3. Introducing user-existence checks

Duplicated users may be created by $\mathcal{A}_1$. According to the available information in the original CoordL pattern (see Figure 2), nodes 1, 3, and 5 are read operations performed on each subsystem to ensure that the user does not exist, before the respective nodes 2, 4, and 6 that actually create it. Note that if a create operation succeeds, but the response does not arrive to the coordinator service, a duplicated user is shown. In $\mathcal{A}_1$ this isisolated in element *ReadLoopCreate*. We address it by defining element *ReadCreateLoop* as indicated in (47) which forces the loop to include the user-existence check on every call. The reconfiguration script $rcl(a,b)$ that creates instances of this element is very similar to $rlc(a,b)$, but for the instantiated element. As expected, the two configurations are not equivalent, *i.e.*, $rcl(a,b) \rhd 1 \not\equiv rlc(a,b) \rhd 1$. However, there is an equivalence $rcl(a,b) \rhd 1 \equiv link(a,b) \rhd link(b,a) \rhd node(a) \boxtimes node(b)$, which indicates how to express the new element in terms of reconfiguration scripts used by $\mathcal{A}(\cdot)$, which helps in translating the resulting specification back to CoordL. We obtain a configuration $\mathcal{A}_2$ (by replacing script rlc with rcl) in which every call to a create is preceded by a call to a read, and such that $\mathcal{A}_1 \not\equiv \mathcal{A}_2$.

$$\begin{aligned} RCL = {} & e(ReadCreateLoop) \diamond \{rs, rec_r, rec_c\} \blacklozenge \{nt, snd_r, snd_c\} \\ & \circledast\, mprc(Do, \{rs, nt\}, \text{“}rs.Loop.nt.Do\text{”}) \\ & \circledast\, prc(Loop, \{rec_r, snd_r, rec_c, snd_c\}, \\ & \text{“}snd_r.rec_r.(\tau + \tau.snd_c.rec_c + snd_c.rec_c.Loop)\text{”}) \end{aligned} \tag{47}$$

6.4. Putting user-update operations in parallel

The subsequent re-engineering step modifies the coordinator by placing user-updates, *i.e.*, instances representing nodes 7, 8, 9, in parallel. For this, we define in (48) a new configuration $\mathcal{A}_3$, that differs from $\mathcal{A}_2$ in a number of forks and joins allowing the concurrent execution of these instances. As expected $\mathcal{A}_2 \not\equiv \mathcal{A}_3$.

$$\begin{aligned} \mathcal{A}_3 = {} & link(j_1, f_3) \rhd (\ fork(f_1, f_2, 1) \rhd join(j_1, j_2, c(1,2)) \rhd rcl(1,2) \\ & \quad \boxtimes (\ fork(f_2, c(3,4), c(5,6)) \rhd join(j_2, c(3,4), c(5,6)) \rhd \\ & \quad\quad rcl(3,4) \boxtimes rcl(5,6)\ )\ ) \\ & \boxtimes (\ fork(f_3, 7, f_4) \rhd join(j_3, 7, j_4) \rhd upd(7) \\ & \quad \boxtimes (\ fork(f_4, 8, 9) \rhd join(j_4, 8, 9) \rhd upd(8) \boxtimes upd(9)\ )\ ) \end{aligned} \tag{48}$$

6.5. Avoiding deadlocks

At this stage, a potential for deadlock still remains. Actually, when an update or create operation does not respond, the call is repeated until a response is obtained. As a consequence, if one of the operations is off-line, the integrated user-update neither fails nor succeeds. The problem is solved by adding elements that have a counter and an iteration limit, and replacing in a new configuration $\mathcal{A}_4$ the corresponding instances. Element *ReadCreateFiniteLoop* in (49) replaces *ReadCreateLoop* and element *UpdateFiniteLoop* in (50) is used instead of *RLink* for updates. We need to define new configuration scripts for these elements. Script $rcfl$ in (51) replaces rcl, only differing in the creation of the variable and the instance, which now receives an integer N as a parameter. The set R of renamings remains unaltered. In a similar way, script ufl in (52) replaces upd, which has a constructor that receives an integer as parameter as well. We observe that the configurations obtained by the respective scripts are not equivalent, *i.e.*, $rcfl(a,b) \rhd 1 \not\equiv rcl(a,b)$ and $ufl(a) \rhd 1 \not\equiv upd(a) \rhd 1$. Note that the new elements just introduced convey information that cannot be translated to CoordL.

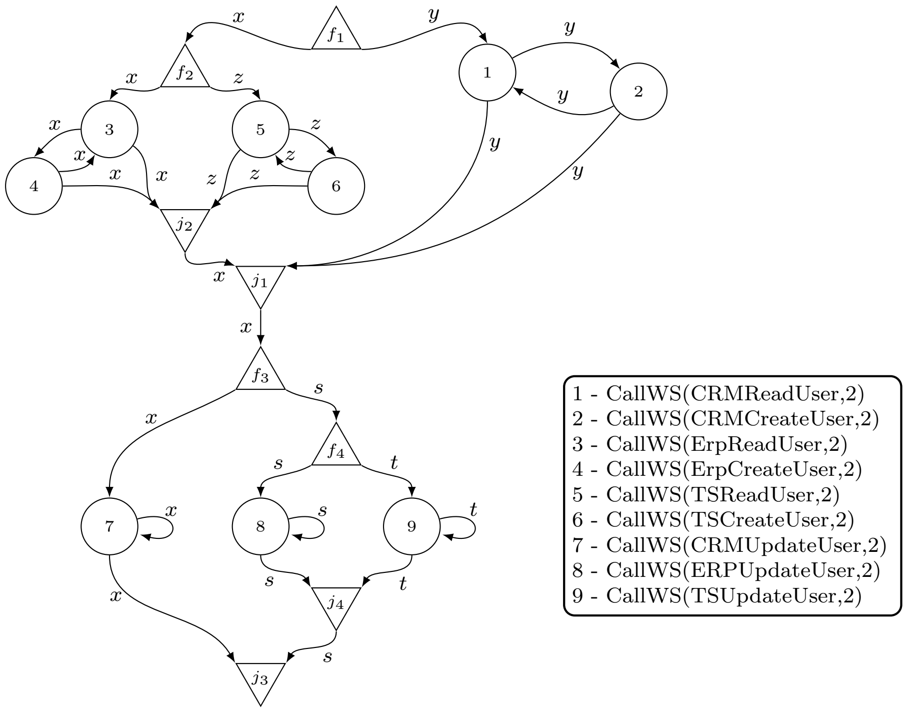
$$\begin{aligned}
RCFL = {} & e(\mathit{ReadCreateFiniteLoop}, [\langle \mathit{max}, \mathit{Int}\rangle]) && (49)\\
& \diamond \{rs, rec_r, rec_c\} \blacklozenge \{nt, snd_r, snd_c\}\\
& \ast\ \mathit{mprc}(\mathit{Do}, [\langle n, \mathit{Int}, \mathit{max}\rangle], \{rs, nt\},\\
& \qquad \text{“}\mathit{rs.Loop}(0,n).\mathit{nt.Do}(n)\text{”})\\
& \circledast\ \mathit{prc}(\mathit{Loop}, [\langle i, \mathit{Int}\rangle, \langle n, \mathit{Int}\rangle], \{rec_r, snd_r, rec_c, snd_c\},\\
& \qquad \text{“}\mathit{snd_r.rec_r}.(\tau + \tau.\mathit{snd_c.rec_c}\\
& \qquad + \tau.\mathit{snd_c.rec_c}.(i < n) \mathrel{-\!\!>} \mathit{Loop}(i+1,n) <> \tau)\text{”})\\
UFL = {} & e(\mathit{UpdateFiniteLoop}, [\langle \mathit{max}, \mathit{Int}\rangle]) && (50)\\
& \diamond \{rs, rec\} \blacklozenge \{se, snd\}\\
& \ast\ \mathit{mprc}(\mathit{Do}, [\langle n, \mathit{Int}, \mathit{max}\rangle], \{rs, nt\},\\
& \qquad \text{“}\mathit{rs.Loop}(0,n).\mathit{nt.Do}(n)\text{”})\\
& \circledast\ \mathit{prc}(\mathit{Loop}, [\langle i, \mathit{Int}\rangle, \langle n, \mathit{Int}\rangle], \{rec, snd\},\\
& \qquad \text{“}\mathit{snd.rec}.(\tau + \tau.(i < n) \mathrel{-\!\!>} \mathit{Loop}(i+1,n) <> \tau)\text{”}\\
rcfl(a,b)(x) \triangleq {} & x \boxed{\square} ((id_{c(a,b)}, \mathit{ReadCreateFiniteLoop}) && (51)\\
& \boxed{\circ}\ \mathit{einst}(\mathit{ReadCreateFiniteLoop}, [N])) \boxed{\triangle} R\\
ufl(a)(x) \triangleq {} & x \boxed{\square} ((id_a, \mathit{UpdateFiniteLoop}) && (52)
\end{aligned}$$

$$\boxdot_{id_a}\ einst(UpdateFiniteLoop, [N]))$$

6.6. Translating back to CoordL

Upon obtaining a satisfactory specification, translation $\mathcal{C}(\cdot)$ is applied to obtain the corresponding CoordL model. Given that the model in section 6.5 cannot be expressed in CoordL, we translate $\mathcal{A}_3$ instead, the one obtained in section 6.4, shown in (48). Before applying the translation, we need to replace scripts *rcl* and *upd* by their equivalent forms expressed in terms of scripts *link* and *node*. Translation $\mathcal{C}(\cdot)$ applied to the resulting model yields (53), which is graphically depicted in Figure 6.

$$\begin{aligned}
&(\ (\ f_1 \prec^{f_2}_{1}\ 1 \otimes 2 \leftarrow\!\urcorner^{2}_{1} \leftarrow\!\urcorner^{1}_{2} \\
&\quad \otimes (\ f_2 \prec^{3}_{5}\ (5 \otimes 6) \leftarrow\!\urcorner^{6}_{5} \leftarrow\!\urcorner^{5}_{6} \otimes (3 \otimes 4) \leftarrow\!\urcorner^{4}_{3} \leftarrow\!\urcorner^{3}_{4}\ {}^{3,4}_{5,6}\!\succ j_2\ )\ {}^{j_2}_{1,2}\!\succ j_1\ ) \\
&\otimes (\ f_3 \prec^{7}_{f_4}\ 7 \leftarrow\!\urcorner^{7}_{7} \otimes (\ f_4 \prec^{8}_{9}\ 8 \leftarrow\!\urcorner^{8}_{8} \otimes 9 \leftarrow\!\urcorner^{9}_{9}\ {}^{8}_{9}\!\succ j_4\ )^{7}_{j_4} \succ j_3)\ )\leftarrow\!\urcorner^{f_3}_{j_1}
\end{aligned} \tag{53}$$



f1
f2
f3
f4
j1
j2
j3
j4
x
y
z
s
t
1 - CallWS(CRMReadUser,2)
2 - CallWS(CRMCreateUser,2)
3 - CallWS(ErpReadUser,2)
4 - CallWS(ErpCreateUser,2)
5 - CallWS(TSReadUser,2)
6 - CallWS(TSCreateUser,2)
7 - CallWS(CRMUpdateUser,2)
8 - CallWS(ERPUpdateUser,2)
9 - CallWS(TSUpdateUser,2)

Figure 6: Coordination pattern for the re-engineered integrated update.

7. Related work and conclusions

This paper introduced an approach to the architectural re-engineering of legacy software systems, ranging from the reconstruction of coordination pattens from source code to the specification, analysis and modification of the corresponding architectural model at a higher level of abstraction, as well as its mapping back to the implementation layer. Conceptually, it focuses on the architecture's structure of interactions which is often strongly weaved with the application at the source code level. Methodologically, the approach combines an extraction and analysis stage with an iterative re-engineering process at a higher level of abstraction. Both stages are tool-supported, by CoordPat and Archery, respectively. The former is an extension to CoordInspector [1] to which adds a notion of coordination pattern, a language and a calculus of such patterns, and a facility for pattern search and identification over a program dependency graph. The latter is a high level architectural description language which resorts to mCRL2 for behaviour simulation and analysis.

7.1. Related work

There is a number of tools and methodologies targeting the identification of architectural elements from source code or intermediate code representations based on programs analysis techniques (see, e.g. [18] for an early reference). CoordPat shares a number of intuitions discussed in [19, 20, 21]. CoordPat search, however, is driven by coordination patterns, parametric on the communication primitives used in the source programming platform (the *glue* primitives), embodying complex interaction and architectural models. To the best of our knowledge, this has not been made before. The CoorL language, however, was initially inspired by operational notations for the description of Reo circuits [22], namely in [23] and [24].

Archery is an *architectural description language* (ADL). ADLs model software architectures in terms of components and connectors arranged according to their interfaces in configurations. The interaction points of components and connectors are called ports and roles, respectively. Reference [25], identifies these abstractions as essential for an ADL, and stresses the importance of providing tool-support to the development and evolution of architectural models. Notations that do not provide first-class constructors for these abstractions, such as UML [26], cannot be considered as architectural languages. Actually, UML was conceived to provide a unique syntax and modelling framework for object oriented software development. Although it can be used to model software architectures, as is shown in [27], a proper ADL-like extension of object oriented constructors to represent architectural abstractions is required [25].

ACME [28], ADR [29], Darwin [30, 31, 32], and Wright [33, 34] are among the languages that provide the essential abstractions. While ACME is focussed in the structural dimension of architectural specifications, the others address, in different ways, the representation and analysis of architectures able to reconfigure themselves at run-time [30, 34, 27]. All of them provide constructors to define types of architectural elements (components and connectors), with associated interfaces defined in terms of a broad notion of interaction points. However, two main approaches can be distinguished. In one of them different constructors are used to deal with component and connector abstractions separately [28, 33, 34]. In the other there is a single constructor to manage them uniformly [29, 35, 32]. The Archery language follows the latter approach providing a single constructor to define architectural element types. All of these languages provide a constructor to build configurations out of instances of architectural element types previously defined.

Tool supported development and analysis of architectural models, and their evolution, entail the need for a formal, underlying semantics. Reference [36] provides an extensive discussion of this issue and proposes a classification of ADLs based on the style of semantics adopted. Two groups emerge as particularly important: process algebra and graph-based approaches. While Darwin [30] and Wright [34] are examples of the former, ADR [29] combines both approach. Reference [16], in particular, presents a way of interpreting process algebra descriptions as graphs and an algebra of for (ADR-like) graphs made of graph composition operators and a sound of complete axiomatization of graph isomorphism. Similarly to Darwin and Wright, Archery [3] models the behavioural dimension in software architectures with process algebras. It also exploits the higher order, equational data types provided by mCRL2 by allowing the specification of data-typed interactions and data-state of architectural element instances.

Although the formal semantics of ACME is debatable (ACME places itself at a meta-level for interchanging different types of architectural abstractions), the language gained recognition as the least common denominator for architectural design [28, 27]. Actually, it adds to the essential abstractions a *representation* to model hierarchical composition and *representation maps* which map internal interaction points of a configuration to its external interface. Archery represents these abstractions with a constructor that indicates that an architectural element instance has an internal architecture whose interaction points are mapped to the externally visible ones.

ADLs that support the concept of architectural pattern [37] or style [38] facilitate the development of specifications because they are able to abstract recurring forms. Reference [39] gives a general characterisation as a description of element and configuration types, and a set of constraints their use. Unfortunately, the notion is often used without a proper formalisation.

Patterns can be enriched by the specification of architectural constraints in a suitable logic. The latter can be enforced either *by construction*, or either *by restriction* [40]. ADR [29] uses the former mechanism leaving constraints implicit. The latter approach requires the explicit specification of constraints that forbid generic (re)configuration operations leading to incorrect configurations. Darwin [31] enforces constraints by restriction with a translation of the structural dimension of architectures to Alloy [41]. Archery follows a similar approach. A language extension for the specification of architectural constraints in modal logics is under development. How those can be mapped back to constraints at the level of coordination patterns as the ones specified in CoordL, remains a topic for future research.

7.2. Contributions and future work

The main contribution of this paper was the re-engineering approach itself, and a systematic translation scheme from code-level models (as generated by CoordPat) to conceptual models (as represented in Archery) and back. The CoordPat tool and a formal semantic model for its pattern description language (CoordL) was also introduced here.

The combined use of these two methodologies/tools provides the working software architect with an interesting, yet simple, framework for architectural re-engineering. Further case studies are being made to provide extra empirical evidence of its usefulness and identify possible improvements.

But, of course, a number of questions remain to be answered. A main topic concerns the automatisation of the whole bi-directional translation process from CoordL to Archery and back, and the integration of the whole approach in the Eclipse platform for easier deployment. Another one concerns the enrichment of both CoordL coordination patterns and Archery architectural patterns with *quantitative* annotations, for example to measure communication throughput or, in general, QoS levels associated to architectures, and to propagate such information along composition. More and more a system behaviours of interest are broader than the traditional Boolean "correct" or "incorrect" judgment. Quantitative aspects include, among others, timing (discrete, continuous or hybrid), probability of success or failure including cost and reward, and quantified information flows in a software architecture. Addressing such aspects along the lines of the approach discussed in this paper constitutes a main challenge to our current work.

References

[1] N. F. Rodrigues, L. S. Barbosa, CoordInspector: a tool for extracting coordination data from legacy code, in: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China, IEEE Computer Society, 2008, pp. 265–266.

[2] N. F. Rodrigues, N. Oliveira, L. S. Barbosa, The role of coordination in software integration projects, in: R. Meersman, T. Dillon, P. Herrero (Eds.), OTM 2011 Workshops, volume 7046 of *Lecture Notes in Computer Science*, Springer-Verlag, 2011, pp. 83–92.

[3] A. Sanchez, L. S. Barbosa, D. Riesco, Bigraphical modelling of architectural patterns, in: F. Arbab, P. C. Ölveczky (Eds.), Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers, volume 7253 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 313–330.

[4] L. S. Barbosa, P. R. Henriques, A. Sanchez, Towards rigorous analysis of open source software, in: M. Kyas, S. Meng, V. Stolz (Eds.), Proceedings of TTSS115th Inter. Workshop on Harnessing Theories for Tool Support in Software, Sep. 2011, University of Oslo, RR-409, 2011, pp. 77–89.

[5] L. S. Barbosa, A. Cerone, A. K. Petrenko, S. A. Shaikh, Certification of open-source software: A role for formal methods?, International Journal of Computer Systems Science and Engineering (2010) 273–281.

[6] N. F. Rodrigues, L. S. Barbosa, Slicing for architectural analysis, Sci. Comput. Program. 75 (2010) 828–847.

[7] D. Gelernter, N. Carrier, Coordination languages and their significance, Communications of the ACM 2 (1992) 97–107.

[8] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, in: PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation, ACM Press, 1988, pp. 35–46.

[9] A. Sanchez, L. S. Barbosa, D. Riesco, A language for behavioural modelling of architectural patterns, in: Proceedings of the Third Workshop on Behavioural Modelling, BM-FA '11, ACM, New York, NY, USA, 2011, pp. 17–24.

[10] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, M. van Weerdenburg, The formal specification language mCRL2, in: E. Brinksma, D. Harel, A. Mader, P. Stevens, R. Wieringa (Eds.), Methods for Modelling Software Systems (MMOSS), volume 06351 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[11] R. Milner, Bigraphical reactive systems, in: K. G. Larsen, M. Nielsen (Eds.), CONCUR, volume 2154 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 16–35.

[12] J. C. M. Baeten, T. Basten, M. A. Reniers, Process Algebra: Equational Theories of Communicating Processes, Cambridge University Press, 2010.

[13] J. Baeten, A brief history of process algebra, Theoretical Computer Science 335 (2005) 131–146.

[14] A. Aldini, M. Bernardo, F. Corradini, A Process Algebraic Approach to Software Architecture Design, volume 54, Springer London, London, 2010.

[15] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, T. A. C. Willemse, An overview of the mCRL2 toolset and its recent advances, in: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice

of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, volume 7795 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 199–213.

[16] R. Bruni, A. Lluch-Lafuente, U. Montanari, Style-based architectural reconfigurations, Bulletin of the EATCS 94 (2008) 161–180.

[17] A. Sanchez, A calculus of architectural patterns, Ph.D. thesis, Universidad de San Luis, Argentina, 2014.

[18] K. J. Ottenstein, L. M. Ottenstein, The program dependence graph in a software development environment, in: Proc. of the first ACM SIGSOFT/SIGPLAN software engineering posium on Practical software development environments, ACM Press, 1984, pp. 177–184.

[19] V. P. Ranganath, J. Hatcliff, Slicing concurrent Java programs using Indus and Kaveri, Int. J. Softw. Tools Technol. Transf. 9 (2007) 489–504.

[20] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M. B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Trans. Program. Lang. Syst. 29 (2007).

[21] M. G. Nanda, S. Ramesh, Interprocedural slicing of multithreaded programs with applications to java, ACM Trans. Program. Lang. Syst. 28 (2006) 1088–1144.

[22] F. Arbab, Reo: a channel–based coordination model for component composition, Mathematical Structures in Comp. Sci. 14 (2004) 329–366.

[23] C. Krause, Z. Maraikar, A. Lazovik, F. Arbab, Modeling dynamic reconfigurations in reo using high-level replacement systems, Sci. Comput. Program. 76 (2011) 23–36.

[24] N. Oliveira, L. S. Barbosa, Reconfiguration mechanisms for service coordination, in: M. H. ter Beek, N. Lohmann (Eds.), Web Services and Formal Methods - 9th International Workshop, WS-FM 2012, Tallinn, Estonia, September 6-7, 2012, Revised Selected Papers, Lecture Notes in Computer Science, Springer, 2012, pp. 134–149.

[25] N. Medvidovic, R. Taylor, A classification and comparison framework for software architecture description languages, Software Engineering, IEEE Transactions on 26 (2000) 70–93.

[26] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Boston, MA, 2 edition, 2005.

[27] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins, Modeling software architectures in the unified modeling language, ACM Trans. Softw. Eng. Methodol. 11 (2002) 2–57.

[28] D. Garlan, R. Monroe, D. Wile, ACME: An architecture description interchange language, in: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97, IBM Press, 1997, pp. 169–183.

[29] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto, Style Based Architectural Reconfigurations, Bulletin of the European Association for Theoretical Computer Science (EATCS) 94 (2008) 161–180.

[30] J. Magee, J. Kramer, Dynamic structure in software architectures, in: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '96, ACM, New York, NY, USA, 1996, pp. 3–14.

[31] I. Georgiadis, J. Magee, J. Kramer, Self-organising software architectures for distributed systems, in: Proceedings of the first workshop on Self-healing systems, WOSS '02, ACM, New York, NY, USA, 2002, pp. 33–38.

[32] J. Kramer, J. Magee, S. Uchitel, Software architecture modeling & analysis: A rigorous approach, in: M. Bernardo, P. Inverardi (Eds.), Formal Methods for Software Architectures, volume 2804 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2003, pp. 44–51.

[33] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Trans. Softw. Eng. Methodol. 6 (1997) 213–249.

[34] R. Allen, R. Douence, D. Garlan, Specifying and analyzing dynamic software architectures, in: E. Astesiano (Ed.), Fundamental Approaches

to Software Engineering, volume 1382 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1998, pp. 21–37.

[35] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: W. Schfer, P. Botella (Eds.), Software Engineering ESEC '95, volume 989 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1995, pp. 137–153.

[36] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger, A survey of self-management in dynamic software architecture specifications, in: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, WOSS '04, ACM, New York, NY, USA, 2004, pp. 28–33.

[37] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley, 1996.

[38] M. Shaw, D. Garlan, Software architecture: perspectives on an emerging discipline, Prentice Hall, 1996.

[39] L. Bass, P. Clements, R. Kazman, Software architecture in practice, Addison-Wesley Longman Publishing Co., Inc., second edition, 2003.

[40] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, A. Lluch Lafuente, Graph-based design and analysis of dynamic software architectures, in: P. Degano, R. Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models, volume 5065 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 37–56.

[41] D. Jackson, Alloy: a lightweight object modelling notation, ACM Trans. Softw. Eng. Methodol. 11 (2002) 256–290.