Computer Science and Information Systems 12(2):607–634 DOI: 10.2298/CSIS140912019R

Towards an engine for coordination-based architectural reconfigurations

Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa

HASLab - INESC TEC & Universidade do Minho
Braga, Portugal
pg22826@alunos.uminho.pt
nuno.s.oliveira@inesctec.pt
luis.s.barbosa@inesctec.pt

Abstract. Software reconfigurability became increasingly relevant to the architectural process due to the crescent dependency of modern societies on reliable and adaptable systems. Such systems are supposed to adapt themselves to surrounding environmental changes with minimal service disruption, if any.
This paper introduces an engine that statically applies reconfigurations to (formal) models of software architectures. Reconfigurations are specified using a domain specific language— ReCooPLa—which targets the manipulation of software coordination structures, typically used in service-oriented architectures (soa). The engine is responsible for the compilation of ReCooPLa instances and their application to the relevant coordination structures. The resulting configurations are amenable to formal analysis of qualitative and quantitative (probabilistic) properties.

Keywords: domain-specific languages, architectural reconfiguration, coordination.

1. Introduction

This paper addresses reconfiguration of the coordination layer in service-oriented architectures (SOA). As the typical architectural style underlying modern adaptable and intensive software systems [15], SOA depends crucially on the set of protocols that interconnect services across different platforms and providers. If services are understood as distributed, loosely-coupled entities offering a specific computational functionality via published interfaces, the *glue code* that keeps them together, adapting and articulating their interfaces to achieve the desirable emergent behaviour, constitutes the *coordination layer*. Interaction, as a main concern in software design, may be achieved and encapsulated in a multitude of ways [6]. The so-called coordination approach [7] favours a complete decoupling of the individual sources of computation (services, components) from the protocols that govern their interaction. The latter are encoded into software *connectors*, following approaches of traditional frameworks like Reo [7] in which complex connectors are built compositionally.

This separation of concerns makes SOA flexible and naturally dynamic: although policies are usually pre-established at design time, services may be discovered and bound to the architecture only at run time [17]. Such a dynamic behaviour plays an important role in the context of adaptable systems. For instance, whenever a service stops providing results, because of a failure in the remote server where it runs, a similar one (usually with equal interface and functionality) may be found and chosen as a replacement. The

whole system is, therefore, able to continue performing within specific levels of quality, previously agreed between consumer and provider, without significant or no disruption at all.

Usually, reconfigurations in SOA target services themselves, for example, through the dynamic update of a service functionality, addition of new services, substitution of services with compatible interfaces (but not necessarily the same behaviour) or removal of unnecessary services [22,32,43,46,49]. However, in some situations, this is not enough. For instance, when a substituting service has an incompatible interface, it may be necessary to deal carefully with the way it interacts with other services. Another typical example is the case when an emerging requirement establishes that a synchronous communication shall thereafter become asynchronous. To tackle these situations, reconfigurations must address the coordination layer and be able to modify its topology, for example, by adding or removing specific connectors, moving communication interfaces between components and rearranging complex interaction structures [27,28]. In the sequel this sort of architectural reconfiguration will be qualified as *coordination-based*.

Oliveira and Barbosa [37,38] recently proposed a formal framework for modelling and analysing (at design time) coordination-based reconfigurations in the context of SOA. In this framework, a coordination structure (referred to as a *coordination pattern*) is regarded as a graph whose nodes represent interaction points with either services or other coordination patterns. Edges, on the other hand, are communication channels uniquely identified, and exhibiting a specific behaviour. Finally, reconfigurations are obtained by sequential combination of primitive operations that manipulate the graph-like structure of coordination patterns.

This paper aims at providing a *proof of concept* for this framework. Therefore, it introduces a domain specific language—referred to as ReCooPLa—and a reconfiguration engine to express coordination-based reconfigurations and apply them to coordination patterns expressed in CooPLa, a tiny domain specific language also presented in the sequel.

Domain specific languages [14,34,40] focus on particular application domains and build on specific domain knowledge. Their level of abstraction is tailored to the intended domain, allowing for embedding its vocabulary and concepts into the language constructors, and hiding low-level details under their processors. In addition, they allow for validation and optimisation at domain level, offering considerable gains in expressiveness and ease of use, compared with general-purpose programming languages [26]. In this spirit, ReCooPLa provides a precise, high-level interface for the software architect to plan and experiment with reconfiguration strategies.

Contributions. As a revised and extended version of reference [47], the paper s contributions are as follows:

- full description of the CooPLa language for the design of coordination patterns, which, although briefly mentioned in [37], was never documented, and extension of its twin language, ReCooPLa, with new constructors to deal with the application of reconfigurations upon running instances of coordination patterns;
- development of a reconfiguration engine based on ReCooPLa;
- discussion of a case study in the area of adaptive systems as a possible illustration scenario.

Outline. Background notions are introduced in Section 2. This includes an informal exposition of both the reconfiguration framework and the CooPLa domain-specific language for the specification of coordination structures. In Section 3 the ReCooPLa language is introduced in detail and illustrated by small examples. Then, Section 4 introduces the ReCooPLa engine focusing on the reconfiguration engine model and the suitable translation of ReCooPLa constructors into that model. Section 5 presents a case study in the area of (self-)adaptive systems. Related work is presented in Section 6 and finally, Section 7 concludes the paper and proposes some topics for future work.

2. Coordination patterns

The model. The reconfiguration model introduced in [37,38] is strongly based in exogenous coordination models, *e.g.* Reo [7] or BIP [8].

Thus, a coordination structure, referred to as *coordination pattern* in [37], is a reusable, compositional architectural element formalised as a graph of channels. Nodes in the graph are interaction points through which other coordination patterns or services can be plugged together. Edges, on the other hand, are uniquely identified point-to-point communication devices with a specific behaviour, generically mentioned as *channels*.

A channel has two ends. They can be classified as *source* ends, when they admit data into the channel, or *sink* ends, when they dispense data out of the channel. Formally, a channel c is a structure

$$c \in 2^{\mathcal{E}} \times \mathcal{I} \times \mathcal{T} \times 2^{\mathcal{E}},$$

where $\mathcal{E}$ is a set of channel ends, $\mathcal{I}$ is a set of identifiers and $\mathcal{T}$ is a set of channel types. Channel types can be defined by the software architect to meet particular needs — the model and the language is independent of them. We assume, however, a small set of basic channels, borrowed from the Reo framework: $\mathcal{T} = \{\mathsf{sync}, \mathsf{lossy}, \mathsf{fifo}, \mathsf{drain}\}$. In brief, the sync channel transmits data synchronously from one end to another whenever an input and an output request are simultaneously present at both channel ends, otherwise one request has to wait for the other. The lossy channel behaves likewise, but data may be lost whenever an output request (at the source end) is not matched by an input one (at the sink end). Differently, a fifo channel has buffering capacity of (usually) one memory position, therefore allowing for asynchronous occurrence of input/output requests. Finally, the drain channel accepts data synchronously at both ends and loses it (note that, in this case, both ends are actually sink ends).

Nodes in a coordination pattern represent interaction points. The latter are locations where channels synchronise their interfaces (ends) for interaction with other patterns or external components.

Thus, we define a coordination pattern ρ as a pair

$$\rho = \langle \mathcal{C}, \mathcal{N} \rangle,$$

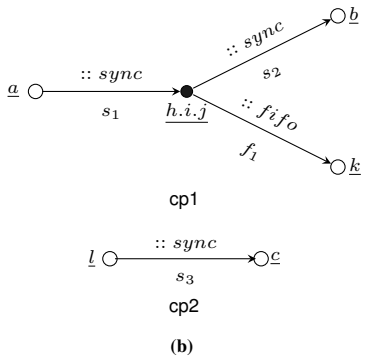
where $\mathcal{C}$ is a set of channels, and $\mathcal{N}$ a partition of the union of all ends of all channels in $\mathcal{C}$.

Fig. 1 presents two coordination patterns. Pattern `cp1` comprises three channels: s_1 and s_2 of type sync, and f_1 of type fifo. Channel s_1 has an input end `a` and an output end `h`. In its turn, channel s_2 has an input end `i`, and an output end `b`. Finally, channel f_1 has

an input end `j` and an output end `k`. The nodes are formed by sets of ends[1] that convey the connection between the channels.

```
1  cp1: <{
2     <{a}, s1, sync, {h}>,
3     <{i}, s2, sync, {b}>,
4     <{j}, f1, fifo, {k}>},
5     {a, b, k, h.i.j}
6  >
7
8  cp2: <{
9     <{l}, s3, sync, {c}>},
10    {l, c}
11 >
```

(a)



(b)

Fig. 1. Textual (a) and graphical (b) description of two simple coordination patterns.

New patterns from old. Let us turn attention to the way a coordination pattern can be modified into another. This is achieved through a number of operations described below. Anticipating Section 3, we should emphasise that those are exactly the operations which are used, not only to build new patterns from old, but also to reconfigure them. This explains way they are also referred to as *reconfiguration primitives*. They are introduced throughout the following paragraphs.

Let ρ be a coordination pattern. The simplest reconfiguration primitives are the identity (`id`) and the constant (`const(`ρ`)`) operations. Expectedly, when applied to a coordination pattern, the former keeps it unchanged, while the latter replaces it with ρ.

The `par(`ρ`)` primitive sets the original coordination pattern in parallel with ρ, without creating any connection between them. It is assumed, without loss of generality, that nodes and channel identifiers in both patterns are disjoint. Fig. 2 depicts the resulting coordination pattern, after applying `par(cp2)` to `cp1`.

The `join(`N`)` primitive, where N is a set of nodes, creates a new node by merging all nodes in N, into a single one. For instance, applying `join(k,l)` to `cp1` (*c.f.*, Fig. 2) creates a connection on node `k.l`, as depicted in Fig. 3. The new pattern is usually referred to as the `Sequencer`, because ports `b` and `c` are activated in sequence after detecting an external stimulus in port `a`.

The `split(`n`)` primitive, where n is a node, is dual to `join`: it breaks connections within a coordination pattern by separating all channel ends coincident in n. Fig. 4 illustrates the application of `split(h.i.j)` to `cp1` in Fig. 3.

Finally, the `remove(`c`)` primitive, where c is a channel identifier, removes channel c, if it exists, from the coordination pattern. In addition, if c was connected to other channel(s),

[1] Notation `h.i.j` is used to express the node {`h`, `i`, `j`}, where `h`, `i` and `j` are channel ends. For presentation sake, traditional set notation is replaced by a line under the elements of the set.

```
cp1: <{
    <{a}, s1, sync, {h}>,
    <{i}, s2, sync, {b}>,
    <{j}, f1, fifo, {k}>,
    <{l}, s3, sync, {c}>},
    {a, h.i.j, b, k, l, c}
>
```

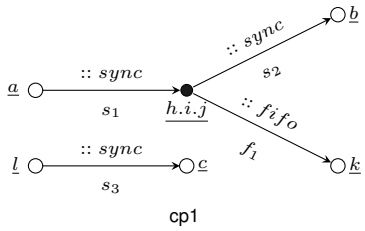


Fig. 2. The result of applying the `par` primitive.

```
cp1: <{
    <{a}, s1, sync, {h}>,
    <{i}, s2, sync, {b}>,
    <{j}, f1, fifo, {k}>,
    <{l}, s3, sync, {c}>},
    {a, h.i.j, b, c, k.l}
}
```

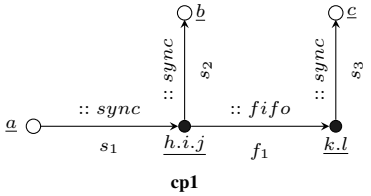


Fig. 3. The result of applying the `join` primitive.

```
cp1: <{
    <{a}, s1, sync, {h}>,
    <{i}, s2, sync, {b}>,
    <{j}, f1, fifo, {k}>,
    <{l}, s3, sync, {c}>},
    {a, h, i, j,b, c, k.l}
>
```

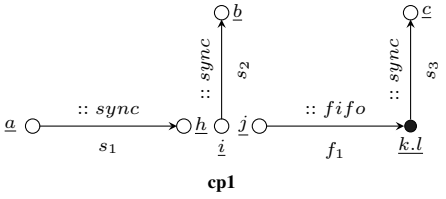


Fig. 4. The result of applying the `split` primitive.

these connections are also broken as it happens with `split`. Fig. 5 depicts the result of applying `remove(s3)` to `cp1` in Fig. 4. Note how node `k.l` was split and its end `l` removed along with channel s_3.

```
1 cp1: <{
2     <{a}, s1, sync, {h}>,
3     <{i}, s2, sync, {b}>,
4     <{j}, f1, fifo, {k}>},
5     {a, h, i, j, b, k}
6 >
```

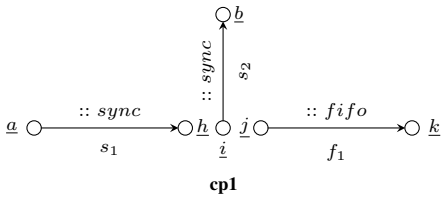


Fig. 5. The result of applying the `remove` primitive.

These operations are assumed to be applied in sequence. Their parallel application is also possible, but only when they can be shown to be mutually independent: *i.e.*, affecting separated substructures of the target coordination pattern. The possibility of composing primitive operations in sequence or parallel, allows for the definition of complex reconfigurations, referred to as *reconfiguration patterns*. Those affect significant parts of a coordination pattern at the same time, and are expected to be generic, parametric and reusable. This notion is relevant in ReCooPLa, the reconfiguration language to be introduced in section 3. Before that, however, the basic DSL for expressing coordination patterns has to be presented.

2.1. CooPLa - A language for coordination patterns

The purpose of CooPLa (**Coo**rdination **P**atterns **La**nguage) is to materialise into a DSL the structure of a coordination pattern, as presented above. Thus, the language offers to software architects a tool to design the coordination layer of a system by building and composing coordination patterns.

Channels. In CooPLa, a channel is specified as a structure with well defined input and output ports and a behaviour. The behaviour is given as a list of cases which defines how data flows within the channel depending on its internal state and on the stimulus received on its ports.

Channels differ from each other mainly by their behaviour and possible internal specificities. Indeed, CooPLa allows for the definition of channels with (i) a structure (*e.g.*, a buffer), to store data in asynchronous communications; this is expressed with constructor `~K`, where K is either a value or a list of values; (ii) a clock, to impose delays in a normal data flow; constructor `@T` is used for this, where T is the delay; and finally (iii) a pattern-based condition, to match data and decide the course of future flow based on the result of the matching; `cond=<...>` is the corresponding construct.

Behaviour, on the other hand, is defined by rules `R->f`, where `R` is a subset of ports of the channel or internal state observers (possibly negated with `!`) and `f` is a flow as explained below. It means that whenever there are read/write requests at the ports in `R` or its state observers hold, flow `f` occurs.

A flow is defined by the constructor `flow p1 to p2`, where `p1` and `p2` refer to ports of the channel, the internal state of the channel or `NULL` (a special port where data is lost or automatically produced). Its meaning is that data flows from `p1` to `p2`. Flows may occur synchronised or restricted by a condition. For the former case, the constructor `f1|f2` is used to express synchronisation between flows `f1` and `f2`. For the latter, *e.g.*, when a channel has a datatype condition `cond` that needs to be met, the constructor `cond ? f1 : f2` is used to trigger `f1` if the condition holds, or `f2`, otherwise. Flows may be further annotated with a stochastic label of the form `#L`, where each `L` is a unique identifier within the channel. These labels are used to assign to a flow a processing delay rate upon which stochastic analysis of coordination patterns can be achieved as discussed in [41].

Finally, a new channel may extend a basic one `b` through the addition of new flows, with the constructor `extends b` appended to its signature.

Fig. 6 shows how four well-known Reo channels are defined in CooPLa.

```
channel sync(a:b) {
   a,b -> flow a to b #ab ;
}
```

```
channel lossy(a:b) extends sync {
   a,!b -> flow a to NULL #aL;
}
```

```
channel fifo~N(a:b){
  state: buffer; observers: E, F;
  a,!F -> flow a to buffer #aB;
  !E,b -> flow buffer to b #Bb;
}
```

```
channel drain(a,b:) {
   a,b -> flow a to NULL
            |
          flow b to NULL #ab;
}
```

Fig. 6. CooPLa description of four basic Reo channels.

The `sync` channel may be read as a structure with input port `a` and output port `b`. Whenever there are input/output requests pending at both ports simultaneously, then data flows from `a` to `b`.

The `lossy` channel extends `sync` with an extra flow: whenever there is a write (output) request at port `a` and no read (input) request at port `b` (notice the use of `!` to convey negation), then data is lost (*i.e.*, it flows from `a` to `NULL`).

The `fifo` channel is a structure with input port `a`, output port `b`, and a state named `buffer` with dimension `N`. The observers `E` and `F` are operations over the state that check whether the `buffer` is (E)empty and (F)ull, respectively. The behaviour of this channel is defined taking into account the pending requests at the ports as well as the value of its internal state. As an example, data flows from port `a` to the buffer only when there is a writing request at `a` and the `buffer` is not full.

Finally, the `drain` channel expects simultaneous requests at its two input ports; whenever this clause is fulfilled, data flows synchronously (notice the use of constructor `|`) from each of these ports to `NULL`, being therefore lost.

Patterns. In CooPLa, a coordination pattern is specified by its set of ports and a body which defines their interconnection.

This is achieved in two stages. First, under the reserved word `use`, all the elements to be connected in the pattern are declared (*i.e.* instantiated); this declaration resorts to the signature of channels and coordination patterns using logical names to refer to the corresponding ports. Then, under the reserved word `in`, the ports of the pattern are concretely defined and, only then, the assembly of interconnections is performed.

As an illustration, Fig. 7 (left) shows how the `Sequencer` coordination pattern obtained in Fig. 3 is specified in CooPLa.

```
pattern Sequencer (a : b, c) {

use :
   sync(i:o) as s1, s2, s3;
   (E)fifo~1(i:o) as f1;
in:
    a  = s1.i
    b  = s2.o;
    c  = s3.o;
    join [s1.o, s2.i, f1.i] as hij;
    join [f1.o, s3.i] as kl;
}
```

```
stochastic Sequencer @ {
   a     = 100.0;
   b     = 10.0;
   c     = 90.0;
   s1#ab = 1000.0;
   s2#ab = 500.50;
   s3#ab = 1000.0;
   f1#aB = 980.45;
   f1#Bb = 1500.0;
   hij   = 100000.0;
   kl    = 100000.0;
} sseq
```

Fig. 7. CooPLa specification of the Sequencer (left) and a stochastic instance of it (right).

The construction of patterns is intended to be intuitive. In the declaration of channels used in this pattern, the names `i` and `o` are used as logical references to the input and output ports of each channel. The declaration of the `fifo` channel has to define the concrete value for the buffer dimension, and optionally its initial configuration (empty in the case). The actual composition of channels is carried out by binding ports. The connections between the composed elements are defined using the join operation.

Since all the channels used in the `Sequencer` coordination patterns provide stochastic labels, then a stochastic instance of the `Sequencer` could be derived as shown in Fig. 7 (left). The stochastic extension of CooPLa is still under development and will be omitted in the sequel; but basically it consists of adding a list of identifiers (nodes, stochastic labels or ports) to which processing delay and request arrival rates are assigned.

CooPLa specifications form models of coordination patterns, implemented in the Java programming language, as shown in Fig. 8. Such models are high-level representations of the system s coordination layer. From this model different kinds of analysis and transformations can be developed. Pattern reconfiguration is, certainly, a main one.

3. ReCooPLa: A reconfiguration language

ReCooPLa (**Re**configuration for **Coo**rdination **P**atterns **La**nguage) is a domain-specific language for designing coordination-based reconfigurations.

According to the Domain-specific Language (DSL) classification system introduced by Mernik, Heering and Sloane [34], ReCooPLa can be classified inside the *decision* phase [1],

[1] The framework introduces four development phases: decision, analysis, design, and implementation.

as a mixture of the notation and the task automation patterns. Indeed, the language encodes domain-level concepts in the notation hiding an API for programming reconfigurations; also it serves as a front-end that automates low level, often error prone programming details. Its analysis was based on a formal analysis pattern, resorting to ontology-based domain engineering [16]. Its design is strongly based on attribute grammars; this makes ReCooPLa to fit in the formal design pattern. Finally, the implementation of ReCooPLa followed a compiler/application generator pattern: its constructors are translated into suitable Java code as discussed later in the paper.

3.1. Overview

In ReCooPLa, a *reconfiguration* is a first-class citizen, as much as functions or procedures are in common, general-purpose programming languages. Similarly to the latter, a reconfiguration has a signature which specifies its identifier and arguments, and a body which prescribes a specific behaviour. However, a reconfiguration is always applied to, and always returns, a coordination pattern. Additionally, reconfigurations accept arguments of the following data types: *Name*, *Node*, *XOR Set*, *Pair*, *Triple*, *Pattern* and *Channel*.

The reconfiguration body is a list of instructions, most of them concerned with the *application* of (primitive, or previously defined) reconfigurations. As auxiliary operations, other ReCooPLa constructors act on the parameters of a reconfiguration. In particular, they provide ways to declare, assign and manipulate local variables, for example, field selectors, and to use typical connectives (as set union, intersection, subtraction and an iterative control structure over the elements of a set).

3.2. The Language

In the sequel, ReCooPLa is introduced through (the most relevant) fragments of the underlying grammar. A number of constructors are defined for further reference in the paper.

Reconfiguation. A reconfiguration (see Listing 1.1) consists of a header, under the reserved word `reconfiguration` followed by a unique identifier (the reconfiguration name) and a list of arguments, which may be empty, followed by the body. The latter is a list of instructions as explained below.

Listing 1.1. EBNF notation for the *reconfiguration* production.

```
1 reconfiguration
2   : reconfiguration ID ( args* ) { instruction+ }
3 args    : arg ( ; arg)*
4 arg     : datatype ID ( , ID)*
```

A reconfiguration constructor is represented as $rcfg(n, t_1, a_1, \ldots, t_k, a_n, b)$, where n is the reconfiguration identifier; each a_i is an argument of type t_i; and b its body.

Data types. ReCooPLa builds on a small set of data types: primitive (*Name*, *Node* and *XOR*), generic (*Set*, *Pair* and *Triple*) and structured (*Pattern* and *Channel*). *Name* is a string and represents a channel identifier or a channel end. *Node*, although considered as a primitive data type, is internally regarded as a set of names, for compatibility with its definition in Section 2. *XOR* is a particular case of *Node*, which has at least one input end and two (mutual exclusive) output ends. The generic data types (based on the Java generics) specify a type for their contents, as shown in Listing 1.2.

Listing 1.2. EBNF notation for the *datatype* production.

```
datatype: ...
   | ( Set | Pair | Triple ) < datatype >
```

Structured data types have an internal state, according to their definition in Section 2. Each instance of these types is endowed with attributes and operations, which can be accessed using selectors (see below).

The constructor of a data type is either given as $T()$ or $T_G(t)$, where T is a ReCooPLa data type and t is a subtype of a generic data type T_G.

Reconfiguration body. The reconfiguration body is a list of instructions, where each instruction can be a declaration, an assignment, an iterative control structure, or an application of a reconfiguration. A declaration is expressed as usual: a data type followed by an identifier or an assignment. In its turn, an assignment associates an expression, or an application of a reconfiguration, to an identifier. The respective constructors are, then, $decl(t, v)$ and either $assign(t, v, e)$ or $assign(v, e)$, where t is a data type, v a variable name and e an expression.

The control structure `forall` is used to iterate over a set of elements. Again, a list of instructions defines the behaviour of this structure. The corresponding production rule is shown in Listing 1.3.

Listing 1.3. EBNF notation for the *forall* production.

```
forall: forall ( datatype ID : ID ) { instruction+ }
```

The constructor for the iterative control structure is $forall(t, v_1, v_2, b)$, where t is a data type, v_1, v_2 are variables and b is a set of instructions.

The application of a reconfiguration (see the `reconfiguration_apply` production in Listing 1.4), is expressed by an identifier followed by the `@` operator and a reconfiguration name. The latter may be a primitive reconfiguration or any other reconfiguration previously declared. The `@` operator stands for *application*. A reconfiguration is applied to a variable of type *Pattern*. In particular, this variable may be omitted (optional identifier in the production rule `reconfiguration_apply`); when this is the case, the reconfiguration is applied to the original pattern. This typical use is shown in Listing 1.8.

Listing 1.4. EBNF notation for the *reconfiguration_apply* production.

```
reconfiguration_apply
   : ID? @ reconfiguration_call
reconfiguration_call
   : ( join | split | par | remove | const | id |ID) op_args
```

Application is specified either as $@(c)$ or $@(p, c)$, where p is a *Pattern* and c a reconfiguration call. A reconfiguration is called as $r(a_1, \ldots, a_n)$, for r a reconfiguration name, and each a_i one of its arguments.

Operations. An expression is composed of one or more operations. They can be specific constructors for generic data types, including nodes, or operations over generic or structured data types, as shown in Listing 1.5.

Each constructor is defined by a reserved word (`S` stands for *Set*, `P` for *Pair* and `T` for *Triple*), and a list of values which is expected to comply to the data type involved, as illustrated in Listing 1.6.

Listing 1.5. EBNF notation for the *constructor* production.

```
constructor
        : P ( expression , expression )
        | T ( expression , expression , expression )
        | S ( ( expression ( , expression)*)? )
```

Listing 1.6. *Constructors* input example.

```
    Pair<Node> a = P(n1, n2);
    Triple<Pair<Node>> b = T(a, P(n1,n2), P(n3,n4));
    Set<Node> c = S(n1, n2, n3, n4, n5, n6);
```

The relevant constructors are $P(e_1, e_2)$, $T(e_1, e_2, e_3)$ and $S(e_1, \ldots, e_n)$ for the Pair, Triple and Set constructors, respectively; with each e_i representing an expression.

For the Set data type, ReCooPLa provides the usual binary set operators: `+` for union, `-` for subtraction and `&` for intersection. For the remaining data types (except *Node*, *XOR* and *Name*), selectors are used to apply the operation, as shown in Listing 1.7 (production rule `operation`). Symbol `#` is used to access a specific channel from the internal structure of a pattern.

Listing 1.7. EBNF notation for the `operation` and `attribute_call` productions.

```
operation
        : ID ( # ID)? . attribute_call
attribute_call
        : in ( ( INT ) )?
        | out ( ( INT ) )?
        | name | nodes | names
        | fst | snd | trd
```

An `attribute_call` corresponds to an attribute or an operation associated to the last identifier, which must correspond to a variable of type *Channel*, *Pattern*, *Pair* or *Triple*. The list of attributes/operations in the language is presented in Listing 1.7 and described below:

- `in`: returns the input ports from the *Pattern* and *Channel* variables. It is possible to obtain a specific port by providing an optional integer parameter indexing a specific entry from the set (seen as a 0-indexed array).
- `out`: returns the output ports from the *Pattern* and *Channel* variables. The optional parameter can be used as above for a similar effect.
- `name`: returns the name of a *Channel* variable, i.e., a channel identifier.
- `nodes`: returns all input and output ports plus all the internal nodes of a *Pattern* variable.
- `names`: returns all channel identifiers associated to a *Pattern* variable.
- `fst`, `snd`, `trd`: are, respectively, the first, second and third projection of a tuple (*Pair* and *Triple* variables).

All these operations give rise to their own language constructors. Field selection is specified by $\bullet(v, c)$, where v is a variable and c a call to an operation. The constructor for the `#` operator is $\#(p, n)$, where p is a pattern and n is a channel identifier. Constructors for the set operators fare similarly defined: $+(s_1, s_2)$, $-(s_1, s_2)$ and $\&(s_1, s_2)$, for union, difference and intersection, respectively, with s_1, s_2 being variables of the sort *Set*. The constructors for the other operators are generalised as either $oper(a)$ or $oper()$, depending on whether the operation with name $oper$ has an argument a or not.

Listing 1.8 shows an example of valid ReCooPLa sentences.

Listing 1.8. A first example.

```
reconfiguration removeP (Set<Name> Cs ) {
    forall ( Name n : Cs) {
        @ remove(n);
    }
}
reconfiguration overlapP(Pattern p; Set<Pair<Node>> X) {
    @ par (p);
    forall(Pair<Node> n : X) {
        Node n1, n2;
        n1 = n.fst;
        n2 = n.snd;
        Set<Node> E = S(n1, n2);
        @ join(E);
    }
}
```

Therein, two reconfigurations are declared: `removeP` and `overlapP`. The former removes from a coordination pattern an entire set of channels by applying the `remove` primitive repeatedly. The latter sets a coordination pattern in parallel with the original one, using the `par` primitive, and performs connections between the two patterns by applying the `join` primitive with suitable arguments.

Main. A special reconfiguration block, marked with the reserved word `main`, is used to specify the actual application of reconfigurations to coordination patterns. It accepts a (possibly empty) list of arguments aggregated by data type, as in a normal reconfiguration. The difference is that in this case, data types are only references to available coordination patterns expressed in imported CooPLa files. The arguments are, thus, assumed as instances of the given patterns that are to be reconfigured. These instances may be concrete, when the argument identifier matches the identifier of stochastic instances declared in CooPLa files; or anonymous, otherwise.

Listing 1.9 presents a partial grammar for the syntax of these main reconfigurations.

Listing 1.9. EBNF notation for the *main* production.

```
main      : main [ main_args* ] { main_instruction+ }
main_args : main_arg ( ; main_arg)*
main_arg  : CPNAME ids
ids       : ID ( , ID)*
```

The corresponding constructor is $main(cp_1, a_1, \ldots, cp_k, a_n, b)$, where each a_i is an instance of a coordination pattern of type cp_i; and b its body. The latter amounts to a list of specific instructions, where an instruction is either a declaration, an explicit or an implicit assignment.

A declaration is expressed here as usually, with a data type and a list of identifiers. The data type corresponds to a coordination pattern name, which may or may not exist in the imported ones. In the latter case it becomes a structureless coordination pattern. The declaration constructor is $decl_m(t, v)$, where t is a name (for a coordination pattern) and v is the variable (instance) being declared. In its turn, an assignment in the main reconfiguration block associates the result of an expression to a declared pattern. The expressions available in this context are limited to the concrete application of a reconfiguration to pattern instances, either passed as arguments or freshly declared. This is usually used to assign a structure to a structureless pattern previously declared. Assignments are explicit when the result of applying a reconfiguration is stored in a declared variable; they are implicit otherwise. Since reconfigurations change the structure of the pattern to which they are applied, the result of an implicit assignment is stored in the reconfigured coordination pattern. The `@` symbol is again used to express the application of reconfigurations. The arguments of a reconfiguration are obtained from the arguments of the main body and freshly declared pattern instances, using the operations explained above to access nodes, channels and alike. The constructors for these specific instructions are $assign_m(t, v, a, r, e_1, \ldots, e_n)$, $assign_m(v, a, r, e_1, \ldots, e_n)$ or $assign_m(a, r, e_1, \ldots, e_n)$, where t is a data type (coordination pattern), v a variable name, a another coordination pattern (usually an argument of the main reconfiguration), r a reconfiguration call with each e_i one of its suitable argument.

A simple example of a main reconfiguration is presented in Listing 1.10. Therein, an instance `sseq` of the `Sequencer` coordination pattern (as depicted in Fig. 7) is reshaped through the application of the `removeP` reconfiguration (introduced in Listing 1.8). Notice the files containing the patterns and reconfigurations involved are previously imported.

Listing 1.10. An example of a main block reconfiguration in ReCooPLa.

```
import patterns.cpla;
import reconfigs.rcpla;

main [Sequencer sseq] {
   sseq @ removeP(S(sseq#s1.name));
}
```

4. An engine for architectural reconfiguration

This section introduces the engine, which executes reconfigurations specified in ReCooPLa. It is composed of two main modules —the reconfiguration and the translation engine— described in the following sections.

4.1. Reconfiguration Engine

As it often happens with domain specific languages, ReCooPLa is translated into a subset of Java, which is then recognised and executed by an engine. This engine, referred to as the *Reconfiguration Engine*, is developed in Java to execute reconfigurations specified in ReCooPLa over coordination patterns, which are defined in CooPLa. The model of the engine is simple, based only in a few entities. Fig. 8 presents the corresponding Unified Modelling Language (UML) class diagram.

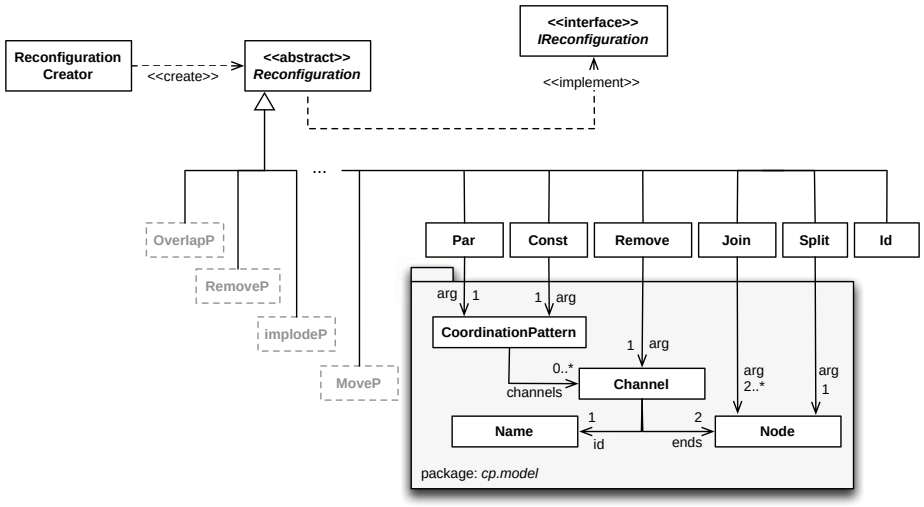


Fig. 8. The reconfiguration engine model.

Package *cp.model*, represented as a shaded diagram, concerns the model of a coordination pattern. Actually, this is the implementation of the formal model presented in Section 2 as well as the target model of the CooPLa language processor. Both CoordinationPattern and Channel classes provide attributes and methods that match the attributes and operations of the *Pattern* and *Channel* types in ReCooPLa. For instance, the attribute `nodes` of the *Pattern* type has its corresponding method `getNodes()` in the CoordinationPattern class.

The remaining entities in the diagram deal with reconfigurations themselves, and are also assumed to belong to a *cp.reconfiguration* package. Clearly, classes Par, Const, Remove, Join, Split and Id are the implementation of the corresponding primitive reconfigurations introduced in Section 2. The relationships with the elements of the *cp.model*

package define their arguments. These classes implement the abstract class Recon guration, which represents a generic reconfiguration. Because it *implements* the IRecon guration interface, all its subclasses have the implicit method: apply(CoordinationPattern p), which is where the behaviour of the reconfigurations is defined as the combined effect of their application to the coordination pattern p given as an argument.

The careful reader may have noticed that some of the concrete classes of Recon guration are greyed-out, and also that they are not all presented. This is where the most interesting part of the engine comes into play. In fact, there are no such concrete classes (apart of the primitives) at compile time. They are created dynamically, at run time, by the Recon gurationCreator class, and loaded into the running Java Virtual Machine (JVM), taking advantage of its reflection features. This implementation follows a similar approach to the well-known Factory design pattern, but instead of creating instances, it creates concrete classes of Recon guration. The idea is that each reconfiguration definition within a ReCooPLa specification gives rise to a new subclass of Recon guration with an apply(CoordinationPattern p) method, whose content is derived from the content of the ReCooPLa reconfiguration, conveniently translated into a Java file. The engine is still responsible for automatically compiling such files into Java class files and dynamically load them into the running JVM to be used in the reconfiguration process as defined by the *main* reconfiguration. The *main* reconfiguration is itself translated and compiled into a *Run.class* file, which is then used from within the ReCooPLa engine to enact the reconfiguration process.

However, for this to be possible, it is first necessary to correctly translate ReCooPLa constructors into the code accepted by the Reconfiguration Engine. Section 4.2 goes through the details of such a translation.

4.2. ReCooPLa Translation

In order not to burden the diagram in Fig. 8, a number of classes were omitted. These classes match the types further accepted in ReCooPLa: Pair, with a getFst() and a getSnd() methods to access its fst and snd attributes; Triple, extending Pair with an attribute trd and method getTrd(); and the LinkedHashSet from the *java.util* package, which is hereafter abbreviated to LHSet for increasing readability.

To keep exposition simple, some minutiae like imports, semicolons, annotations, auxiliary variables, control or try-catch structures and efficiency concerns are ignored in this schema. Moreover, abstractions are used to wrap complex constructions; for instance, method mkRecfg(n, t_1, a_1, …, t_k, a_n, b) abstracts details of the creation of a Reconfiguration class with name n; attributes $a_1, \ldots, a_n$ of type $t_1, \ldots, t_k$; and method apply with body b, which always ends with a return p instruction, where p is the argument of apply.

This being said, the translation of ReCooPLa constructors into the Reconfiguration Engine is given by the rule-based function $\mathcal{T}(C)$, where C is a constructor of ReCooPLa as presented in Section 3. Table 1 defines $\mathcal{T}()$[2].

[2] By convention n is used for identifiers; t, t_i for data types; a_i for arguments; b for set of instructions; T for non-generic data type; T_G for generic data type, except *Set*; v, v_i for local variables; e, e_i for expressions; p for patterns; s_i for sets; c for channel names; i for numbers; and finally *oper* for the operations enumerated in Section 3.2.

[3] T comes from the context where the constructor appears or the type of composing expressions e_i.

[4] For horizontal space reasons, CoordinationPattern is abbreviated to CoordPatt.

Table 1. Translation rules for ReCooPLa constructors.

$$
\begin{aligned}
\mathcal{T}(rcfg(n, t_1, a_1, ...t_k, a_n, b)) &\rightarrow \text{mkRcfg}(n, \mathcal{T}(t_1), a_1, ... \mathcal{T}(t_k), a_n, \mathcal{T}(b)) \\
\mathcal{T}(T()) &\rightarrow \text{T} \\
\mathcal{T}(T_G(t)) &\rightarrow \text{T}_G <\mathcal{T}(t)> \\
\mathcal{T}(Set(t)) &\rightarrow \text{LHSet}<\mathcal{T}(t)> \\
\mathcal{T}(decl(t, v)) &\rightarrow \mathcal{T}(t)\ v \\
\mathcal{T}(assign(t, v, e)) &\rightarrow \mathcal{T}(decl(t, v)) = \mathcal{T}(e) \\
\mathcal{T}(assign(v, e)) &\rightarrow \text{v} = \mathcal{T}(e) \\
\mathcal{T}(forall(t, v_1, v_2, b)) &\rightarrow \text{for}(\mathcal{T}(t)\ v_1 : v_2)\{\mathcal{T}(b)\} \\
\mathcal{T}(@(r(e_1, \ldots, e_n))) &\rightarrow \text{r rec = new r}(\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)); \text{rec.apply}(p) \\
\mathcal{T}(@(r(p, e_1, \ldots, e_n))) &\rightarrow \text{r rec = new r}(\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)); \text{rec.apply}(p) \\
\mathcal{T}(P(e_1, e_2)) &\rightarrow \text{new Pair}(\mathcal{T}(e_1), \mathcal{T}(e_2)) \\
\mathcal{T}(T(e_1, e_2, e_3)) &\rightarrow \text{new Triple}(\mathcal{T}(e_1), \mathcal{T}(e_2), \mathcal{T}(e_3)) \\
\mathcal{T}(S(e_1, \ldots, e_n)) &\rightarrow \text{new LHSet<T>()\{\{add}(\mathcal{T}(e_1)); \ldots; \text{add}(\mathcal{T}(e_n)); \}\}\ ^{3} \\
\mathcal{T}(N(n_1, \ldots, n_n)) &\rightarrow \text{new Node(new LHSet<String>()\{\{add}(n_1); \ldots; \text{add}(n_n); \}\}) \\
\mathcal{T}(+(s_1, s_2)) &\rightarrow \text{(new LHSet}(s_1)).\text{addAll}(s_2) \\
\mathcal{T}(-(s_1, s_2)) &\rightarrow \text{(new LHSet}(s_1)).\text{removeAll}(s_2) \\
\mathcal{T}(\&(s_1, s_2)) &\rightarrow \text{(new LHSet}(s_1)).\text{retainAll}(s_2) \\
\mathcal{T}(\#(p, c)) &\rightarrow \text{p.getChannel}(c) \\
\mathcal{T}(\bullet(v, c)) &\rightarrow \text{v.}\mathcal{T}(c) \\
\mathcal{T}(in(i)) &\rightarrow \text{getIn}(i) \\
\mathcal{T}(out(i)) &\rightarrow \text{getOut}(i) \\
\mathcal{T}(ends(p)) &\rightarrow \text{getEnds}(p) \\
\mathcal{T}(oper()) &\rightarrow \text{getOper()}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}(main(cp_1, a_1, ...cp_k, a_n, b)) &\rightarrow \text{mkMain}(Run, \mathcal{T}(cp_1, a_1), ... \mathcal{T}(cp_k, a_n), \mathcal{T}(b)) \\
\mathcal{T}(cp, a) &\rightarrow \text{CoordPatt } a = \text{new CoordPatt(paterns.get}(cp))\ ^{4} \\
\mathcal{T}(decl_m(cp, v)) &\rightarrow \mathcal{T}(cp, v) \\
\mathcal{T}(assign_m(t, v, a, r, e_1, \ldots, e_n)) &\rightarrow \mathcal{T}(decl_m(t, v)); \mathcal{T}(@(r(a, e_1, \ldots, e_n))) \\
\mathcal{T}(assign_m(v, a, r, e_1, \ldots, e_n)) &\rightarrow \text{v} = \mathcal{T}(@(r(a, e_1, \ldots, e_n))) \\
\mathcal{T}(assign_m(a, r, e_1, \ldots, e_n)) &\rightarrow \mathcal{T}(@(r(a, e_1, \ldots, e_n)))
\end{aligned}
$$

Listing 1.11. Example of a translated ReCooPLa reconfiguration.

```
1  public class OverlapP extends Reconfiguration {
2      private CoordinationPattern p;
3      private LinkedHahsSet<Pair<Node, Node>> X;
4      public OverlapP(CoordinationPattern arg1,LinkedHahsSet<Pair<Node, Node>> arg2) {
5          this.p = arg1;
6          this.X = arg2;
7      }
8      public CoordinationPattern apply(CoordinationPattern $cp) {
9          Par par;
10         Join join;
11         par = new Par(this.p);
12         par.apply($cp);
13         for(Pair<Node> n : this.X) {
14             Node n1, n2;
15             n1 = n.getFst();
16             n2 = n.getSnd();
17             LinkedHahsSet<Node> E = new LinkedHahsSet<Node>() {{
18                 add(n1); add(n2);
19             }};
20             join = new Join(E);
21             join.apply($cp);
22         }
23         return $cp;
24     }
25 }
```

This translation schema is part of the ReCooPLa processor, implemented in ANTLR v3 [44]. The full processor counts on a parser for ensuring syntactic correctness, which outputs an Abstract Syntax Tree (AST) as an intermediate representation for use in subsequent processing stages like the creation of an identifiers table; a module for semantic analysis, where errors concerning structure, behaviour and data types are reported; and the translator itself.

The translator module implements the translation function $\mathcal{T}()$, by applying the attribute grammars formalism upon the AST, as a tree grammar walker of ANTLR v3; moreover, to simplify the whole process, template mechanisms were used, featuring the StringTemplate engine.

Listing 1.11 shows the result of applying the translation rules to the *OverlapP* reconfiguration encoded in ReCooPLa in Listing 1.8.

4.3. The ReCooPLa engine

The ReCooPLa Engine is implemented as an Eclipse plugin, as an editor extension. The system high-level architecture is presented in Fig. 9.

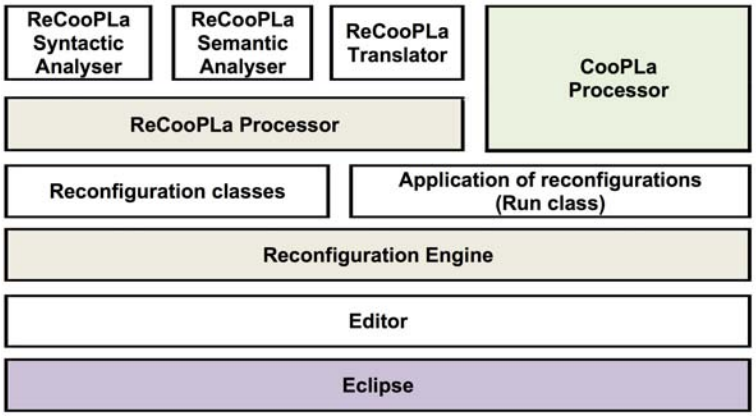


Fig. 9. The ReCooPLa engine high-level architecture.

Essentially, it wraps both the reconfiguration engine and the ReCooPLa processor (where the ReCooPLa translator belongs to) with a special connection to the CooPLa processor. As an Eclipse plugin, it takes advantage of patterns for features like syntax highlighting, code completion or error reporting and annotation. Additionally, views are associated to the ReCooPLa engine, that provide visual representation of the applied reconfigurations. The obtained coordination patterns can be saved as CooPLa files for further analysis with powerful external tools [18,21,29] that connect well with possible outputs of CooPLa [41].

The operation of the ReCooPLa engine, as a cyclic process, is sketched in Fig. 10. The process starts with the processing of CooPLa files by the CooPLa processor. It produces an internal representation of all coordination patterns present in the CooPLa files, which is used by the reconfiguration engine for validation, first, and reconfiguration, then.

The ReCooPLa processor takes the ReCooPLa files and (after syntactic and semantic analysis) uses the translator module to generate Java files into a temporary directory. Then, the ReCooPLa engine calls the external `javac` compiler in order to compile the java files into Java byte code. Using the reflection features associated to the JVM and the Java language, the created Reconfiguration classes are loaded into the running ReCooPLa engine; and so it is the main class `Run`.

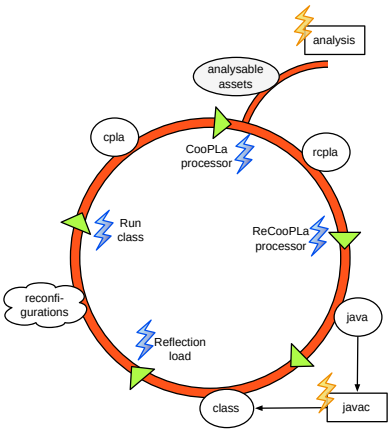


Fig. 10. ReCooPLa engine main process.

At this point, the main method within the `Run` class is called (via reflection, because it does not exist at the engine compilation time) so that reconfigurations are enacted and *deployed*. This step generates visualisations of the reconfigured coordination patterns and allows for saving the respective CooPLa files, closing the reconfiguration design loop.

Although out of the scope of this article, it makes sense to refer that the CooPLa processor is also responsible for producing assets for further analysis. This completes the design of reconfigurations, since analysis is required to check for problematic configurations which may trigger new iterations of the ReCooPLa process.

Fig. 11 is a print out of the engine interface dealing with the reconfigurations in Listings 1.8 and 1.10, along with the visualisation of the generated configuration.

5. Case Study

This section shows, through (a fragment of) a case study how the ReCooPLa engine can be used to assist the design of self-adaptive software systems, integrated in an approach that deals with both the design and runtime monitoring of this sort of systems, introduced by N. Oliveira and L. S. Barbosa in [39]. This approach builds on the reconfiguration framework presented in Section 2 and its supporting tools. It follows the feedback control loop MAPE(-K) model [23], integrating an active transition system of reconfigurations, referred to as the Reconfiguration Transition System (RTS).

An RTS is a transition system model where states define architectural configurations and transitions define the reconfiguration operations that transform a configuration into another. This model lays down reconfiguration strategies planed and prepared at design time by taking into account partial knowledge about relevant environment attributes.

At runtime, the RTS is integrated in the control loop. This loop is responsible for monitoring the system and the environment, acquiring data that is delivered to an analyser module. Such a module uses the data and a pool of possible system configurations from the current one (picked from the RTS). Resorting to suitable quantitative analysis tools, it analyses each possible configuration. A decision module verifies the results and by matching them with system properties and adaptation logic triggers, decides whether it is necessary to reconfigure the system and which reconfiguration shall be applied. In case of

Fig. 11. A printout of the engine interface on applying a reconfiguration.

need for adaptation, an executer module receives the decision previously made and enacts the reconfiguration at runtime.

More details on this approach can be found in [39], where a case study is reported to simulate the runtime adaptation of the Access Society s Knowledge (ASK) system. The case study presented in this section, on the other hand, is the twin, design-time component.

5.1. The ASK system

The ASK system is a communication software from Almende, a Dutch software house, whose objective is to mediate consumers and service providers. For instance, it may connect and mediate between a company looking for a temporary worker and a person matching such requirement. The system relies on powerful matching mechanisms that analyse the characteristics of both consumers requirements and providers profiles. Offering such a match effectively and within the minimum amount of time is the global goal.

The architecture of the ASK system is based on three main components: a web-based front-end for user interaction, a database for storing business data, and a contact engine for matching and contacting interveners. The contact engine collects the user requests, which are converted into tasks and shipped to an executor component responsible for the matching operation. In the latter tasks are enqueued into an execution-queue (EQ) until a handler web service (HRE) is prepared to perform the match between consumers and providers. The HRE service runs on a server separated from the EQ, but which is not dedicated; having the limit of spawning 20 HRE service instances in parallel. This constitutes the main part of the system, and the relevant one for the remaining of this section.

5.2. Adaptable-ASK design

The ASK system was previously studied with respect to performance and resource allocation from a static point of view in references [36,35,42]. These studies revealed bottlenecks and performance decay when the environment changed over time. By analysis of logs and monitored data, the ASK team identified a number of critical points (e.g. user requests reach peaks, downtime intervals and recurring environment fluctuations).

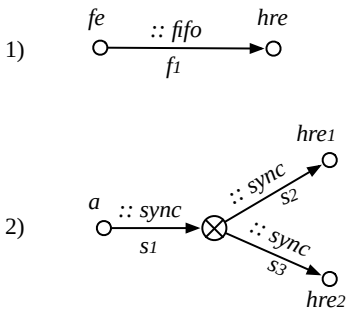


Fig. 12. Patterns for Adaptable-ASK.

This lead to the proposal, design and implementation of a system with an architecture able to adapt to contextual changes, referred to in the sequel as the Adaptable-ASK system.

In order to design the control loop, following the RTS-based approach briefly recalled above, the ReCooPLa engine was used. Our focus here is restricted to the executor component and its architectural adaptation, bypassing the conversion of user requests into tasks, which, in fact, does not insert any performance disturbance on the system.

The initial work was to design (in CooPLa) the basic coordination layer for this component. This is shown in Fig. 12.1. Note that communication between users and the HRE service is asynchronous with all requests enqueued into a FIFO-like structure. This coordination pattern, called `Original`, became then a building block for broader reconfiguration patterns.

By creating a stochastic instance of this pattern and using the CooPLa processor, analysable assets were generated for quantitative analysis. Such an exercise was done, taking user request rates as the relevant contextual change. When the number of requests increased, the basic architecture, with only one server available to host a limited number of instances of the HRE service, was shown unsuitable. This entailed the need for adding a second server to increase production. The coordination pattern in Fig. 12.2 (`ExRouter`) is another building block that can be used for adding a new server to the architecture[5], while rearranging the coordination between the relevant architectural elements. An instance of this coordination pattern can then be used to design the envisaged reconfiguration with two HRE servers. Fig. 13 shows how this is done with ReCooPLa (within the ReCooPLa engine).

```
import "buildingblocks.cpla";

reconfiguration overlapP(Pattern p; Set<Pair<Node>> x) {
    @ par (p);
    forall(Pair<Node> n : x) {
        Node n1, n2;
        n1 = n.fst;
        n2 = n.snd;

        Set<Node> E = S(n1, n2);
        @ join(E);
    }
}

main [Original sf; ExRouter sr] {
    ScaledOut sso = sf @ overlapP(sr, S(P(sf.out[0], sr.in[0])));
}
```

Fig. 13. ReCooPLa implementation of a reconfiguration to scale up the Adaptable-ASK system from the original configuration.

The resulting coordination pattern was saved in a file named `scaledout.cpla`. It was then processed with CooPLa tools for analysis with respect to user request fluctuations retrieved from the logs of the ASK system. This was shown to suit most of them, but for the case in which requests decrease and the company has still to pay the rent of the second server. Thus, in such a context it is necessary to go back to the original configuration. Fig. 14 shows a reconfiguration solution for both cases. This resulted in three exported coordination patterns. The second one was saved in a `scaledinout.cpla` file (the others already exist in CooPLa files).

By designing reconfigurations we end up in the construction of the RTS as requested (depicted in Fig. 15).

[5] Symbol ⊗ represents a node that routes data for one of its outgoing channels in a mutual-exclusive way.

```
import "buildingblocks.cpla";
import "scaledout.cpla";

reconfiguration removeP (Set<Name> Cs ) {
    forall ( Name n : Cs)
    {
        @ remove(n);
    }
}

reconfiguration insertP(Pattern p; Node n, mi, mo) {
    Pattern p1 = @ par(p);
    Pattern p2 = @ split(n);
    Set<Node> n1 = p2.in - p1.in ;
    Set<Node> n2 = p2.out - p1.out ;
    Set<Node> E1 = n1 + S(mi);
    @ join(E1);
    Set<Node> E2 = n2 + S(mo);
    @ join(E2);
}

main [Original so; ExRouter sr; ScaledOut sso] {
    ScaledOut sso_aux = sso @ id();
    ScaledOutIn ssoi = sso @ insertP(so, sso#f.out[0], so#f1.out[0], so#f1.in[0]);
    Original sonew = sso_aux @ removeP(sr.names); //names of the channels
}
```

Fig. 14. ReCooPLa implementation of three reconfigurations: the first is an auxiliary one to preserve the scaled out configurations; the other two act in opposite directions.

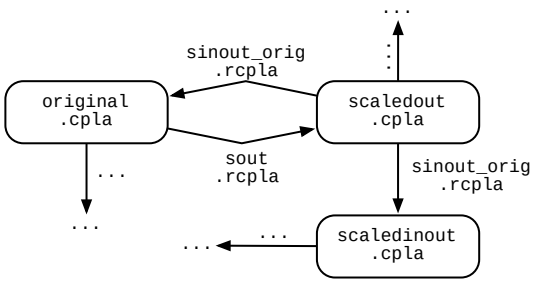


Fig. 15. Partial RTS for the design of the Adaptable-ASK system.

Certainly, the complete RTS for the Adaptable-ASK has a lot more states and transitions. For instance, a state representing a configuration where a log service is added, or another where a certain amount of requests is acceptable to be lost, do appear to cope with some deployment contexts. The elegance of this approach is that, most appropriately, the RTS may be changed at runtime with new or revised reconfigurations.

In summa, the role of the ReCooPLa engine in this case study was to lead the design of the necessary reconfigurations. But, as shown in Figure 10, the design of reconfigurations is just the start of a process. The engine allows for achieving new coordination structures by applying the designed reconfigurations upon coordination patterns. Each obtained pattern may be analysed/simulated to check whether it fulfils the requirements of the system for a given environment state.

6. Related work

As stated above, this paper aimed at introducing a DSL and an engine for the specification, compilation and application of architectural reconfigurations. Since the topic of architectural reconfiguration is a pressing issue in the SOA community, there is substantial related work to discuss. The following paragraphs provide a brief overview.

In general, Architecture Description Languages (ADLs) are widely accepted as the rigorous foundation for describing software architectures. While there are numerous representatives [2,3,12,19,30,31,48,50], they all take components, connectors, and their configurations as the main building blocks of an architecture. Their use has been mostly limited to static analysis and code generation, being generically unable to support architectural changes. Only a few ADLs (and their extensions) allow for conveying architectural modifications. For instance, Wright [3,5] with its dynamic extension [4] introduces reconfiguration actions that manipulate the topology of the architecture and are triggered by design-time known control events. ACME [19,20], through the Plastik extension [9], enables the specification of reconfigurations with specific constructors for attaching and detaching architectural elements as well as to establish dependencies between them. The run time triggering of Plastik s reconfigurations is based on conditional structures and event listening. Pilar [12] natively supports reconfigurations strongly relying on reflective computation concepts; reconfiguration triggers are specified as constraints and reconfigurations themselves are typical operations to manipulate the architecture topology.

These approaches to reconfiguration based on ADLs typically support run time changes. In the case of ReCooPLa and its engine, the focus is set on the static (design-time) analysis, which is rarely provided by such approaches. Moreover, ADLs reconfigurations target only the topology of the architecture and do not address the coordination behaviour as ReCooPLa intends to do.

While ADLs aim at describing software architectures for the purpose of system generation, Architectural Modification Languages (AMLs) focus on describing changes to architecture descriptions and are, thus, useful for introducing unplanned changes to deployed systems. The Extension Wizard s modification scripts [43], C2 s AML [33], and Clipper [1] are examples of such languages. These languages endow Software Architecture (SA) design approaches with mechanisms to specify reconfigurations. However, they focus on the high-level entities of the architectures, rather than on the coordination layer. In

contrast, ReCooPLa targets the whole coordination structure of a system and is mainly designed towards reconfiguration analysis.

Fractal [10] is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems, which embodies mechanisms for dynamic reconfiguration. Separate domain specific languages, FPath and FScript [13], are used to apply changes. The former eases the navigation inside a Fractal architecture through queries. The latter, which embeds FPath, enables the definition of adaptation scripts to modify the architecture of a Fractal application. It includes transactional support for architectural reconfigurations in order to ensure the reliability and consistency of the application. This approach targets industry-level applications by resorting to standards for architectural definition like Service Component Architecture (SCA), although strongly biased towards Fractal-based applications.

Plump [45] introduces the GP programming language. It is a language for solving graph problems, based on a notion of graph transformation and four operators, and shown to be Turing-complete. GP allows for the creation of programs over graphs at a high-level of abstraction. It was not designed for specifying architectures nor reconfiguration strategies. However, its applications are numerous and architectural design could be one of them. In fact, rule-based definition of graph transformations are also used in some foundational works on architectural reconfiguration [11,51]. ReCooPLa, which also acts over graph-based structures, replaces the use of rules by a set of (coordination-oriented) constructors.

Oreizy et al. [43] proposed ArchStudio, a tool suite that implements several interrelated mechanisms for supporting the runtime reconfiguration of software architectures. It is implemented in Java and targets C2-style applications, which limit its applicability to Service-Oriented Architecture (SOA) design.

Krause [27,28] formalises a framework to specify and apply reconfigurations in the context of the Reo coordination model [7]. Reconfigurations, therein, are specified through graph transformation rules upon a graph-like structure with typed-edges, representing the coordination structure of a system. This was latter adapted to deal with distributed networks of software connectors [25] and with data flow aware reconfigurations [24]. The materialisation of this work is made on the Extensible Coordination Tools (ECT) Eclipse plugin. In this tool, Reo circuits are visually specified and reconfigurations are defined directly upon the circuits as pattern-matching rules for their transformation. Reconfiguration rules can be applied either to a local connector or to the global coordination structure. Simulation of the application of such rules is possible, but a centralised engine (ReoLive) for dynamic reconfigurations is also provided. The work of Krause is, to the best of our knowledge, the most recent on on coordination-based reconfigurations with suitable tool support. ReCooPLa provides a more generic way of expressing reconfigurations, but as it is currently limited to design-time application, comparison is not meaningful.

7. Conclusions and future work

This paper introduces the ReCooPLa engine, an Eclipse plugin for the design of coordination-based reconfigurations. It incorporates the ReCooPLa processor, its companion DSL, and an engine that applies reconfigurations expressed in ReCooPLa to coordination structures expressed in CooPLa. In this graph-based framework, reconfigurations act,

through the application of primitive atomic operations, over a graph structure, which is an abstract representation of the coordination layer of a SOA system.

The ReCooPLa engine resorts to generative programming for processing and translation of ReCooPLa specifications. It takes advantage of reflection features, associated to the Java programming language, to compile and dynamically load the generated Java files.

A distinctive characteristic of ReCooPLa, with respect to other architectural reconfiguration tools, is its focus on the coordination layer rather than on the high-level architecture. In the same way, ReCooPLa, differs from other architectural languages by focussing on reconfigurations rather than on the definition of architectural elements like components, connectors and their interconnections. At the moment of writing, however, ReCooPLa engine is targeted to the early stages of software development; *i.e.*, the design of reconfigurations and their analysis against requirements. Nevertheless, as shown in the case study, the engine may play an important role in the design of adaptive systems.

Planned for future work is the import of architectural configurations specified in Wright, ACME, or a similar ADLs, whenever their connectors are suitable instances keeping a full specification of a system architecture. Therefore, ReCooPLa can be part of a control loop for adaptive systems.

Acknowledgments. This work is partly funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT, the Portuguese Foundation for Science and Technology, within project FCOMP-01-0124-FEDER-028923. The second author is supported by an Individual Doctoral Grant from FCT, with reference SFRH/BD/71475/2010.

References

1. Agnew, B., Hofmeister, C., Purtilo, J.: Planning for change: a reconfiguration language for distributed systems. In: Proceedings of 2nd International Workshop on Configurable Distributed Systems, 1994. pp. 15–22 (1994)
2. Aldrich, J., Chambers, C., Notkin, D.: Architectural reasoning in ArchJava. In: Magnusson, B. (ed.) Object-Oriented Programming (ECOOP 2002), Lecture Notes in Computer Science, vol. 2374, pp. 334–367. Springer, Berlin Heidelberg (2002)
3. Allen, R.: A Formal Approach to Software Architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science (Jan 1997)
4. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. Configurations 1382, 1–15 (1998)
5. Allen, R., Garlan, D.: Formalizing architectural connection. In: Proceedings of the 16th International Conference on Software Engineering. pp. 71–80. ICSE 1994, IEEE Computer Society Press, Los Alamitos, CA, USA (1994)
6. Andrews, G.R.: Paradigms for process interaction in distributed programs. ACM Computing Surveys 23(1), 49–90 (Mar 1991)
7. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (Jun 2004)
8. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous Component-Based system design using the BIP framework. IEEE Software 28(3), 41–48 (May 2011)
9. Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in Component-Based systems. In: Morrison, R., Oquendo, F. (eds.) Software Architecture, Lecture Notes in Computer Science, vol. 3527, chap. 1, pp. 1–17. Springer, Berlin, Heidelberg (2005)

10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. Software: Practice and Experience 36(11-12), 1257–1284 (Sep 2006)
11. Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lafuente, A.L.: Graph-Based design and analysis of dynamic software architectures concurrency, graphs and models. In: Degano, P., Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models, Lecture Notes in Computer Science, vol. 5065, chap. 4, pp. 37–56. Springer, Berlin, Heidelberg (2008)
12. Cuesta, C.E., de la Fuente, P., Barrio-Solórzano, M., Beato, E.: Coordination in a reflective architecture description language. In: Arbab, F., Talcott, C. (eds.) Coordination Models and Languages, Lecture Notes in Computer Science, vol. 2315, pp. 141–148. Springer, Berlin, Heidelberg (2002)
13. David, P.C., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of fractal architectures. Annals of Telecommunications 64(1-2), 45–63 (Feb 2009)
14. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An annotated bibliography. SIGPLAN Notices 35(6), 26–36 (Jun 2000)
15. Erl, T.: SOA Design Patterns. Prentice Hall PTR, NJ, USA, 1st edn. (2009)
16. Falbo, R., Guizzardi, G., Duarte, K.: An ontological approach to domain engineering. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering. pp. 351–358. SEKE 02, ACM, New York, NY, USA (2002)
17. Fiadeiro, J.L., Lopes, A.: A model for dynamic reconfiguration in service-oriented architectures. Software and Systems Modeling 12(2), 349–367 (Feb 2013)
18. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer pp. 1–19 (2012)
19. Garlan, D., Monroe, R., Wile, D.: ACME: An architecture description interchange language. In: Proceedings of the CASCON 97. pp. 7–. IBM Press (1997)
20. Garlan, D., Monroe, R.T., Wile, D.: ACME: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
21. Guck, D., Han, T., Katoen, J.P., Neuhäußer, M.R.: Quantitative timed analysis of interactive markov chains. In: Goodloe, A.E., Person, S. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 7226, pp. 8–23. Springer, Berlin, Heidelberg (2012)
22. Hnetynka, P., Plásil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) Component-Based Software Engineering, Lecture Notes in Computer Science, vol. 4063, chap. 27, pp. 352–359. Springer, Berlin, Heidelberg (2006)
23. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (Jan 2003)
24. Koehler, C., Arbab, F., de Vink, E.: Reconfiguring distributed reo connectors. In: Corradini, A., Montanari, U. (eds.) Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, vol. 5486, pp. 221–235. Springer, Berlin, Heidelberg (2009)
25. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. In: Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques. GT-VMT 08. Electronic Communications of the EASST, vol. 10, pp. 1–13 (2008)
26. Kosar, T., Oliveira, N., Mernik, M., Pereira, M.J.V., Crepinsek, M., da Cruz, D., Henriques, P.R.: Comparing General-Purpose and Domain-Specific Languages: An empirical study. Computer Science and Information Systems 7(2), 247–264 (May 2010)
27. Krause, C.: Reconfigurable Component Connectors. Ph.D. thesis, Leiden University, Amsterdam, The Netherlands (2011)

28. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Science of Computer Programming 76(1), 23–36 (2011)
29. Kwiatkowska, M., Norman, G., Parker, D.: A framework for verification of software with time and probabilities. In: Chatterjee, K., Henzinger, T.A. (eds.) Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems. FORMATS 10. Lecture Notes in Computer Science, vol. 6246, pp. 25–45. Springer, Berlin, Heidelberg (2010)
30. Luckham, D.C., Vera, J.: An Event-Based Architecture Definition Language. IEEE Trans. Softw. Eng. 21(9), 717–734 (Sep 1995)
31. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering. SIGSOFT 1996, vol. 21, pp. 3–14. ACM, New York, NY, USA (Nov 1996)
32. Malohlava, M., Bures, T.: Language for reconfiguring runtime infrastructure of component-based systems. In: Proceedings of the Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. MEMICS 2008, Znojmo, Czech Republic (Nov 2008)
33. Medvidovic, N.: ADLs and dynamic architecture changes. In: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints 96) on SIGSOFT 96 workshops. pp. 24–27. ISAW 1996, ACM, New York, NY, USA (1996)
34. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (December 2005)
35. Moon, Y.J.: Stochastic Models for Quality of Service of Component Connectors. Ph.D. thesis, Universiteit Leiden (Oct 2011)
36. Moon, Y.J., Arbab, F., Silva, A., Stam, A., Verhoef, C.: Stochastic Reo: a case study. In: Proceedings of the TTSS 11 (2011), to appear.
37. Oliveira, N., Barbosa, L.S.: On the reconfiguration of software connectors. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC 13, vol. 2, pp. 1885–1892. ACM, New York, NY, USA (Mar 2013)
38. Oliveira, N., Barbosa, L.S.: Reconfiguration mechanisms for service coordination. In: ter Beek, M.H., Lohmann, N. (eds.) Web Services and Formal Methods, Lecture Notes in Computer Science, vol. 7843, pp. 134–149. Springer, Berlin, Heidelberg (2013)
39. Oliveira, N., Barbosa, L.S.: A self-adaptation strategy for service-based architectures. In: Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse. SBCARS 2014, vol. 2, pp. 1–10. SBC – Brazilian Computer Society, Piscataway, NJ, USA (Sep 2014)
40. Oliveira, N., Pereira, M.J.V., Henriques, P.R., da Cruz, D.: Domain-Specific Languages: a theoretical survey. In: Proceedings of INForum 09. pp. 35–46. Lisbon, Portugal (September 2009)
41. Oliveira, N., Silva, A., Barbosa, L.S.: Quantitative analysis of Reo-based service coordination. In: Proceedings of SAC 14. vol. 2, pp. 1247–1254. ACM, NY, USA (March 2014)
42. Oliveira, N., Silva, A., Barbosa, L.S.: $\mathsf{IMC}_{\mathsf{Reo}}$: interactive Markov chains for stochastic Reo. Journal of Internet Services and Information Security 5(1) (February 2015), imprint
43. Oreizy, P., Taylor, R.N.: On the role of software architectures in runtime system reconfiguration. In: Proceedings of the 4th International Conference on Configurable Distributed Systems. pp. 61–70. IEEE (May 1998)
44. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Raleigh (2007)
45. Plump, D.: The graph programming language GP. In: Bozapalidis, S., Rahonis, G. (eds.) Algebraic Informatics, Lecture Notes in Computer Science, vol. 5725, chap. 6, pp. 99–122. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
46. Ramirez, A.J., Cheng, B.H.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 49–58. SEAMS 2010, ACM, New York, NY, USA (2010)

47. Rodrigues, F., Oliveira, N., Barbosa, L.S.: ReCooPLa: a DSL for coordination-based reconfiguration of software architectures. In: Pereira, M.J.V., Leal, J.P., Simões, A. (eds.) Proceedings of SLATE 2014. OpenAccess Series in Informatics (OASIcs), vol. 38, pp. 61–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014)
48. Sanchez, A., Barbosa, L.S., Riesco, D.: Bigraphical modelling of architectural patterns. In: Arbab, F., Ölveczky, P.C. (eds.) Formal Aspects of Component Software. FACS 2011, Lecture Notes in Computer Science, vol. 7253, pp. 313–330. Springer, Berlin, Heidelberg (2011)
49. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience 42(5), 559–583 (2011)
50. Tracz, W.: Parametrized programming in LILEANNA. In: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice. pp. 77–86. SAC 93, ACM, New York, NY, USA (1993)
51. Wermelinger, M.A., Fiadeiro, J.L.: Algebraic software architecture reconfiguration. In: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 393–409. ESEC/FSE 1999, Springer-Verlag, London, UK (1999)

Flávio Rodrigues is currently a back-end developer in a Software House. Previously, he obtained a degree in Informatics from Instituto Politécnico do Cávado e do Ave, and received a Master s Degree in Informatics Engineering from Universidade do Minho, with specialisation in Language Engineering and Applications Engineering.

Nuno Oliveira is a lecturer at Instituto Politécnico do Cávado e Ave and a software engineer at Checkmarx. He holds a PhD in computer science, from Universidade do Minho, for his work on architectural reconfigurations. Previously, he obtained a degree and a MSc in Computer Science also from Universidade do Minho.

Lu´s S. Barbosa is an Associate Professor at the Informatics Department of Universidade do Minho, Braga, Portugal, and a senior researcher at INESC TEC in high-assurance software and formal methods. He coordinates the joint Doctoral Programme in Computer Science of Universidades do Minho, Aveiro and Porto.

Received: September 12, 2014; Accepted: May 21, 2015.