ReCooPLa: a DSL for Coordination-based Reconfiguration of Software Architectures

Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa

HASLab – INESC TEC & Universidade do Minho
Braga, Portugal
pg22826@alunos.uminho.pt, {nunooliveira,lsb}@di.uminho.pt

Abstract

In production environments where change is the rule rather than the exception, adaptation of software plays an important role. Such adaptations presuppose dynamic reconfiguration of the system architecture, however, it is in the static setting (design-phase) that such reconfigurations must be designed and analysed, to preclude erroneous evolutions. Modern software systems, which are built from the coordinated composition of loosely-coupled software components, are naturally adaptable; and coordination specification is, usually, the main reference point to inserting changes in these systems.

In this paper, a domain-specific language—referred to as ReCooPLa—is proposed to design reconfigurations that change the coordination structures, so that they are analysed before being applied in run time. Moreover, a reconfiguration engine is introduced, that takes conveniently translated ReCooPLa specifications and applies them to coordination structures.

1998 ACM Subject Classification D.2.11 Software Architectures, F.3.2 Semantics of Programming Languages

Keywords and phrases domain-specific languages, architectural reconfiguration, coordination

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.61

1 Introduction

For the last few years, Service-Oriented Architecture (SOA) has been adopted as the architectural style to support the needs of modern intensive software systems [9]. SOA systems are based on services, which are distributed, loosely-coupled entities that offer a specific computational functionality via published interfaces. Within SOA, services are coordinated, so that the ensemble delivers the system required functionality. Coordination is the design-time definition of a system behaviour. It establishes interactions between software building blocks (services, in SOA systems), including their communication constraints and policies. Such policies may be encapsulated in a multitude of ways [3], but point-to-point communication approaches (*e.g.*, channels [4]), gain relevance by fomenting the desired decoupling between computation and coordination concerns. This separation of concerns makes SOAs flexible, easier to analyse and naturally dynamic. Although policies are pre-established, services with similar interface may be discovered and bound to the architecture at run time, rather than fixed at design time.

Flexibility and dynamism are desired features in production environments where change is the rule rather than the exception. Constant environment evolution brings new requirements to the system, may contribute to degradation of contracted Quality of Service (QoS) values, or introduce failure [24, 28]. These changes raise the need for systems to adapt to new contexts while running.

Reconfigurations upon SOA systems usually target the manipulation of services: dynamic update of service functionality, substitution of services with compatible interfaces (but not necessarily the same behaviour) or removal of services [26, 23, 11]. However, in some situations, this may not be enough. For instance, when a substituting service has incompatible interface, it may be necessary to target, with further detail, the way services interact with each other. This sort of reconfiguration goes into the coordination layer and usually substitute, add or remove interaction components (*e.g.*, *communication channels*), move communication interfaces between components and may even rearrange a complex interaction structure [13, 14]. Thus, there is a mismatch between project needs and what is currently offered in practice. More worryingly it is the lack of rigorous (formal) methods to correctly design and analyse this sort of reconfigurations.

In the authors' previous work [19, 20], a formal framework for modelling and analysing coordination-based reconfigurations in the context of SOA was defined. In this framework, a coordination structure (referred to as a *coordination pattern*) is regarded as a graph whose nodes reprsent interaction points (with either services or other coordination patterns), and edges are communication channels with a specific behaviour. However, this framework lacks mechanisms to express and apply reconfigurations, in practice. Such is the purpose of this paper: to introduce a Domain-specific Language (DSL), referred to as ReCooPLa, to express coordination-based reconfigurations, materialising the formal model presented in [19] and briefly discussed in further sections.

DSLs [27, 18, 21] are languages focused on particular application domains and building on specific domain knowledge. Their level of abstraction is tailored to the specific domain, allowing for embedding high-level doamin concepts in the language constructs, and hiding low-level details under their processors. In addition, they allow for validation and optimisation at the domain level, offering considerable gains in expressiveness and ease of use, compared with General-purpose Programming Languages (GPLs) [12].

In this spirit, ReCooPLa is a simple and small language that provides a precise, high-level interface for reconfiguration designers. The reconfiguration construct plays, then, a main role in ReCooPLa. It resembles functions, as in GPLs, with a header and a body. The header defines the reconfiguration identifier and its arguments; the body is composed of instructions, where coordination-specific notions are embodied in constructs that manipulate the graph structure which underlies coordination patterns.

A suitable reconfiguration engine, for application of the reconfigurations expressed in ReCooPLa is also proposed in this paper. It is regarded as a *machine* that executes reconfigurations over the target coordination patterns. To this end, a translation of ReCooPLa constructs into the engine's running code is carefully defined.

Outline. Related work is presented in Section 2 and background notions are introduced in Section 3. In Section 4 the ReCooPLa language is introduced with a detailed way and illustrated by small examples. Then, Section 5 introduces the reconfiguration engine along with a suitable translation of ReCooPLa constructs into it. Section 6 discusses an example. Finally, Section 7 concludes and proposes some topics for future work.

2 Related Work

Domain-specific languages constitute an important tool to tackle the specificities of particular application domains. The design of reconfigurations in the context of SOA is the domain underlying this work. Typical design approaches to reconfigurability in software architecture and component based design are discussed in this section.

Fractal [6] is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems, which embodies mechanisms for component composition and dynamic reconfiguration. It counts on FPath and FScript [8], which are DSLs, to securely apply changes. The former eases the navigation inside a Fractal architecture trhough queries. The latter, which embeds FPath, enables the definition of adaptation scripts to modify the architecture of a Fractal application, with transactional support.

Reference [7] proposes Architectural Design Rewriting (ADR) as a declarative rule-based approach for modeling reconfigurable Software Architectures (SAs). It is based on an algebraic presentation of graph structures and conditional rewrite rules, suitable to model. hierarchical designs, and inductively defined reconfigurations.

In general, Architecture Description Languages (ADLs) provide a rigorous foundation for describing SAs, specifying syntax and semantics to describe components, connectors, and their configurations. Numerous ADLs have been developed, each providing complementary capabilities for architectural development and analysis. Their use has been limited to static analysis and generation focused on static issues and, therefore, unable to support architectural changes. However, a few ADLs, such as Darwin [16], Rapide [15], Wright [2] and Acme [10] can express run time architectural provided they have been previously specified.

While ADLs aim at describing SAs for the purposes of analysis and 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, C2's AML [17], and Clipper [1] are examples of such languages. Similarly, Architectural Constraint Languages (ACLs) have been used to restrict the system structure using imperative [5] as well as declarative [16] specifications.

These languages endow SA design approaches with mechanisms to specify reconfigurations. However, the latter focus on the high-level entities of architectures, rather than on the coordination glue code. A ReCooPLa, in contrast, is targets the whole coordination pattern of a system and is oriented towards reconfiguration analysis.

Also related to ReCooPLa is the GP programming language presented in [25]. It is a language for solving graph problems, based on a notion of graph transformation and four operators shown to be Turing-complete. Like GP, ReCooPLa actuates over a graph-based structure to perform modifications. While GP does so with program rules, ReCooPLa defines reconfiguration methods based on primitive (coordination-oriented) constructs.

3 Reconfiguration Model

This section provides an informal account of the reconfiguration model, which has been introduced and formalised in [19, 20]. In particular, it introduces the notions of a coordination pattern and coordination-based reconfiguration, which are later embodied in the constructs of ReCooPLa.

3.1 Coordination Protocols

A coordination protocol works as a *glue code* to define and constrain the interaction between components or services of a system. In this model, it is called a *coordination pattern* and regarded as a reusable and composable architectural element. It is formalised as a graph of channels whose nodes are interaction points through which it can plug to other coordination patterns or services; edges are uniquely identified point-to-point communication devices with a specific behaviour given by a channel typing system. Formally,

$$\rho \subseteq \mathcal{N} \times \mathcal{I} \times \mathcal{T} \times \mathcal{N},$$

where $\mathcal{N}$ is a set of nodes (to be precise, a node in a coordination pattern corresponds to a set of channel ends), $\mathcal{I}$ is a set of channel identifiers and $\mathcal{T}$ is a channel typing system. Set $\mathcal{T}$={sync, lossy, fifo, drain} is adopted in the sequel as the working channel typing system in the spirit of the Reo coordination language [4]. Nodes that are used exclusively for data input (respectively, output) constitute the input (respectively, output) ports of the pattern. All the others are classified as internal or mixed nodes.

Listing 1 presents two coordination patterns. Coordination pattern `cp1` comprises two channels: a channel `x1` of type sync, and channel `x2` of type lossy. Channel `x1` has an input node `a` and an output node `b.c`[1]. In turn, channel `x2` has an input node `b.c` (corresponding to output node of channel `x1`, once they are connected), and an output node `d`.

Listing 1 Two simple coordination patterns.

```
cp1: {(a, x1, sync, b.c), (b.c, x2, lossy, d)}
cp2: {(g, x3, sync, h.i.j), (h.i.j, x4, lossy, k),
      (h.i.j, x5, fifo, l)}
```

3.2 Coordination-based Reconfigurations

A reconfiguration is a modification of the original structure of a coordination pattern obtained through sequential or parallel application of parametrised elementary operations, which are called reconfiguration *primitives*.

Let ρ be a coordination pattern.The simplest reconfigurations are the identity (`id`) and the constant (`const(`ρ`)`) primitives. The former returns the original coordination pattern, while the latter replaces it with ρ.

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

Listing 2 Resulting coordination pattern after applying the `par` primitive.

```
cp1: { (a, x1, sync, b.c), (b.c, x2, lossy, d), (g, x3, sync, h.i.j),
       (h.i.j, x4, lossy, k), (h.i.j, x5, fifo, l)}
```

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(a,g)` to `cp1` (*c.f.*, Listing 2) creates a connection on node `a.g`, as presented in Listing 3.

Listing 3 Resulting coordination pattern after applying the `join` primitive.

```
cp1: {(a.g, x1, sync, b.c), (b.c, x2, lossy, d),
(a.g, x3, sync, h.i.j), (h.i.j, x4, fifo, k), (h.i.j, x5, drain, l)}
```

The `split(`n`)` primitive, where n is a node, is dual to the `join` combinator because it breaks connections within a coordination pattern by separating all channel ends coincident

[1] Notation `b.c` is used to express the node {`b,c`}, where `b` and `c` are channel ends.

in n. Listing 4 presents the resulting coordination pattern, after applying `split(h.i.j)` to `cp1` from Listing 3. Notice that the ends composing node `h.i.j` are assigned to each channel that previously shared this node (viz. channels `x3`, `x4` and `x5`), in a non-deterministic way.

Listing 4 Resulting coordination pattern after applying the `split` primitive.

```
cp1: {(a.g, x1, sync, b.c), (b.c, x2, lossy, d), (a.g, x3, sync, h),
(i, x4, fifo, k), (j, x5, drain, l)}
```

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), these connections are also broken as it happens with `split`. Listing 5 presents the resulting coordination pattern, after applying `remove(x2)` to `cp1` from Listing 4. Notice how node `b.c` was split and its end `c` was removed along with channel `x2`. Again, this process is non-deterministic.

Listing 5 Resulting coordination pattern after applying the `remove` primitive.

```
cp1: { (a.g, x1, sync, b), (a.g, x3, sync, h), (i, x4, fifo, k),
       (j, x5, drain, l)}
```

These primitive operations are assumed to be applied in sequence. Their parallel application is also valid, but only when they can be shown to be mutually independent: *i.e.*, affecting separated substructures of the target coordination pattern. This possibility of composing primitive operations in sequence or parallel, allows for the definition of complex reconfigurations, referred to as *reconfiguration patterns*. Actually, they affect significant parts of a coordination pattern at a time, and are expected to be generic, parametric and reusable. The ReCooPLa language offers a way of specifying such combinations in an imperative-like style.

4 ReCooPLa: Reconfiguration Language

ReCooPLa is a language for designing coordination-based reconfigurations. As a DSLs tailored to the area of architectural reconfigurations, it makes possible to abstract away from specific details, such as the effect of each primitive operation and their actual application (whether in sequence or in parallel), as well as to hide their actual computation under a processor.

4.1 Overview

In ReCooPLa, a *reconfiguration* is a first class citizen, as much as functions are in some programming languages. In fact, these two concepts share characteristics: both have a signature (identifier and arguments) and a body which designates 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*, *Set*, *Pair*, *Triple*, *Pattern* and *Channel*.

The reconfiguration body is a list of different sorts of instructions. The main one concerns *application* of (primitive, or previously defined) reconfigurations, since this is the only way of modifying a coordination pattern. As auxiliar operations, ReCooPLa resorts to other constructs that mainly manipulate the parameters of a reconfiguration. In particular, they provide ways to declare, assign and manipulate local variables, for example, field selectors, the usual set connectives (union, intersection and subtraction), and an iterative control structure to iterate over the elements of a set.

In brief, ReCooPLa is a small language borrowing most of its constructs from imperative programming languages. Actually, reconfigurations are better expressed in a procedural/algorithmic way, which justifies the choice of an imperative style.

4.2 The Language

In the sequel, we introduce ReCooPLa by presenting (the most important) fragments of the underlying grammar. A number of constructs are defined for further reference in the paper. Formally, a sentence in ReCooPLa specifies one or more reconfigurations.

Reconfiguation

A reconfiguration (see Listing 6) is expressed similarly to a function. The header is composed of a reserved word `reconfiguration` followed by a unique identifier (the reconfiguration name) and a list of arguments, which may be empty. The body is a list of instructions as explained below. Arguments are aggregated by data type, differently from what happens in conventional languages where data types are replicated for every different argument.

Listing 6 Extended Backus–Naur Form (EBNF) notation for the *reconfiguration* production.

```
reconfiguration
        : 'reconfiguration' ID '(' args* ')' '{' instruction+ '}'
args    : arg (';' arg)*
arg     : datatype ID (',' ID)*
```

The constructor for a reconfiguration is given by: $rcfg(n, t_1, a_1, \ldots, t_n, a_n, b)$, where n is the name of the reconfiguration; each a_i is an argument of type t_i; and b is the body of the reconfiguration.

Data types

ReCooPLa builds on a small set of data types: primitive (*Name* and *Node*), 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 seen as a set of names, to maintain compatibility with its definition in Section 3. The generic data types (based on the Java generics) specify a type by its contents, as seen in Listing 7.

Listing 7 EBNF notation for the *datatype* production.

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

Structured data types have an internal state, matching their definition in Section 3. Each instance of these types is endowed with attributes and operations, which can be accessed using selectors (later in this section).

The construct 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 constructs 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. In Listing 8 it can be seen the corresponding production rule.

Listing 8 EBNF notation for the *forall* production.

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

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

The application of a reconfiguration, (*c.f.*, `reconfiguration_apply` production in Listing 9), is expressed by an identifier followed by the `'@'` operator and a reconfiguration name. The latter may be a primitive reconfiguration or any other 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 called is applied to the original pattern. This typical usage can be seen in Listing 13

Listing 9 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 called either as $@(c)$ or $@(p, c)$, where p is a *Pattern* and c a reconfiguration call. Each reconfiguration call also has its own constructor: $r(a_1, \ldots, a_n)$, 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. Listing 10 shows examples of these types of operation. Each constructor is defined as a reserved word (`S` stands for *Set*, `P` for *Pair*, `T` for *Triple* and `N` for *Node*); and a list of values which is expected to comply to the data type involved. The corresponding production rule is given in Listing 10 and exemplified in Listing 11.

Listing 10 EBNF notation for the constructor production.

```
constructor
        : 'P' '(' expression ',' expression ')'
        | 'T' '(' expression ',' expression ',' expression ')'
        | 'S' '(' ( expression (',' expression)*)? ')'
        | 'N' '(' ID (',' ID)* ')'
```

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* and *Name*), selectors are used to apply the operation, as shown in Listing 12 (production rule `operation`). Symbol # is used to access a specific channel from the internal structure of a pattern.

Listing 11 *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);
    Node d = N(e1, e2);
```

Listing 12 EBNF notation for the `operation` and `attribute_call` productions.

```
operation
        : ID ('#' ID)? '.' attribute_call
attribute_call
        : 'in' ( '(' INT ')' )?
        | 'out' ( '(' INT ')' )?
        | 'ends' '(' ID ')'
        | 'name' | 'nodes' | 'names' | 'channels'
        | '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 12 and described below:

- `in`: returns the input ports from the *Pattern* and *Channel* variables. It is possible to obtain a specific port refered by an optional integer parameter indexing a specific entry from the set (seen as an array).
- `out`: returns the output ports from the *Pattern* and *Channel* variables. The optional parameter can be used as explained for the `in` *attribute call*.
- `name`: returns the name of a *Channel* variable, i.e., a channel identifier.
- `ends`: returns the ends of a *Channel* variable in the context of a *Pattern* given as parameter.
- `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.
- `channels`: returns a set of channels associated to a *Pattern* variable.
- `fst`, `snd`, `trd`: act, respectively, as the first, second and third projection from a tuple (*Pair* and *Triple* variables).

All these operations give rise to their own language constructors. For example, the constructor of a *Pair* data type is $P(e_1, e_2)$, where e_1, e_2 are expressions; for field selection $.(v, c)$ is used, where v is a variable and c a call to an operation; for set union we write $+(s_1, s_2)$, with s_1, s_2 variables of type *Set*. The remaining constructors are defined similarly.

Listing 13 shows an example of valid ReCooPLa sentences which declare two reconfigurations: `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.

5 ReCooPLa: Language Compilation

This section introduces the reconfiguration engine, which executes reconfigurations specified in ReCooPLa, and the correspnding translation schema into Java code.

Listing 13 ReCooPLa input 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);
    }
}
```

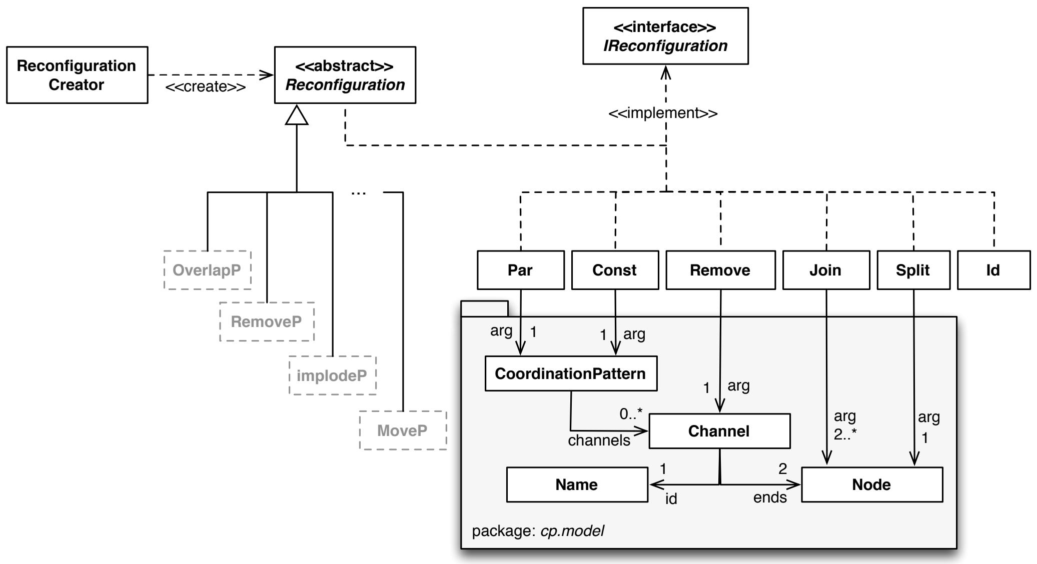

Figure 1 The Reconfiguration Engine model.

5.1 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 [20], a lightweight language to define the graph-like structure of coordination patterns. The model of the engine is as simple as it can be, taking into account only a few entities. Figure 1 presents the corresponding Unified Modelling Language (UML) class diagram.

Package *cp.model*, represented as a shaded diagram, concerns the model of a coordination pattern. This is actually, the implementation of the formal model presented in Section 3. 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 of the diagram are concerned with reconfigurations themselves, and 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 also introduced in Section 3. The relationships with the elements of the *cp.model* package define their arguments. Moreover, these classes have a common implicit method (given by the interface IReconfiguration): apply(CoordinationPattern p), where the behaviour of these primitives is defined as the combined effect of their application to the coordination pattern p given as an argument.

The Reconfiguration class represents a generic reconfiguration that requires its concrete classes to implement the apply(CoordinationPattern p) method. The careful reader may have noticed that the concrete classes of Reconfiguration 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 at design time. All of them are created dynamically, at run time, by the ReconfigurationCreator class, taking advantage of reflection in Java Virtual Machine (JVM) and working packages like *Javassist*[2]. This implementation follows a similar approach to the well-known Factory design pattern, but instead of creating instances, it creates concrete classes of Reconfiguration. The idea is that each reconfiguration definition within a ReCooPLa specification gives rise to a newly created class with an apply(CoordinationPattern p) method. Then, the content of such method is derived from the content of the ReCooPLa reconfiguration and added dynamically, via reflection, to the created class. Once the classes are loaded into the running JVM, the application of reconfigurations becomes as simple as calling the apply method from instances of such classes.

However, for this to be possible, it is first necessary to correctly translate ReCooPLa constructs into the code accepted by the Reconfiguration Engine. Section 5.2 goes through the details of such a translation.

The application of reconfigurations is also specified in ReCooPLa, taking into consideration the coordination patterns defined in CooPLa (which may be imported to ReCooPLa, a detail omitted in this paper). A script-based structure is assumed to define how reconfigurations are concretely applied to coordination patterns. A glimpse of how this can be achieved is unveiled in Listing 14.

Listing 14 Sketch of a reconfiguration script.

```
import "patterns.cpl", "reconfigurations.rcpl"
reconfigure (UserUpdate sq1)
    UserUpdate sq2 ;
    sq1 @ OverlapP(sq2, S(P(sq1#f2.out[0],sq2#s1.in[0])));
```

Its meaning is straightforward. First, the necessary definitions (reconfigurations and patterns) are imported. Then, the `reconfigure` reserved word marks the beginning of the reconfiguration script. Parameter `UserUpdate sq1` defines a *UserUpdate* coordination pattern (*c.f.* Figure 2) in some configuration. This is not limited to one pattern and in the future it may be a pointer to some ADL specification, where coordination patterns play the role of connectors. The declaration `UserUpdate sq2` defines a fresh instance of this coordination pattern. Finally, an OverlapP reconfiguration is applied on `sq1` with appropriated arguments.

[2] `http://www.javassist.org`

5.2 ReCooPLa Translation

Throughout this subsection, it is assumed the existence of Java classes to match the types in ReCooPLa. This means that, besides the classes already mentioned in Figure 1, the following ones are also assumed: 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 abbreviated to LHSet for increased readability. Moreover, keep exposition simple, details about reflection will be ignored or abstracted. For instance, method mkClass(cl, t_1, a_1, ..., t_n, a_n, b) abstracts the dynamic creation of a Reconfiguration class with name cl; attributes $a_1, \ldots, a_n$ of type $t_1, \ldots, t\_n$, respectively; and method apply with body b, which always ends with a return p instruction, where p is the argument of apply.

This 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 4 and defined as shown in Table 1.

Table 1 Translation rules for ReCooPLa constructs.[3]

$\mathcal{T}(rcfg(n, t_1, a_1, ...t_n, a_n, b))$	$\rightarrow$	mkClass(n, $\mathcal{T}(t_1)$, a_1, ... $\mathcal{T}(t_n)$, a_n, $\mathcal{T}(b)$)
$\mathcal{T}(T())$	$\rightarrow$	T
$\mathcal{T}(T_G(t))$	$\rightarrow$	T$_G$<$\mathcal{T}(t)$>
$\mathcal{T}(Set(t))$	$\rightarrow$	LHSet<$\mathcal{T}(t)$>
$\mathcal{T}(decl(t, v))$	$\rightarrow$	$\mathcal{T}(t)$ v
$\mathcal{T}(assign(t, v, e))$	$\rightarrow$	$\mathcal{T}(decl(t, v)) = \mathcal{T}(e)$
$\mathcal{T}(assign(v, e))$	$\rightarrow$	v = $\mathcal{T}(e)$
$\mathcal{T}(forall(t, v_1, v_2, b))$	$\rightarrow$	for($\mathcal{T}(t)$ v_1 : v_2){$\mathcal{T}(b)$}
$\mathcal{T}(@(r(e_1, \ldots, e_n)))$	$\rightarrow$	r rec = new r($\mathcal{T}(e_1)$, ..., $\mathcal{T}(e_n)$); rec.apply(p)
$\mathcal{T}(@(r(p, e_1, \ldots, e_n)))$	$\rightarrow$	r rec = new r($\mathcal{T}(e_1)$, ..., $\mathcal{T}(e_n)$); rec.apply(p)
$\mathcal{T}(P(e_1, e_2))$	$\rightarrow$	new Pair($\mathcal{T}(e_1)$, $\mathcal{T}(e_2)$)
$\mathcal{T}(T(e_1, e_2, e_3))$	$\rightarrow$	new Triple($\mathcal{T}(e_1)$, $\mathcal{T}(e_2)$, $\mathcal{T}(e_3)$)
$\mathcal{T}(S(e_1, \ldots, e_n))$	$\rightarrow$	new LHSet<T>(){{add($\mathcal{T}(e_1)$); ...; add($\mathcal{T}(e_n)$); }} [4]
$\mathcal{T}(N(n_1, \ldots, n_n))$	$\rightarrow$	new Node(new LHSet<String>(){{add(n_1); ...; add(n_n); }})
$\mathcal{T}(+(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).addAll(s_2)
$\mathcal{T}(-(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).removeAll(s_2)
$\mathcal{T}(\&(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).retainAll(s_2)
$\mathcal{T}(\#(p, c))$	$\rightarrow$	p.getChannel(c)
$\mathcal{T}(.(v, c))$	$\rightarrow$	v.$\mathcal{T}(c)$
$\mathcal{T}(in(i))$	$\rightarrow$	getIn(i)
$\mathcal{T}(out(i))$	$\rightarrow$	getOut(i)
$\mathcal{T}(ends(p))$	$\rightarrow$	getEnds(p)
$\mathcal{T}(oper())$	$\rightarrow$	getOper()

[3] 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 4.2.

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

Listing 15 Example of a ReCooPLa reconfiguration translated.

```
public class OverlapP extends Reconfiguration {
    private CoordinationPattern p;
    private LHSet<Pair<Node, Node>> X;
    public OverlapP(CoordinationPattern arg1,
        LHSet<Pair<Node, Node>> arg2) {
        this.p = arg1;
        this.X = arg2;
    }
    public CoordinationPattern apply(CoordinationPattern pat) {
        Par par;
        Join join;
        par = new Par(this.p);
        par.apply(pat);
        for(Pair<Node> n : this.X) {
            Node n1, n2;
            n1 = n.getFst();
            n2 = n.getSnd();
            LHSet<Node> E = new LHSet<Node>() {{
                add(n1); add(n2);
            }};
            join = new Join(E);
            join.apply(pat);
        }
        return pat;
    }
```

It goes without saying that a translation can only occur when the ReCooPLa specification is syntactically and semantically correct. The ReCooPLa parser ensures syntactic correctness; on the other hand, a semantic analyser is defined to report errors concerning structure, behaviour and data types. Its definition is out of the scope of this paper.

Listing 15 shows the result of applying the translation rules to the *OverlapP* ReCooPLa reconfiguration documented in Listing 13.

6 Example

Consider a company that sells training courses on line and whose software system originally relied on the following four components: Enterprise Resource Planner (ERP), Customer Relationship Management (CRM), Training Server (TS) and Document Management System (DMS). In seeking an expedite expansion of the company and its information systems, a major software refactoring project was launched adopting a SOA solution. This entailed the need to change from the original structure of monolithic components into several services and their integration and coordination with respect to the different business activities.

One of the most important activities for the company concerns the updating of user information, which is accomplished taking into account the corresponding new user update services derived from the original ERP, CRM and TS components. Originally such an update was designed to be performed sequentially as shown in the coordination pattern of Figure 2.

However, other configurations were considered and studied taking advantage of the ReCooPLa language and the underlying reconfiguration reasoning framework. For instance,

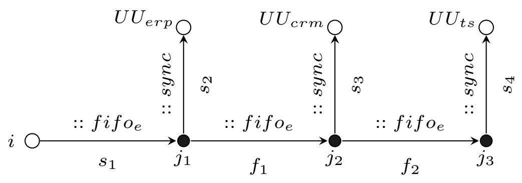

Figure 2 The User update coordination pattern. Each channel is identified with a unique name and a type (::t notation). It defines an instance of a sequencing pattern, where UU_{erp} executes first, then UU_{crm} and finally UU_{ts} with data entering in port i. Graphically, white circles represent input and output nodes while black ones represent mixed nodes.

Listing 16 `implodeP` reconfiguration pattern.

```
reconfiguration implodeP(Set<Node> X; Set<Name> Cs) {
        @ removeP(Cs);
        @ join (X);
}
```

another configuration for the user update activity may be given by the coordination pattern in Figure 3. This can be obtained from the initial pattern by application of a reconfiguration that collapses nodes and channels into a single node. In ReCooPLa, this is easy to define, as shown in Listing 16, resorting to removeP already defined in Listing 13.

This reconfiguration pattern takes as parameters the set of nodes and channels realtive to the strucutre one pretends to implode. Channels are removed and the nodes are joined. The translation mechanism of ReCooPLa specifications produces a Java class similar to the one presented in Listing 17.

Listing 17 ImplodeP class generated.

```
public class ImplodeP extends Reconfiguration {
    private LHSet<Node> X;
    private LHSet<Name> Cs;
    public OverlapP(LHSet<Node> arg1,LHSet<Name> arg2) {
        this.X = arg1;
        this.Cs = arg2;
    }
    public CoordinationPattern apply(CoordinationPattern pat) {
        RemoveP removeP;
        Join join;
        removeP = new RemoveP(this.Cs);
        removeP.apply(pat);
        join = new Join(this.X);
        join.apply(pat);

        return pat;
    }
}
```

In this example, applying $implodeP(\{j_1, j_2, j_3\}, \{f_1, f_2\})$ to the original coordination pattern would result in the one depicted in Figure 3, where (for reading purposes) node k is used to represent the union of j_1 and j_2.

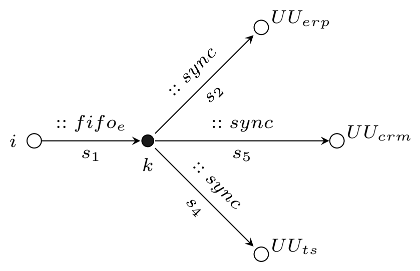

Figure 3 The User update coordination pattern reconfigured. It defines an instance of a parallel pattern, where UU_{erp}, UU_{crm} and UU_{ts} execute in parallel with data entering in port i.

7 Conclusions and Future Work

The paper introduces ReCooPLa, a DSL for the design of coordination-based reconfigurations. These reconfigurations act, through the application of primitive atomic operations, over a graph-based structure, which is an abstract representation of the coordination layer of a SOA-based system. ReCooPLa resorts to a Reconfiguration Engine that, via reflection, processes and applies such reconfigurations.

ReCooPLa differs from other architecture-oriented languages in the sense that it focus on reconfigurations rather than on the definition of architectural elements like components, connectors and their interconnections. Moreover, the language and the underlying approach is intended to target the early stages of software development; i.e., the design of reconfigurations and their analysis against requirements of the system. However, this approach may be lifted to the dynamic setting by mapping the code of each reconfiguration and coordination pattern to the actual coordination layer of a system. This would allow to reconfigure deployed systems offering an abstract, but effective way of planning such reconfigurations.

As future work, it is planned the full integration of ReCooPLa with the framework for reconfiguration analysis conceptualised in [19, 20]. In particular, it is intended to extend the language to cope with the probabilistic coordination model introduced in [22].

Acknowledgements. 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`. Nuno Oliveira is supported by an Individual Doctoral Grant from FCT, with reference `SFRH/BD/71475/2010`.

References

1 B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: a reconfiguration language for distributed systems. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems, 1994*, pages 15–22, 1994.
2 R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
3 G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computer Surveys*, 23(1):49–90, March 1991.
4 F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.
5 R. Balzer. Enforcing architecture constraints. In *Proceedings of ISAW'96*, pages 80–82, NY, USA, 1996. ACM.

6 E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

7 R. Bruni, A. Lluch-Lafuente, U. Montanari, and E. Tuosto. Style-based architectural reconfigurations. *Bulletin of the European association for theoretical computer science*, 94:161–180, February 2008.

8 P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications - Annales des Télécommunications*, 64(1-2):45–63, 2009.

9 T. Erl. *SOA Design Patterns*. Prentice Hall PTR, NJ, USA, 1st edition, 2009.

10 D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the CASCON'97*, pages 7–. IBM Press, 1997.

11 P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In I. Gorton, G. T. Heineman, I. Crnković, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *LNCS*, chapter 27, pages 352–359. Springer, 2006.

12 T. Kosar, N. Oliveira, M. Mernik, M. J. V. Pereira, M. Črepinšek, D. da Cruz, and P. R. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

13 C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, Amsterdam, The Netherlands, 2011.

14 C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011.

15 D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, September 1995.

16 J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of SIGSOFT'96*, page 3–14, NY, USA, 1996. ACM.

17 N. Medvidovic. Adls and dynamic architecture changes. In *Proceedings of ISAW'96*, pages 24–27, NY, USA, 1996. ACM.

18 M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys.*, 37(4):316–344, December 2005.

19 N. Oliveira and L. S. Barbosa. On the reconfiguration of software connectors. In *Proceedings of SAC'13*, pages 1885–1892, NY, USA, 2013. ACM.

20 N. Oliveira and L. S. Barbosa. Reconfiguration mechanisms for service coordination. In M. H. Beek and N. Lohmann, editors, *Web Services and Formal Methods*, volume 7843 of *LNCS*, pages 134–149. Springer, 2013.

21 N. Oliveira, M. J. V. Pereira, P. R. Henriques, and D. da Cruz. Domain-specific languages: a theoretical survey. In *INForum'09*, pages 35–46, Lisbon, Portugal, September 2009.

22 N. Oliveira, A. Silva, and L. S. Barbosa. Quantitative analysis of Reo-based service coordination. In *Proceedings of SAC'14*, volume 2, pages 1247–1254. ACM, March 2014.

23 P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of ICSE'98*, pages 177–186, WA, USA, 1998. IEEE Computer Society.

24 D. E. Perry. An overview of the state of the art in software architecture. In *Proceedings of ICSE'97*, pages 590–591, NY, USA, 1997. ACM.

25 D. Plump. The graph programming language GP. In S. Bozapalidis and G. Rahonis, editors, *Algebraic Informatics*, volume 5725 of *LNCS*, chapter 6, pages 99–122. Springer, Berlin, Heidelberg, 2009.

26 A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of SEAMS'10*, pages 49–58, NY, USA, 2010. ACM.

27 A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

28 A. L. Wolf. Succeedings of the isaw-2. *SIGSOFT Softw. Eng. Notes*, 22(1):42–56, January 1997.