ReCooPLa: a DSL for coordination-based reconfiguration of software architectures*

Flávio Rodrigues[†1], Nuno Oliveira[‡2], and Luís S. Barbosa[2]

1 **Departamento de Informática, Universidade do Minho**
Braga, Portugal
pg22826@alunos.uminho.pt

2 **HASLab - INESC TEC & Universidade do Minho**
Braga, Portugal
{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 language, software architecture, reconfiguration, software coordination

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

In the last few years, Service-Oriented Architecture (SOA) has been adopted as the architectural style to support the needs of modern intensive software systems [10]. 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

* This work was partially supported by someone.
† This author is supported by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by Portuguese Foundation for Science and Technology I. P. (PIDDAC) within project FCOMP-01-0124-FEDER-028923.
‡ This author is supported by an Individual Doctoral Grant from FCT, with reference SFRH/BD/71475/2010.

Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–17

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

approaches (*e.g.*, channels [4]), gain relevance by fomenting the desired decoupling between computation and coordination concerns. This separation of concerns makes SOAs flexible, reliable 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 is a desired feature 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 states to systems [27, 33]. These changes raise the need for the system to adapt to the new context while running.

Reconfigurations in 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 [30, 26, 13, 19, 31]. However, in some situations, this may not be enough. For instance, when a substituting service have incompatible interface, it may be necessary to target the way services interact with each other. This sort of reconfigurations go into the coordination layer and usually substitute, add or remove the components of the interaction (*e.g.*, *communication channels*), move communication interfaces between components and may even rearrange a complex interaction structure [16, 15]. Thus, there is a mismatch between the need and the offer currently existing in practice. More worryingly it is the lack of suitable formal methods to correctly design and analyse this sort of reconfigurations.

In previous work of these authors [22, 23], 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 *coordination pattern*) is regarded as a graph of channels where nodes are interaction points for either services or other coordination patterns; and edges are communication channels with a specific behaviour. But, to express and apply these reconfigurations, in practice, is not yet delivered by such framework, hindering its applicability. Therefore, in this paper it is introduced a Domain-specific Language (DSL), referred to as ReCooPLa, to express coordiantion-based reconfigurations, materialising the formal model presented in [22] and briefly discussed in further sections.

DSLs [32, 21, 24] are languages focused on a particular application domain, used to abridge the programming and the jargon of the domain. Their level of abstraction is tailored to the specific domain, allowing for embedding high-level 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) [14].

Consequently, ReCooPLa is a simple and small language that provides a formal, but still high-level, interface for reconfiguration designers. The reconfiguration construct plays, then, the 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 constituted of instructions, where coordination-specific notions are embodied in constructs that manipulate the graph structure underlying 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 the reconfigurations over the target coordination patterns. To this end, a translation of ReCooPLa constructs into the engine's running code is carefully defined.

Outline. In the reminder of this paper it is presented relevant related work (Section 2) and suitable background notions to make this document self-contained (Section 3). Then, in

Section 5 it is presented the reconfiguration engine. In Section 4 the ReCooPLa language is introduced with detailed description and examples; a translation of this language into the reconfiguration engine is also provided. Finally, in Section 8 the paper is closed with concluding remarks, discussing the work presented.

2 Related work

Domain-specific languages constitute an important tool to tackle the specificities of the domains to which they are tailored. Reconfigurations design in the context of SOA is the domain underlying this work. As such, in the following paragraphs, similar reconfiguration design approaches are addressed.

Fractal [6] is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems, that offers several features as, for instance, composite components and dynamic reconfiguration. To adapt a running system there should be identified the places where the changes have to be realized; and these changes must be applied taking into account the safety of the system, regarding the states of the components. For this, FPath and FScript [9] can be used. FPath and FScript notations are two DSLs to encode dynamic adaptation of Fractal-based systems. The former eases the navigation inside a Fractal architecture by using queries. The latter, which embeds FPath, enables the definition of adaptation scripts to modify the architecture of a Fractal application. FScript provides transactional support for architectural reconfigurations in order to ensure the reliability and consistency of the application if the reconfiguration fails at a given point.

In [7] Architectural Design Rewriting (ADR) is proposed as a declarative rule-based approach for modeling reconfigurable Software Architectures (SAs). ADR is a suitable and expressive framework based on an algebraic presentation of graph-based structures and conditional rewrite rules. The features of ADR are particularly tailored to understanding and solving Architecture Description Language (ADL) problematics. In fact, an ADR can serve as the basis to formalise or extend ADLs with features such as conditional reconfigurations.

ADLs provide a formal foundation for describing SAs by specifying the syntax and semantics for modelling and describing components, connectors, and their configurations. Their use has been limited to static analysis and system generation focused on static issues and, therefore, do not supporting architectural changes. However, a few ADLs, such as Darwin [18] or Rapide [17]can express run time modification to architectures provided that the modifications are specified during the design of the application.

Wright [2] and Acme [11] also offer mechanisms for specification of reconfigurations. For instance, in Acme, it is possible to represent reconfigurable architectures by expressing the possible reconfigurations in terms of Acme structures, which extend the original language. This defines Acme extensions to represent different types of reconfigurations at the architecture level.

While ADLs focus on 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 [20], and Clipper [1] are examples of such languages. In turn, Architectural Constraint Languages (ACLs) have been used to restrict the system structure using imperative [5] as well as declarative [18] specifications; other authors advocate behavioural constraints on components and their interactions [17].

The presented languages endow SA design approaches with means to specify reconfigurations. However, the reconfigurations targeted focus on the high-level entities of architectures,

rather than on coordination details. Also their focus is the dynamic reconfiguration; ReCooPLa, in contrast, is more of a static approach with a purpose of reconfiguration analysis.

Also related to ReCooPLa is the GP programming language presented in [28, 29]. It is a language for solving graph problems, based on a notion of graph transformation and four operators shown to be Turing-complete. The language allows for the creation of programs over graphs at a high-level of abstraction. A program is a set of rules defining a transformation scheme based on the double-pushout approach [8, 12]. Guards may be used to restrict the application of the program rules.

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 [22, 23]. In particular, it introduces the notions of coordination patterns and coordination-based reconfiguration notions, which are later embodied in the constructs of ReCooPLa.

3.1 Coordination protocols

A coordination protocol works as *glue-code* defining and constraining the interaction between components or services of a system. In this model, a coordination protocol is abstracted under the notion of *coordination pattern* and it is seen as a reusable and composable architectural element. It is formalised as a graph of channels where nodes are interaction points to plug 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, it is a set of edges:

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

where $\mathcal{N}$ is the set of nodes (to be precise, a node in a coordination pattern corresponds to a set of channel ends), $\mathcal{I}$ is the set of channel identifiers and $\mathcal{T}$ is a channel typing system. For the sake of examples, $\mathcal{T}$={sync, lossy, fifo, drain} is adopted as the channel typing system in this paper, which is borrowed from the Reo coordination model [4]. Details on this model and constituting channel types are not in the scope of this paper, though. In fact, their absence do not preclude the general understanding of the topic under discussion.

Moreover, it is defined a notion of input and output ends of a channel and this notion is reflected in coordination patterns under the notion of input and output ports. These correspond to nodes that are only input or output ends of the constituting channels. The remaining nodes are named internal or mixed nodes.

Listing 1 presents two coordination patterns. The coordination pattern `cp1`, for instance, comprises two channels: a channel `x1` of type sync, and channel `x2` of type lossy. The channel `x1` has an input node `a` and an output node `b.c`[1]. In turn, the 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 Coordination patterns example.

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

```
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 change to the original structure of a coordination pattern obtained through a sequential or parallel application of parametrised elementary operations, which are called reconfiguration primitives.

The most simple reconfigurations are the identity (`id`) and the constant (`const(`ρ`)`) primitives, where ρ is a coordination pattern. The former, returns the original coordination pattern while the latter replaces it with ρ.

The `par(`ρ`)` primitive, where ρ is a coordination pattern, 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 within 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 the opposite of `join` primitive because it breaks connections within a coordination pattern by separating all channel ends coincident in n. Listing 4 presents the resulting coordination pattern, after applying `split(h.i.j)` to the `cp1` from Listing 3. Notice that the ends composing node `h.i.j` will now belong to each channel that previously shared that 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 a channel from a coordination pattern, if it exists. In addition, if it is connected to other channel(s), the connection is also broken as much as it happens with the `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 composing end `c`, was removed with channel `x2`. Again, this process follows a non-deterministic approach.

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 for this to happen, it is essential that the operations are independent: *i.e.*, that they affect separated substructures of a coordination pattern. This possibility of composing primitive operations in sequence or parallel, allows for the definition of complex reconfigurations, referred to as *reconfiguration patterns*. As such, they affect significant parts of a coordination pattern at a time. Among other characteristics, they shall be generic, parametric and reusable.

In this regard, ReCooPLa language will offer a way of specifying such combinations, abstracting them in an approach close to imperative programming.

4 ReCooPLa: reconfiguration language

ReCooPLa is a language to design coordination-based reconfigurations. Its specific tailoring to the area of architectural reconfigurations, allows for the exploration of important characteristics of DSLs. One of the most important being the possibility of abstracting specific details, such as the effect of each primitive operation and their actual application (whether in sequence or in parallel), and hiding their actual computation under the processor.

4.1 Conceptual description

In ReCooPLa, the *reconfiguration* is assumed to be a first call concept, as much as the function concept is in other programming languages. In fact, these concepts share characteristics: both have a signature (identifier and arguments) and a body which designates a specific behaviour. But, in particular, a reconfiguration is always applied to, and always returns, a coordination pattern. The arguments, in their turn, may be elements of the following data types: *Name*, *Node*, *Set*, *Pair*, *Triple*, *Pattern* and *Channel*, each one with its idiosyncrasies.

The reconfiguration body is a list of different sorts of instructions. Special attention is devoted to the instruction of *applying* (primitive, or previously defined) reconfigurations, since this operation is the only responsible for changing the internals of a coordination pattern. But there are more. To support the application of reconfigurations, ReCooPLa counts on other constructs that mainly manipulate the parameters of each reconfiguration. In concrete, it provides means to declare and assign local variables, as well as the usual operations over such variables. In particular, field selectors and specific operations over structured data types and common set operations (union, intersection and subtraction). Finally, an iterative control structure is provided to iterate over the elements of sets.

From this small overview, it is induced that ReCooPLa is a small language borrowing most of its constructs from programming languages on an imperative paradigm setting.

To this design choice amount the facts that (i) reconfigurations are better expressed in a procedural/algorithmic way; (ii) they represent the flow of actions needed to change the configuration of a coordination structure and (iii) imperative languages present a more natural approach for explicitly describing such step-wise algorithms.

4.2 Formal description

In the sequel, we introduce the syntax of the language by presenting (the most important) parts of the underlying grammar. Along with the syntax, constructs are defined for further reference in this article.

Formally, a sentence of ReCooPLa specifies one or more reconfigurations.

Reconfiguation

A reconfiguration (formally presented in Listing 6) is then expressed like a function. The header is composed of a reserved word `reconfiguration` followed by a unique identifier (the reconfiguration name) and its arguments, which may be an empty list. The body is a list of instructions as explained later. In particular, the arguments are aggregated by data type, unlike 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 construct for a reconfiguration is given as: $rcfg(n, t_1, a_1, \ldots, t_n, a_n, b)$, where n is the name of the reconfiguration; each a_i is an argument with t_i the respective data type; and b is the body of the reconfiguration.

Data types

This language builds on a small set of data types: primitives (*Name* and *Node*), generics (*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, it 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 to their contents, as seen in Listing 7.

Listing 7 EBNF notation for the *datatype* production.

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

The structured data types have an internal state, matching their definition in Section 3. Each instance of these types are 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 traditionally: a data type followed by an identifier or an assignment. In 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 is a variable name; and e is an expression.

The control structure marked by the reserved word `forall`, is used to iterate over a set of elements, as in the spirit of other programming languages. Again, a list of instructions defines the behaviour of this structure. In Listing 8 it can be seen the respective production for this control structure.

Listing 8 EBNF notation for the *forall* production.

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

The construct 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 as an identifier followed by the `'@'` operator and a reconfiguration name. The last may be a primitive reconfiguration or some 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 `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
```

The construct for this operation is given either as $@(c)$ or $@(p, c)$, where p is a *Pattern* and c is a reconfiguration call. Each reconfiguration call also has its own construct: $r(a_1, \ldots, a_n)$, for r being a reconfiguration name, and each a_i its argument.

Operations

An expression is composed of one or more operations. These can be specific constructors for generic data types, including nodes, or operations over generic and structured data types. Listing 10 shows examples of these types of operations. 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 that shall agree to the data type. The corresponding production is given in Listing 10 and exemplified in Listing 11.

Listing 10 EBNF notation for the operations productions.

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

```
operation
        : ID ('#' ID)? '.' attribute_call
```

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);
```

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 10 (production `operation`). Symbol `#` is used to access a specific channel (the proceeding `ID`) from the internal structure of a pattern (the preceding `ID`). An `attribute_call` correspond 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 closed list of attributes/operations possible are 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, using the optional integer parameter, which will point to 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, also known as 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 of a *Pattern* variable.
- `channels`: returns a set of channels of a *Pattern* variable.
- `fst`: returns the first element from the *Pair* and *Triple* variables.
- `snd`: returns the second element from the *Pair* and *Triple* variables.
- `trd`: returns the third element from a *Triple* variable.

Listing 12 EBNF notation for the `attribute_call` production.

```
attribute_call
        : 'in' ( '(' INT ')' )?
        | 'out' ( '(' INT ')' )?
        | 'ends' '(' ID ')'
        | 'name' | 'nodes' | 'names' | 'channels'
        | 'fst' | 'snd' | 'trd'
```

All these operation give rise to its own language construct. For the sake of space, only a few are exemplified: the construct for the constructor of a *Pair* data type is $P(e_1, e_2)$, where e_1, e_2 are expressions; for the field selection it used $.(v, c)$, where v is a variable and c is a call to an operation; for the union of sets it is $+(s_1, s_2)$, with s_1, s_2 being variables of type *Set*. The remaining construcs, follow similar definitions.

Now, putting it all together, one can derive valid sentences of ReCooPLa. Listing 13 presents one such example, where 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 convenient arguments.

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);
    }
}
```

5 Reconfiguration engine

Like traditional programming languages, also ReCooPLa sentences require to be interpreted or executed in order to provide the expected results. While the former are translated into executable machine code, ReCooPLa is translated into a subset of Java code, which is then recognised and executed by an engine. This engine, referred to as the *Reconfiguration Engine*, is developed in Java programming language and, as expected, executes reconfigurations specified in ReCooPLa over coordination patterns, which are defined in CooPLa [23], a lightweight language to define the graph-like structure of coordination patterns. Its model is as simple as it can be, taking into account only a few entities. Figure 1 presents the respective Unified Modelling Language (UML) class diagram of that model.

The shaded package *cp.model* is respected to the model of a coordination pattern. This is actually, the implementation version of the formal model presented in Section 3. Important to know is that 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 counterpart method `getNodes()` in the CoordinationPattern class.

The remaining entities of the diagram are concerned with the reconfigurations themselves, and are assumed to belong to a *cp.reconfiguration* package. As easily induced, the classes Par, Const, Remove, Join, Split and Id are the implementation of the homologous 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 effects of their application over the coordination pattern p, given as 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

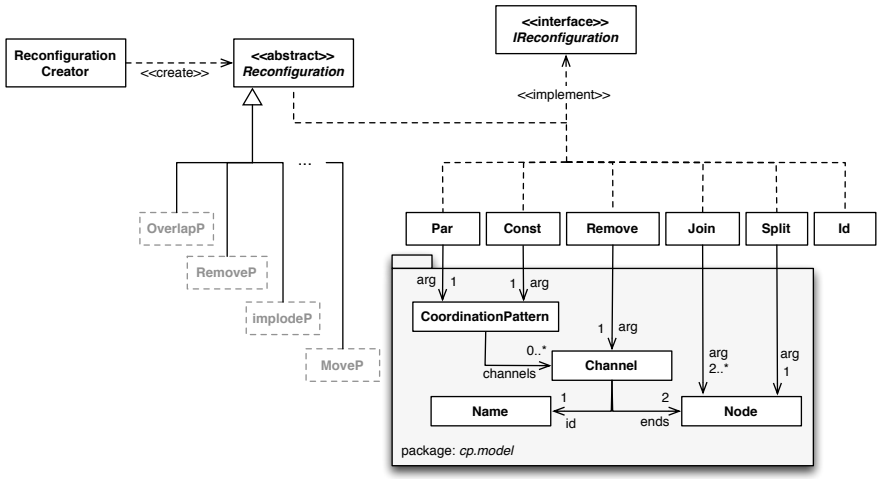

Figure 1 The Reconfiguration Engine model

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, in run time, by the ReconfigurationCreator class, taking advantage of reflection features of the 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, creates concrete classes of Reconfiguration. The idea behind this design 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 being possible, it is necessary to correctly translate ReCooPLa constructs into the code accepted by the Reconfiguration Engine. Section 6 goes through the details of such translation.

6 ReCooPLa translation

Throughout this section, it is assumed the existence of Java classes to match the types in ReCooPLa. This way, besides those classes already mentioned in Figure 1, the following are also assumed: Pair, with a getFst() and a getSnd() methods to access the fst and snd attributes of this class; Triple, extending Pair with an attribute trd and method getTrd(); and the LinkedHashSet from the *java.util* package, which will be abbreviated to LHSet for readability purposes. Moreover, to ease the understanding of the translation process, the details about reflection will be ignored or just abstracted. For instance, the 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, ..., a_n of type t_1, ..., t_n, respectively; and method apply with body b, which always ends with a return p instruction, where p is the argument of apply.

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

This being said, the translation of the constructs of ReCooPLa into the Reconfiguration Engine is given by the rule-based function $\mathcal{T}$(C), where C is a construct of ReCooPLa as presented in Section 4 and defined as shown in Table 1. Notice that details like semicolons and efficiency are not taken into account, for simplicity sake.

$\mathcal{T}(rcfg(n, t_1, a_1, ...t_n, a_n, b))$	$\rightarrow$	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 Node(new LHSet<T>(){{add($\mathcal{T}(e_1)$); ...; add($\mathcal{T}(e_n)$); }}) [3]
$\mathcal{T}(N(n_1, \ldots, n_n))$	$\rightarrow$	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()

Table 1 Translation rules for the ReCooPLa constructs. It is used: n for referring 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 in, out, name, names, nodes, channels, fst, snd and trd.

It goes without saying that a translation can only occur when the ReCooPLa specification is syntactically and semantically correct. The ReCooPLa parser is in charge to ensure the syntax correction of the consuming specifications; in turn, a semantic analyser is defined to report errors concerning structure, behaviour and data types. The definition of this ReCooPLa module is out of the scope of this paper.

In Listing 14, it is shown the result of applying the translation rules to the *OverlapP* ReCooPLa reconfiguration shown in Listing 13, which is a correct ReCooPLa specification.

Listing 14 Example of a ReCooPLa reconfiguraiton translated.

```
public class OverlapP extends Reconfiguration {
    private CoordinationPatter p;
    private LHSet<Pair<Node>> X;
```

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

```
public OverlapP(CoordinationPattern arg1,
    LHSet<Pair<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;
}
```

7 Example

Consider a company that sells training courses on line and whose software system originally relied on the following components: Enterprise Resource Planner (ERP), Customer Relationship Management (CRM), Training Server (TS) and Document Management System (DMS). To follow the tendency of modern software development and for an easier expansion of the company, the Chief Information Officer (CIO) decided to launch a system update project, where the adoption of a SOA solution was the key. This entailed the change of the monolithic components into several services and their integration and coordination with respect to the several business activities.

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

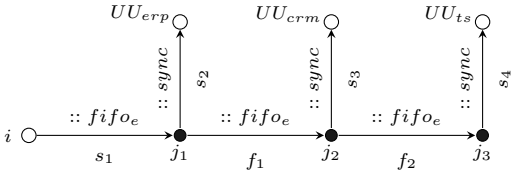

Figure 2 The User update coordination patttern. Each channel is identified with a unique name and a type (::t notation). It defines an instance of a sequencing pattern, where UU_{erp} executes first, then UU_{crm} and finally UU_{ts} with data entering in port i.

However, other configurations where considered and studied taking advantage of the ReCooPLa language and the underlying reconfiguration reasoning framework. For instance, 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 singe node. In ReCooPLa, this is easy to define, as it is shown in Listing 15, where removeP was already defined in Listing 13.

Listing 15 `implodeP` reconfiguration pattern.

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

This reusable reconfiguration pattern takes a set of nodes and another of channels respecting to the desire structure to collapse. Then, it removes the channels and the given nodes are joined. Using the translation mechanism of ReCooPLa specifications It would be obtained a Java class similar to the one presented in Listing 16.

Listing 16 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 the current example, applying $implodeP(\{j_1, j_2, j_3\}, \{f_1, f_2\})$ to the original coordination pattern would result in the one in Figure 3, where (for reading purposes) node k is used to represent the union of j_1 and j_2.

8 Conclusions and Future Work

The paper introduces ReCooPLa, a DSL for design of coordination-based reconfigurations. These reconfigurations actuate, 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 also counts on a Reconfiguration Engine that, via reflection features, processes and applies the reconfigurations over the coordination patterns.

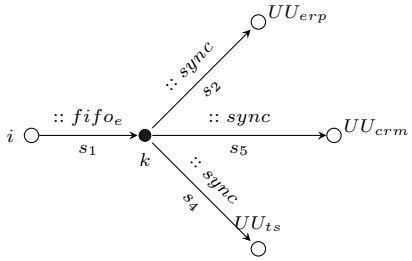

Figure 3 The User update coordination pattern reconfigured. It defines an instance of a parallel pattern, where UU_{erp}, UU_{crm} and UU_{ts} execute in parallel with data entering in port i.

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; in concrete, the design of reconfigurations and their analysis against requirements of the system. However, this approach may be lift 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 taking a very abstract way of planning such reconfigurations. Nevertheless, such lift shall be carefully analysed as several factors may hinder the correct and safe application of reconfigurations.

As future work, it is planned the full integration of ReCooPLa with the framework for reconfiguration analysis [22, 23]. In particular, it is intended to extend the language to cope with a quantitative/probabilistic model of coordination as elaborated in [25].

References

1 B. Agnew, Christine 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 Robert 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 Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computer Surveys*, 23(1):49–90, March 1991.

4 Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.

5 Robert Balzer. Enforcing architecture constraints. 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*, ISAW '96, pages 80–82, New York, NY, USA, 1996. ACM.

6 Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard 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 Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Style-based architectural reconfigurations. *Bulletin of the European association for theoretical computer science*, 94:161–180, February 2008.

8 Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

9 Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry 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.

10 Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

11 David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 7–. IBM Press, 1997.

12 Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

13 Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George T. Heineman, Ivica Crnković, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, chapter 27, pages 352–359. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

14 Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria J. V. Pereira, Matej Črepinšek, Daniela da Cruz, and Pedro R. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

15 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.

16 Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, Amsterdam, The Netherlands, 2011.

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

18 Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, page 3–14, New York, NY, USA, 1996. ACM.

19 M. Malohlava and T. Bures. Language for reconfiguring runtime infrastructure of component-based systems. In *Proceedings of MEMICS 2008*, Znojmo, Czech Republic, November 2008.

20 Nenad Medvidovic. 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*, ISAW '96, pages 24–27, New York, NY, USA, 1996. ACM.

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

22 Nuno Oliveira and Luís S. Barbosa. On the reconfiguration of software connectors. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1885–1892, New York, NY, USA, 2013. ACM.

23 Nuno Oliveira and Luís S. Barbosa. Reconfiguration mechanisms for service coordination. In Maurice H. Beek and Niels Lohmann, editors, *Web Services and Formal Methods*, volume 7843 of *Lecture Notes in Computer Science*, pages 134–149. Springer Berlin Heidelberg, 2013.

24 Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Domain-specific languages: a theoretical survey. In *INForum'09 — Simpósio de Informática: 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, pages 35–46, Lisbon, Portugal, September 2009. Faculdade de Ciências da Universidade de Lisboa.

25 Nuno Oliveira, Alexandra Silva, and Luís S. Barbosa. Quantitative analysis of Reo-based service coordination. In *Proceedings of SAC '14*. ACM, 2014. To Appear.

26 Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

27 Dewayne E. Perry. An overview of the state of the art in software architecture. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 590–591, New York, NY, USA, 1997. ACM.

28 Detlef Plump. The graph programming language GP. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics*, volume 5725 of *Lecture Notes in Computer Science*, chapter 6, pages 99–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

29 Detlef Plump. The design of GP 2. In Santiago Escobar, editor, *Proceedings of the 10th International Workshop on Reduction Strategies in Rewriting and Programming*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, Novi Sad, Serbia, 2011.

30 Andres J. Ramirez and Betty H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 49–58, New York, NY, USA, 2010. ACM.

31 Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

32 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

33 Alexander L. Wolf. Succeedings of the second international software architecture workshop (isaw-2). *SIGSOFT Softw. Eng. Notes*, 22(1):42–56, January 1997.