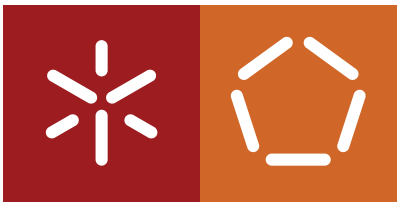


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Flávio Gonçalves Rodrigues

An Engine for Coordination-based Architectural Reconfigurations

Master dissertation

Supervised by: Luís Soares Barbosa

Nuno Oliveira

Braga, 31st October 2014

ACKNOWLEDGEMENTS

The realisation of the work presented in this dissertation would not have been possible without the support and contribution of several people to whom I convey my most sincere acknowledgements.

First of all, I would like to thank to Nuno Oliveira for the amazing support, encouragement, availability and dedication to make this project better and for giving me the opportunity to work on it. I will never be able to truly express my appreciation and gratitude to him. I would like to thank my supervisor Professor Luís Soares Barbosa for giving me guidance when I needed most, and for all the feedback and assistance that helped me to improve my work.

A big thank you goes also to my colleagues of the research office in the department where I worked and to my classmates for providing me an excellent working environment, good mood and willingness throughout this past year.

Last but not least, a special thank you to my parents, brother and all other family members and friends who always have encouraged and supported me unconditionally, and to all the people that were not mentioned here but directly or indirectly helped me throughout my academic journey.

ABSTRACT

In Service-Oriented Architectures (SOA), services are regarded as loosely-coupled components interacting with each other via connection of their public interfaces. Such interaction follows a (coordination) protocol usually established at design-time. However, in an environment where change is the rule rather than the exception, several aspects may contribute to a need for change in the way these services interact. To assess the consequences of applying these changes beforehand is an ultimate requirement for SOA design.

This M.Sc. dissertation proposes a practical approach to model reconfigurations of service coordination patterns. To achieve this, reconfigurations are specified (before being applied in runtime) using a domain-specific language – ReCooPLa – which targets the manipulation of software coordination structures, typically used in SOA. Then, a language processor, built according the traditional approach for compiler construction, is presented. It comprises a parser, a semantic analyser and a translator. The main outcome of this work is a reconfiguration engine that takes ReCooPLa specifications conveniently translated into Java code, and applies them to coordination structures.

This project is part of a broader research initiative aiming at formally modelling, reasoning and analysing reconfigurations of coordination patterns in the context of SOA and cloud-computing.

RESUMO

Em arquiteturas orientadas a serviços (SOA), os serviços são vistos como componentes independentes que interagem uns com os outros através da ligação das suas interfaces públicas. Tal interação segue um protocolo (de coordenação) que normalmente é estabelecido durante o design. No entanto, num ambiente onde a mudança é a regra e não a excepção, vários factores podem contribuir para uma necessidade de alterar a forma como estes serviços interagem. Compreender as consequências da aplicação destas alterações com antecedência é uma exigência final para desenho de uma SOA.

Esta dissertação de mestrado propõe uma abordagem prática para modelar reconfigurações de padrões de coordenação de serviços. Para tal, as reconfigurações são especificadas (antes de serem aplicadas em tempo de execução) através de uma linguagem de domínio específico – ReCooPLa – que visa a manipulação de estruturas de coordenação de software, tipicamente utilizadas em SOA. Posteriormente, é apresentado um processador para a linguagem, construído de acordo com a abordagem tradicional para a construção de compiladores. Este processador inclui o *parser*, o analisador semântico e o tradutor. O principal resultado deste trabalho é um motor de reconfiguração, que usa as especificações ReCooPLa convenientemente traduzidas em código Java e aplica-as a estruturas de coordenação.

Este projeto é parte de uma iniciativa de pesquisa mais ampla que visa modelar e analisar formalmente reconfigurações de padrões de coordenação no contexto de SOA e cloud-computing.

CONTENTS

Contents iii

1 INTRODUCTION 3
1.1 Statement of the Problem . . . 4
1.2 Objectives . . . 5
1.3 Dissemination . . . 6
1.4 Document structure . . . 6

2 BACKGROUND: COORDINATION MODELS 7
2.1 Coordination . . . 7
2.1.1 Overview of coordination models . . . 8
2.1.2 The Reo coordination model . . . 14
2.1.3 Formal models of Reo . . . 17

3 STATE OF THE ART REVIEW: SOFTWARE RECONFIGURATIONS 23
3.1 Architectural Reconfigurations . . . 23
3.2 Coordination based reconfigurations . . . 26
3.3 Languages for reconfiguration . . . 28
3.4 Tool support for reconfigurations . . . 31
3.5 A Reconfiguration Framework . . . 32
3.5.1 Coordination protocols . . . 32
3.5.2 Coordination-based reconfigurations . . . 34

4 RECOOPLA: THE RECONFIGURATION LANGUAGE 39
4.1 Conceptual description . . . 39
4.2 Formal description . . . 40
4.3 ReCooPLa Processor . . . 46
4.3.1 Technologies . . . 46
4.3.2 Development . . . 47

Contents

5 THE RECONFIGURATION ENGINE 57
5.1 Technologies . . . 57
5.2 The Engine Model . . . 58
5.3 ReCooPLa Translation . . . 60
5.3.1 Translation overview . . . 60
5.3.2 Translation implementation . . . 62
5.4 A practical example . . . 64
5.5 Architecture and Development . . . 65

6 CASE STUDY 69
6.1 The ASK system . . . 70
6.2 Adaptable-ASK design . . . 70

7 CONCLUSIONS AND FUTURE WORK 75

LIST OF FIGURES

Figure 1 Primitive Reo channels. 15
Figure 2 Reo connectors: (a) *Sequencer* and (b) *Exclusive Router* 16
Figure 3 Constraint automata for primitive Reo channels and two connectors. . 18
Figure 4 Reo automata for primitive Reo channels and the *sequencer* connector. 19
Figure 5 Two simple coordination patterns. 33
Figure 6 Resulting coordination pattern after applying the `par` primitive. 34
Figure 7 Resulting coordination pattern after applying the `join` primitive. . . . 35
Figure 8 Resulting coordination pattern after applying the `split` primitive. . . 35
Figure 9 Resulting coordination pattern after applying the `remove` primitive. . . 36
Figure 10 Reconfiguration patterns. 36
Figure 11 Internal representation of structured data types. 41
Figure 12 The compiler scheme. 48
Figure 13 The *removeP* AST. 51
Figure 14 The Reconfiguration Engine model 59
Figure 15 Example of a ReCooPLa reconfiguration translated. 63
Figure 16 The User update coordination pattern. 64
Figure 17 The User update coordination pattern reconfigured. 65
Figure 18 The System high-level architecture. 66
Figure 19 Annotations and problem markers on the editor. 67
Figure 20 Patterns Created View as part of the editor. 68
Figure 21 File dialog to save created patterns. 68
Figure 22 `Original` coordination pattern. 71
Figure 23 `ExRouter` coordination pattern. 71
Figure 24 ReCooPLa implementation of the `scaled-out` reconfiguration. 72
Figure 25 ReCooPLa implementation of three reconfigurations. 73
Figure 26 Partial RTS for the design of the Adaptable-ASK system. 73
Figure 27 ReCooPLa implementation of a reconfiguration to add a log. 74

LIST OF TABLES

Table 1 Mapping between the ReCooPLa structured data types and the Reconfiguration Engine classes 59
Table 2 Translation rules for ReCooPLa constructs 61

LIST OF LISTINGS

4.1 EBNF notation for the *reconfiguration* production. . . . 40
4.2 EBNF notation for the *datatype* production. . . . 41
4.3 EBNF notation for the *forall* production. . . . 42
4.4 EBNF notation for the *reconfiguration_apply* production. . . . 42
4.5 EBNF notation for the *constructor* production. . . . 42
4.6 *Constructors* input example. . . . 43
4.7 EBNF notation for the *operation* and *attribute_call* productions 43
4.8 ReCooPLa input example. . . . 44
4.9 EBNF notation for the *main* production. . . . 45
4.10 EBNF notation for the *main_instruction* production. . . . 45
4.11 Main reconfiguration in ReCooPLa 46
4.12 Parser `options` section. . . . 49
4.13 Tree operators example. . . . 49
4.14 Rewrite rules example. . . . 49
4.15 Tokens section in parser. . . . 50
4.16 Rewrite rules using created tokens. . . . 50
4.17 Tree grammar `options` section. . . . 51
4.18 Identifiers table for *removeP* reconfiguration. . . . 52
4.19 Example of the scope of a production. . . . 53
4.20 Error reporting example. . . . 54
4.21 Error message example. . . . 54
4.22 `emitErrorMessage` method rewriting. . . . 55

5.1 Template for the Reconfiguration classes. . . . 62
5.2 Template usage. . . . 62
5.3 Collection to save the result of the translation. . . . 63
5.4 `implodeP` reconfiguration pattern. . . . 64
5.5 ImplodeP class generated. . . . 65

1

INTRODUCTION

Cloud computing is a recent paradigm based on three computational layers: infrastructure, platform and software [BBG11]. These are provided as services by some organisations, and their main objective is to provide high performance while keeping low degradation of Quality of Service (QoS). This intrinsic objective of cloud computing is achieved by its main characteristic — the elasticity — which is responsible for the provision of the right amount of computational resources to the real-time demands of a software system. This sensation of infinite computational power and its need by the emerging software intensive systems, installed a shift in the software development. Software developed to and deployed in the cloud inherits its high performance, and is made available as a service (*e.g.*, Google Docs). Thus, Software as a Service (SaaS) deliver, over the web, a service to the end-users, regardless of their location, which allows activities to be managed from central locations in a "one-to-many" model.

A service, in this context, is a loosely-coupled entity that offers some specific computational functionality via published interfaces (APIs). A single service may result from the composition of several other services that interact with each other, while completely unaware of their surroundings. This notion of service and computation by interaction, gave rise to the adoption of Service-Oriented Architectures (SOA) as the architectural style underlying modern software systems [Erl09]. SOAs are then based on services often distributed by different organisations, agnostic of programming language and deployment platform and are coordinated to provide a desired functionality. Coordination of services (or the definition of their interaction protocols) is usually kept in a layer separated from their internal functionality. Such coordination follows protocols (or interaction policies) that may be encapsulated, for instance, using connectors and are usually established at design time [Arb04]. However, SOAs are flexible, reliable and naturally dynamic: although policies are pre-established, services may be discovered and bound to the architecture only at runtime, rather than fixed at design time [FL13]. Dynamic bound of services is deeply related with QoS needs. Thus, in an environment where change is the rule rather than the exception (*e.g.*, the cloud), keeping QoS levels above some quality contracted is essential and, in general, entails the need for reconfigurations such as the dynamic binding/unbinding of services. But system requirements (whether they are functional or non-functional) also change during the lifetime of a system, and therefore, changing the way services interact play an important role to this end. For instance, considering a simple

interaction of services with a channel which has buffering capacity representing a queue of requests; if we do a minor change on it such as append an additional channel (with same capacity), we might improve substantially the whole system; in this case, the change will allow enqueue more requests.

Reconfigurations in a SOA system usually targets the manipulation of services or the interaction protocols themselves. In the former, reconfigurations consist on the dynamic update of service functionality, substitution of services with compatible interfaces (but not necessarily the same behaviour) or removal of services [RC10, OMT98, HP06, MB08, SMR+11]. In the latter, a reconfiguration is more *low-level* and target the way services interact with each other. This sort of reconfigurations 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 [Kra11, KMLA11a].

This dissertation intends to deliver a formal framework for modelling, reasoning and analysing coordination reconfigurations in the context of SOA and cloud-computing, and it springs from a project that aims at studying reconfiguration of a system architecture [OB13a, OB13b].

1.1 STATEMENT OF THE PROBLEM

To assess the consequences of applying changes in a system beforehand is an ultimate requirement for SOA design. Software systems should adapt, at runtime, in order to meet new requirements and environmental conditions to maintaining, for instance, QoS levels, as initially agreed with the system consumers. In particular, this may lead to a need for change in the way services interact with each other.

In the last few decades, a number of (coordination) models have been designed, developed and presented to abstract and encapsulate the details of communication between services. All these models claim to provide a framework which enhances modularity, interoperability and reuse of components. However, they differ from each other in the definition of coordination, what is being coordinated and how coordination is achieved. They can be roughly divided, by the type of coordination they provide, in two categories: endogenous and exogenous. The main difference lies on the separation (or not) of the coordination and computation layers. In the former exists a mixture of coordination and computation code within a process definition. In the latter, on the other hand, there is a separation of concerns between coordination and computation. Linda [CG89] and Bonita [RW97] are examples of endogenous coordination models whereas Manifold [AHS93], ROAD [CH05] and Reo [Arb04] are examples of exogenous coordination models. Coordination can also be seen on two other perspectives in the context of SOA: orchestration (where components are coordinated without knowing each other) and choreography (where components are coordinated taking into consideration each other).

However, there is a lack of rigorous (formal) methods to correctly design and analyse (*coordination-based*) reconfigurations. In [OB13b], a formal model in which connectors are represented by a graph of *communication channels* is introduced. The graph nodes stand for interaction ports, where a subset of them constitute an interface for plugging concrete services or other such graphs; and edges are labelled with channel identifiers and types defining their behaviour. Such graphs are referred to as *coordination patterns*. A coordination pattern encodes a reusable solution for an architectural (coordination) problem, which is the description of interaction defined to answer to a set of requirements or constraints. In [OB13a] reconfiguration mechanisms to actuate over such patterns are proposed. In this setting, reconfigurations are defined as a combination of elementary reconfiguration primitives. The application of a reconfiguration to a coordination pattern yields a new coordination pattern.

Moreover, formal verification of requirements enclosed in such patterns and classification/organisation of reconfigurations are given as features of the framework discussed in [OB13a, OB13b]. But to express and apply reconfigurations, in practice, is not yet incorporated in such a framework, hindering its applicability. Thus, the following research question arises: *How to simulate a reconfiguration in their design?*

1.2 OBJECTIVES

This dissertation aims at developing a tool for rapid prototyping of coordination-based architectural reconfigurations. This involves several tasks: design of a language to express reconfigurations, through combination of primitive operations; development of a language processor that should report errors from syntactic and semantic analysis; and development of a engine for reconfigurations (based on the language), as an eclipse plugin, to model and simulate reconfigurations of service coordination protocols.

The first task implies the design of a Domain Specific Language (DSL) – referred as ReCooPLa – that supports the model presented in [OB13a] and makes it a suitable tool for the software architect. ReCooPLa provides a precise, high-level interface for the software architecture to plan and experiment with reconfiguration strategies. 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, as well as to hide their actual computation under a processor. The second task encompasses a few steps which ensure that the specification is syntactically and semantically correct. These "steps" follow the traditional approach for compiler construction [ASU86] and include a Lexer, a Parser, a Semantic Analyser as well as a Translator to obtain the necessary executable format. Finally, the last task is developed in Java. Thus, as it often happens with DSLs, the design language must be first translated into a subset of Java (by the language processor), which should be then recognised and executed by an engine. The engine execute coordination-based reconfigura-

tions specified in the design language, and apply them to coordination patterns, which are in turn defined in CooPLa [ROB14b], a lightweight language to define the graph-like structure of coordination patterns. In addition, the reconfiguration primitives described in [OB13a] as elementary operations, are also implemented in Java, to support the reconfiguration engine.

A suitable case study to test the reconfiguration engine is created. It is also intended to make the engine part of the Eclipse Coordination Tools [AKM+08], a fairly known platform for design and verification of Reo based coordination.

1.3 DISSEMINATION

During the research period a paper [ROB14a] named *"ReCooPLa: a DSL for coordination-based reconfiguration of software architectures"* was written, submitted and accepted in an international conference: the 3rd Symposium on Languages, Applications and Technologies (SLATE'14). This paper was invited for extension by the ComSIS journal [1], which has a 2 years impact factor of 0.549. Thus, a new paper [ROB14b] named *"Towards an engine for coordination-based architectural reconfigurations"* has emerged. This paper was also already submitted and is waiting for acceptance.

All information concerning the reconfiguration engine is also accessible in the project website [2].

1.4 DOCUMENT STRUCTURE

The state of the art of this work is subdivided in two chapters: Chapter 2 and Chapter 3. The former reviews some coordination models, giving emphasis to the Reo coordination model and its formal models. Chapter 3 addresses reconfigurations in Software Architectures and, in particular, coordination based reconfigurations. Tools and languages, namely ADL, that already exist to support the reconfigurations and their analysis are also presented in this chapter. In addiction, the reconfiguration framework is also introduced by the end of this chapter, by addressing coordination protocols and coordination-based reconfigurations (*e.g.*, reconfiguration patterns). In Chapter 4, the language developed for processing reconfigurations, as well as the respective processor and the supporting technologies, are presented and described. Chapter 5 describes the Reconfiguration Engine by presenting its model, architecture and development. Furthermore, a set of translation rules is presented and their implementation described in this chapter. Chapter 6 presents a case study in the context of self-adaptive systems, featuring the ASK system. Finally, Chapter 7 concludes the dissertation and raises some issues for future work.

1 http://www.comsis.org/indexing.php
2 http://coopla.di.uminho.pt/

2

BACKGROUND: COORDINATION MODELS

This chapter presents an overview of coordination through a set of distinct coordination models. The Reo coordination model receives particular attention due to its relevance to this work. Some formal models for Reo are also presented.

2.1 COORDINATION

The increase of computational resources has allowed the development of new programming paradigms based on distributed and parallel systems, which are able to, for instance, perform tasks faster and be fault tolerant. This led to the design and implementation of more complex applications and systems such as telecommunication networks, peer-to-peer networks, distributed databases, real-time process control (*e.g.*, aircraft control systems) and parallel computation.

The emergence of these systems drew attention to issues such as reusability, compositionallity and extensibility [PA98]. In order to deal with them and provide interoperation between systems, a blend of language models becomes necessary.

Coordination offers an alternative to mitigate the problems and address some of the issues that arise with the development of complex distributed and parallel computer systems [PA98, Arb98].

Actually, a programming model can be built through the combination of two distinct pieces: the computation model (processes involved in manipulating data) and the coordination model (responsible for the communication between the processes) [GC92, PA98]. According to Gelernter and Carriero [GC92], a *coordination model* is "the glue that binds separate activities into an ensemble".

Coordination models are defined by three main components: coordination entities (*e.g.*, threads, users, etc), coordination media (*e.g.*, channels, tuple spaces, etc) and coordination laws (*e.g.*, enact either synchronous or asynchronous behaviours for coordination entities) [Cia96]. A coordination model can be embedded either in a coordination architecture (*e.g.*, client-server architecture) or in a coordination language. A coordination language is

"the linguistic embodiment of a coordination model" [GC92]. It provides operations to create computational activities and support communication between them.

All coordination models claim to provide a framework which enhances modularity, interoperability and reuse of components. They differ from each other in the definition of what is being coordinated and how coordination is achieved.

2.1.1 *Overview of coordination models*

Papadopoulos and Arbab [PA98] argue that coordination models and languages can be classified as *data-driven* or *control-driven*. Briefly, in the former exists a mixture of coordination and computation code within a process definition whereas in the latter there is a separation of coordination from computational concerns. In the data-driven category, coordination tends to be endogenous and is usually provided as a set of primitives which is mixed within some "host" computational language (*i.e.*, embedded within computational entities). This category is mostly used for parallelising computational problems and coordinating manipulated data. On the other hand, in the control-driven category, coordination tends to be exogenous and is usually delivery as a language by itself (*i.e.*, isolated from computational entities). This category is mainly used for modelling systems and coordinating entities, such as system components, since the structure and contents of data is of minor or no importance in it. Whereas in the data-driven category coordinator processes directly handle and examine data values, in the control-driven category processes are treated as black-boxes, whether coordination or computational processes, and therefore the communication of processes with their environment is achieved through well-defined interfaces (input and output ports).

Most of the coordination models in the data-driven category are deeply related to the notion of a *Shared Data-space*. This is a language paradigm in which computations are performed using a content-addressable communication medium with a tuple-like representation [RC90].

LINDA

Linda [CG89] is a data-driven and endogenous coordination language. It is based on *generative communication*: if two processes need to communicate, the data producing process generates a new data object, referred to as a tuple; on the other hand, to create processes (*e.g.*, a process needs to create a second one) the so-called "live tuple" is first generated and then also turned into an ordinary data object tuple. In both cases, the tuple is stored in a shared data-space called *tuple-space*. Then, herein the receiver can access the new tuple.

Linda offers very simple coordination entities (active and passive tuples, which represent processes and messages, respectively), a unique coordination medium (tuple-space), and a small number of coordination laws (embedded in four primitives only). The active tuples

representing processes in tuple space are turned into ordinary passive tuples right after the completion of their execution.

The small set of coordination primitives provided by Linda to perform operations on the tuples and tuple-space are: `in(t)`, `out(t)`, `rd(t)` and `eval(p)`; where `t` stands for tuple and `p` for process. The `in(t)` primitive retrieves a passive tuple `t` (*i.e.*, atomically reads and consumes a tuple) from the tuple-space. On the other hand, the `out(t)` primitive puts a passive tuple `t` in the tuple-space. The `rd(t)` primitive retrieves a copy of passive tuple `t` (*i.e.*, non-destructively reads a tuple) from the tuple-space. Lastly, `eval(p)` puts an active tuple (*i.e.*, a process `p`) in the tuple-space. While the `out` and `eval` primitives are non-blocking primitives, the `in` and `rd` are blocking primitives and thus will suspend execution until the desired tuple be found. If more than one tuple in the tuple-space matches, then one is selected nondeterministically. However, these primitives also have non-blocking forms (`inp` and `rdp`) which do not block if the required tuple is not found.

Linda is designed to be coupled with a host programming language and their primitives are independent of this host language. There are many implementations and extensions of Linda such as TSpaces (IBM) [WMLF98], JavaSpaces (Oracle) [FAH99], Bauhaus Linda [CGZ95], LAURA [Tol96], Bonita [RW97] and LIME [PMR99] to name but a few. They are developed using many host languages (C, Lisp, Pascal, Prolog, Java, etc) and paradigms (imperative, functional, object-oriented, etc) [ACG86, CG89].

MANIFOLD

Manifold [AHS93, Arb96b, Arb98] is a control-driven and exogenous coordination language that models external components as processes. This language is based on the Idealised Worker Idealised Manager (IWIM) [Arb96a] model, which is an abstract model of communication (through broadcast of events or point-to-point channel connections) that "supports the separation of responsibilities and encourages a weak dependence of processes on their environment". In other words, IWIM is focused on the separation of concerns (computation concerns isolated from the communication ones) and anonymous communication (processes that intercommunicate without knowing each other). This separation, along with the higher-level abstractions, improve software productivity, maintainability, modularity, and even reusability.

Manifold is a strongly-typed, block-structured and event-driven language. Thus, the only control structure that exists in it is an event-driven state transition mechanism and its entities are events, processes, ports and streams (asynchronous channels). The Manifold events are used purely for triggering state changes and do not carry data. In turn, a process is a black-box component with well defined ports, connected between themselves by means of streams. These processes are called Manifold coordinators which are, similarly to the most of the other control-driven coordination languages, clearly distinguished from computational processes.

The Manifold system runs on multiple platforms and consists of a compiler, a runtime system library, a number of utility programs, and libraries of built-in processes. Furthermore, Manifold has a visual programming environment called Visifold [BA96].

ARC

In the past few decades, well-defined mathematical abstractions for concurrent computation in a distributed environment such as CSP [Hoa78], π-Calculus [MPW92], and the Actor model [Agh86] have been studied. The latter is used by the Actor, Role and Coordinator (ARC) coordination model to model the concurrent computational part of an Open Distributed and Embedded (ODE) system. In addition, to deal with the coerced coordination part of an ODE system, the ARC model introduces the concept of a role that provides an abstraction for coordinated behaviours, which in turn may be shared by multiple concurrent entities called *actors*. The ARC model [RYC+06] is described by the authors as a "role-based decentralised coordination model for ODE systems" that intends to better address the dynamicity and scalability issues inherent in ODE systems while fulfilling the system's QoS requirements. These requirements are mapped to coordination constraints and imposed on actors through message manipulations, which are carried out by roles and coordinators – the coordination entities.

Furthermore, the ARC model can be seen as the composition of three layers, related to each of the components of the model: actor layer, role layer and coordinator layer. The actor layer is dedicated to functional behaviour and is independent from the coordination (composed by the role and coordinator layers) which is exogenous and imposes the above mentioned QoS constraints among the actors, preserving the semantics of the original model [Agh86]. The actors – the computation entities – are single-threaded active objects which communicate with each other only through asynchronous messages [SR11]. They have states and behaviours which can only be changed by themselves while processing a message. An actor can perform only three atomic primitive operations: *create* new actors, *send* messages to other actors and *become*, while processing a message, to the actor assume a new state and behaviour. The coordinators on the coordinator layer are responsible for the coordination among roles. The role layer lies between the actor and coordinator layers and therefore acts as a bridge between these two layers. A role may enable the coordination of a set of actors without requiring the coordinator to have an accurate knowledge of the actors that play the role. A role can also be seen as an active coordinator that manipulates the message sent and received by the actor. In addition, a role may generate events (viewable by coordinators) and change its state. In short, just as actors react to messages, roles and coordinators react to events. Due to the introduction of roles, the coordination is divided into inter-role and intra-role coordination, which are the responsibility of coordinators and roles, respectively. This allows a better separation of concerns and reduces the complexity of the entities. Roles and coordinators

themselves can be viewed as *meta-actors* and react to meta-messages and actor events. The role meta-actors are able to observe and manipulate messages in the actor layer.

ROAD

The Role-Oriented Adaptive Design (ROAD) [CH05] framework is a control-driven coordination model that extends work on object-oriented role and associative modelling [KO96, Ken00]. This model targets scalability issues of concurrent and distributed systems through a role-based coordination model. Similarly to the ARC model, the elements being coordinated are roles, which can be added to, and removed from, objects. These roles also abstract coordination behaviours among the objects that play them. However, they have different definitions. Kristensen and Osterbye [KO96] provide a definition of roles based on the distinction between intrinsic and extrinsic members of an object. Intrinsic members provide the core functionality – *i.e.*, the computational and communication capabilities – of the object, while extrinsic members (methods and data) contain the functionality of the role.

The ROAD approach to create adaptive software systems is based on the distinction between functional and management roles. Functional roles are focused on achieving the desired application-domain output and constitute the process as opposed to the control of the system. Since functional roles do not directly reference each other, they are associated by contracts that mediate the interactions between them. In turn, the creation and monitoring of these contracts is the responsibility of a concrete type of management role: the "organiser" role.

ROAD contracts are association classes that express the mutual obligations and interactions of the contract "parties" (*i.e.*, modules or processes) to each other. Similarly to roles, contracts have management and domain function levels. Management contracts specify the type of communication acts and protocols that are permissible between the two parties whereas functional contracts specify, among other things, the performance obligations and inherit control relationships from these management contracts.

PBRD

The Reflective Russian Dolls (RRD) [MT02] is a model of reflective distributed object computation, based on rewriting logic. Rewriting logic [Mes92] is a formalism designed for modelling and reasoning about concurrent and distributed systems. Its states are represented as elements of an algebraic data type and behaviour is given by local transitions between states described by rewrite rules.

The RRD model uses reflection and hierarchical structure to provide a layered exogenous coordination model, wherein each layer controls the communication and the execution of objects, *i.e.*, coordinators, in the layer below. A coordinator has an attribute that holds a nested configuration of objects and messages and its behaviour is specified by rewrite rules.

These rules also control delivery of messages in the configuration of a coordinator and specify how peer-to-peer messages are processed.

Policy-based RRD (PBRD) [Tal06] is a restricted form of RRD in which communication control is specified by declarative policies to, for instance, ordering of message delivery, serialising requests and recording a history of events. It is focused on logical communication constraints. Similarly to the ARC model and based on the actor model of computation [Agh86], the objects coordinated in PBRD are actors and the coordinators are meta-actors [TSR11]. These actors communicate by asynchronous message passing and encapsulate their state and thread of control. On the other hand, requirements of the coordinators are specified by informal constraints on the resulting interactions of the coordinated actors.

BIP

Basu et al. [BBS06] introduced the Behaviour-Interaction-Priority (BIP) language for modelling heterogeneous real-time components. It supports the construction of hierarchically structured components from atomic components characterised by their behaviour and interfaces. These components are obtained through the overlap of three layers: the lower one describes behaviour; the middle layer models interactions between components, specified by a set of connectors; and the upper layer is a set of priority rules (describing scheduling policies for interactions). Any combination of behaviour, interaction and priority models meaningfully defines a component. The connectors are used by the language to specify possible interactions between components and priorities. Interactions express synchronisation constraints between the composed activities of the components and priorities filter possible interactions to guide system evolution in order to address the performance requirements. The combination of interactions and priorities defines an abstract concept of architecture separate from behaviour.

BIP uses a parameterised binary composition operator on components, which allows incremental construction, *i.e.*, obtaining a compound component by successive composition of its constituents. The language also provides a powerful mechanism for structuring interactions involving strong (*rendezvous*) or weak (broadcast) synchronisation.

In contrast with other component frameworks, BIP executes atomic components concurrently and coordinates them in terms of high-level mechanisms such as protocols and scheduling policies [BBB+11]. Since this language focuses on the organisation of computation between components, it can be viewed as an ADL (see Section 3.3).

ORC

Orc [Mis05] is a coordination model, focused on the paradigm of orchestration that supports a structured model of concurrent and distributed programming. This model introduces the concept of a *site* as a basic (web) service, such as sequential computation or data manipu-

lation. It also provides constructs to orchestrate concurrent invocation of sites in order to managing time-outs, delays and failure of components and communication. Orc is highly asynchronous, naturally dynamic, based on ephemeral connections to services and deals well with failure [PC08].

An Orc program is composed by an expression with a set of definitions. An Orc expression can be a primitive site call, a reference to another expression, or a composition of expressions; and it is responsible for (dynamically) initiate contact with external sites. A site call is written as *M(p)*, where *p* is a tuple of arguments which can be constants or variables. Orc's semantics is detailed in [Mis05].

Orc is built upon three composition operators for parallel computation, sequencing and selective pruning. This model represents a multi-threaded computation by an expression which has useful algebraic properties such as CSP [Hoa78] and π-Calculus [MPW92]. However, unlike these, Orc permits integration of arbitrary components – the so-called *sites* – in a computation, which introduces a distinction between it and the environment in which it runs. Since Orc describes the structure of a distributed computation using primitives that define common communication patterns, the author [Mis05] argues that it may be a viable alternative to process calculi. This model can be applied in the development of workflow systems since, according the author, there is no commonly-accepted theory of workflow.

PICCOLA

PICCOLA [ALSN01] is a composition language and a formal model for component-based application development that embodies the paradigm described by the authors as "Applications = Components + Scripts". In other words, applications are seen as compositions of components, which are black-box entities that encapsulate services behind well-defined interfaces whereas scripts encapsulate how the components are composed. Thus, it is made a clear separation of computational elements (*i.e.*, components) and their coordination (*i.e.*, scripts). This separations enhances the flexibility, extensibility, and maintainability of an application.

PICCOLA has a small syntax and a minimal set of features needed for specifying different styles of software composition. Its constructs are translated into the πL-calculus [Lum99] – polymorphic variant of the π-calculus [MPW92]. A component is regarded as a set of interconnected agents used to model coordination abstractions. Its interface is represented as a *form*, which is a special notion of extensible and labeled records, used to model extensible interfaces and contexts. These agents communicate with each other by sending *forms* through private channels instead of tuples. In fact, agents and forms are the foundation of this composition language since they provide a good basis for specifying higher-level components and connectors.

Despite the fact that the natural type of interaction described by the πL-calculus is directed channel communication, PICCOLA defines also language constructs to simplify the encoding

of other types of interaction such as event based communication or failures. These constructs are functions, infix operators to support an algebraic notion of architectural style, and the explicit notion of a (dynamic) context to encapsulate required services.

PICCOLA has a syntax similar to that of Python and Haskell and thus newlines and indentation, for instance, are used rather than braces or end statements to delimit forms or blocks. Forms, however, may also be specified on a single line by using commas and brackets as separators. This language was implemented in Java and a gateway interface was also defined in order to use external components into the PICCOLA system. The reflection package of Java was used internally to embed arbitrary Java objects into PICCOLA scripts. In turn, this scrips can be embedded into stand-alone Java applications, applets or servlets.

2.1.2 *The Reo coordination model*

Reo [Arb04] is a channel-based exogenous coordination model which defines the primitive operations that allow for composition of channels into complex connectors.

Some of the models previously presented are highly expressive but the Reo model is more mature, with several formal semantics and tools for analysis. This model is highlighted in this work since it is the chosen one to describe the associated semantics of the reconfiguration framework, presented in Section 3.5, which in turn is supported by ReCooPLa.

The word ”exogenous” means ”from outside”, so exogenous coordination means coordination from outside. Thus, Reo separates the computation layer from the coordination layer. The Reo model, similarly to other control-driven models, isolate coordination by considering functional entities as black boxes [SR11]. Reo extend the IWIM [Arb96a] model (see Manifold description in previous section) by treating both computation both computation and coordination components as composable Abstract Behaviour Types (ABT), which is, as well as IWIM, a two-level control-driven coordination model where computation and coordination concerns are achieved in separate and independent levels. Coordination in Reo is abstracted as a Reo circuit specified by, for instance, a constraint automaton [BSAR06], while in the PBRD [Tal06] model coordinations are described as informal rule specifications on the resulting interactions of coordinated actors. Reo is closer to being a programming model, while RRD – briefly presented before, along with the PBRD model – focuses on more abstract specifications. A detailed comparison among the Reo, PBRD, and ARC model can be found in [TSR11].

A connector in Reo explicitly represents an interaction between architectural components and acts as ”glue-code” that connects these components and coordinates their activities in a component-based architecture. Formally, a connector is defined as a graph of channels whose nodes represent interaction points and channels have a behaviour that imposes a coordination policy between two points.

Connectors are bound to each other using their interfaces, which are defined by their boundary channel ends. A connector can perform I/O operations — *write* and *take* — on these ends. It is by synchronising these operations that coordination is achieved [CCA07].

CHANNELS

Channels constitute the primitive connectors in Reo. Each channel is defined as a medium of communication with exactly two ends and a specific behaviour [Arb04]. Each channel end can be a source or a sink end. The former accepts data into its channel and the latter expels data out of its channel. Usually, a channel is directed, *i.e.*, have a source and a sink end, but Reo also accepts undirected channels, which have two ends of the same type.

Reo presents a plethora of primitive channels, offering different synchronisation, buffering, lossy and even directionality policies. Figure 1 recalls the basic channels used in Reo that can be used to built more complex connectors.

Figure 1: Primitive Reo channels.

A channel can perform a *write* of some data on a source end, or a *take* on a sink end. The *write*/*take* will succeed when the connector either accepts the data from a writer component, or makes available data for a reader component.

A *sync* channel has a source and a sink end. This channel transmits data from one end to another whenever there is a request at both ends synchronously, otherwise one request shall wait for the other. Thus, this blocks a *write* operation on its source end or a *take* operation on its sink end, as necessary, to ensure that these two operations succeed synchronously.

The *lossy* channel behaves likewise, but data may be lost whenever a request at the source end is not matched by another one at the sink end. Thus, all *write* operations on its source end are immediately succeed, which means that if exists a pending *take* on its sink end, data will be transferred, otherwise the write operation succeeds but data will be lost.

The *fifo* channel has a buffering capacity of one memory position, therefore allowing for asynchronous occurrence of I/O requests. Qualifier e or f refers to the channel internal state, which can be *empty* or *full*, respectively. If the buffer is empty: a *write* operation on its source end succeeds, and fills the buffer; a *take* operation on its sink end is delayed until buffer is full. Then, if the buffer is full: a *take* operation on its sink end succeeds and removes the data from the buffer; a *write* operation on its source end is delayed until buffer is empty again.

The synchronous *drain* channel accepts data synchronously at both ends and loses it, since it has two source ends and does not have any sink end. Thus, it is not possible to take any

data out of this channel, since all data entering is lost, when the operations on both source ends are synchronised.

In Reo, channels can be joined together by their ends. The junction of ends defines nodes, and altogether (channels and nodes) define complex coordination structures called connectors.

NODES

A node is composed of (one or more) channel ends, and are classified into three different types: source, sink or mixed node. The node type depends on the type of their coincident channel ends. If a node connects only source channel ends, it is classified as a source node. On the other hand, if it connects only sink channel ends, it is classified as a sink node. Finally, if it connects both kinds of channel ends, it is classified as a mixed node.

A node acts as a *synchronous replicator* when it connects more than one outgoing channel. It replicates data to all outgoing channels, when a *write* operation can be performed in all of them, synchronously.

On the other hand, a sink node acts as a *non-deterministic merger* when it connects more than one incoming channel. It merges data from the incoming channels, when at least one of them offers data (*i.e.*, a *take* can be performed). In particular, if more than one channel offers data, only one of them is selected non-deterministically, *i.e.*, data is randomly chosen from one of the channels with fairness.

A a mixed node combines both behaviour by consuming data from one of its sink ends and replicating it to all source ends.

CONNECTORS

In Reo, connectors are built from channels and nodes and have an interface, or interaction points, that correspond to a set of source and sink nodes. These connectors are entities that provide the ”glue-code” for coordinating the interactions between architectural components. In Figures 2 depict two examples of connectors.

Figure 2: Reo connectors: (a) *Sequencer* and (b) *Exclusive Router*

Figure 2 (a) presents a *Sequencer* connector composed by four *sync* and one *fifo* channels connected together. Graphically, white circles represent the connector interface, *i.e.*, source or sink nodes. In turn, black ones represent mixed (internal) nodes.

This connector implements a generic sequencing protocol that can be parameterised to have as many nodes as required, by inserting more *sync* and *fifo* channel pairs [Arb04].

Figure 2 (b) shows an *Exclusive Router* connector that accepts data on its source node (a), and depending on the context (*i.e.*, depending on the existence of requests in one of the sink ends), routes data synchronously to one of its two sink nodes (b or c).

If both sink nodes are able to accept data, only one of them is selected non-deterministically. This is due to the fact that node `k` merges its inputs without priority, *i.e.*, exactly one of the *sync* channels is activated, replicating data on the active side to the corresponding sink node (b or c); the data item on the other side is destroyed by its *lossy* channel [AKM+08, Kra11].

2.1.3 *Formal models of Reo*

In the last decade, many semantic formalisms for describing the behaviour of Reo connectors have emerged, including coalgebraic models, operational models, and models based on graph-colouring. [JA12] presents an overview of the existing semantic formalisms for modelling Reo connectors.

In this section, we will focus on three of the most adopted models: Constraint Automata [BSAR06], Reo Automata [BCS12] and Connector Colouring model [CCA07].

CONSTRAINT AUTOMATA

Baier et al.[BSAR06] introduce an operational model for Reo called Constraint Automata (CA), to describe the behaviour and possible data flow in coordination models that connect anonymous entities to enable their coordinated interaction. CA yields a foundation for the formal verification of coordination mechanisms, *i.e.*, conceptual generalisations of automata where data constraints, *i.e.* boolean expressions for the data values, influence applicable state transitions. The automata states stand for the possible configurations (*e.g.*, the contents of the *fifo* channels of a Reo connector) while the automata transitions represent the possible data flow and its effect on these configurations, *i.e.* the set of ports enabled and its constraints.

The constraint automaton of a given Reo connector is defined in a compositional way: CA defines the Reo primitive channels, and through application of *join* and *hide* operations compose them to devise the behaviour of a complex connector.

The constraint automata corresponding to the Reo channels (*sync*, *lossy*, *drain* and *fifo*), as well as two connectors (*Sequencer* and *Exclusive Router*), can be seen in Figure 3. The transitions of constraint automata are labeled with pairs consisting of a non-empty subset N of $\{A_1, ..., A_n\}$ and a data constraint g. Data constraints can be viewed as a symbolic representation of sets of data-assignments. Formally, data constraints are propositional formulae built from the atoms "$d_A = d$" where data item d is assigned to port A. The most

commonly used boolean connectors are $\wedge$ (conjunction), $\oplus$ (exclusive or), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence).

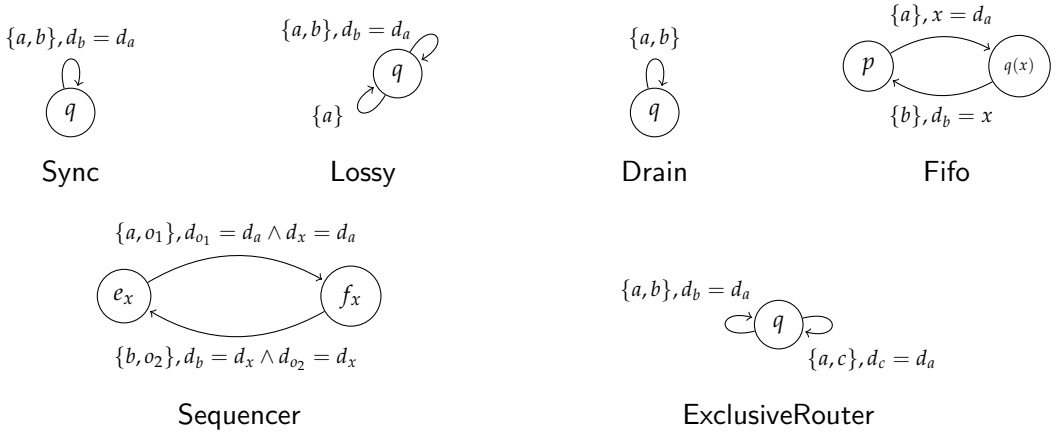


Figure 3: Constraint automata for primitive Reo channels and two connectors.

The semantics of Reo connectors can also be given using Constraint Automata with State Memory (CASM) [PSHA12]. There are several variations of ordinary CA and CASM is one of them [JA12]. In CASM, communication and synchronisation is realised using port names, and states can be enriched with local memory cells, which can be seen as finite representation of data elements. These automata derive executable coordinator models from connectors (*e.g.*, as generated Java code) in the Reo implementation, using the Extensible Coordination Tools (ECT) [Kra11], presented in the section 3.4.

Notions of bisimulation equivalence and a simulation relation for CA that provide methods for checking language equivalence or language inclusion for non-deterministic automata are also introduced in [BSAR06].

In their basic form, CA cannot express context dependency. Context dependency enables connectors to be more responsive to changes in their environment, and can express, for instance, the priority of one behaviour over another. In CA, for instance, the constraint automaton corresponding to *lossy* Reo channel *c.f.*, Figure 3, allows data to be lost regardless of the context, *i.e.*, irrespective of the existence or not of a request (*take*) on sink end, thus misinterpreting the intended operational semantics.

REO AUTOMATA

Bonsangue et al. [BCS12] introduce a new automata-based semantic model for expressing context-dependent Reo connectors, called Reo Automata. This model took into account the failures and benefits of previous automata-based approaches such as CA [BSAR06], to provide a behavioural description of Reo connectors. The Reo automata corresponding to primitive channels are very compact and intuitive, with a small number of states and transitions, comparatively to other contemporary models [Cos10].

Reo Automata extend the notion of context-dependent automata to include the modelling of data flow, as in CA. Reo automata overcomes the CA deficient handling of context by explicitly modelling both the presence and absence of requests on the channel ends, which means that data can only be lost, in a *lossy* channel sink end, if a request does not exist on it. Thus, this extra information correctly gives semantics to context-dependent channels and connectors, such as the *lossy* channel or the *exclusive router* connector, as can be seen in Figure 4. The Reo automata corresponding others Reo channels, as well as the sequencer connector, are also depicted in Figure 4. Intuitively, a Reo automaton is a non-deterministic automaton whose transitions have labels of the form $g|f$, where g is a *guard* (boolean condition) and f a set of nodes that fire synchronously. In addition, $\overline{g}$ can be used to describe the negation of g. A transition can be taken only when its guard g (or its negation) is true.

Each transition labeled by $g|f$ satisfies two criteria: reactivity and uniformity. According the former, data flows only on nodes where a request is pending. The latter captures two properties: the request set corresponding precisely to the firing set is sufficient to cause firing; and removing additional unfired requests from a transition will not affect the (firing) behaviour of the connector [BCS12].

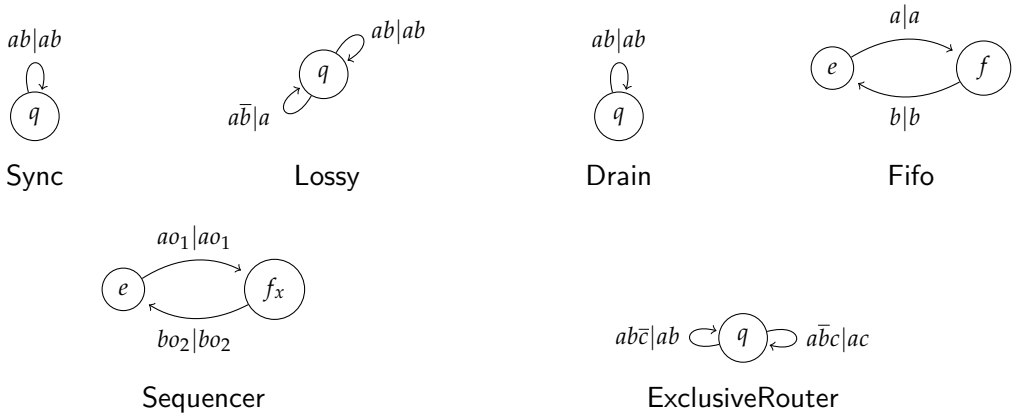


Figure 4: Reo automata for primitive Reo channels and the *sequencer* connector.

CONNECTOR COLOURING

Clarke et al. [CCA07] present a semantic model based on connector colouring for resolving the context dependent synchronisation and mutual exclusion constraints required to determine the routing for data flow in Reo connectors. This model aims at facilitating the data flow computation (and implementation) of Reo connectors in a distributed computing environment. It requires less mutual exclusion in a distributed implementation and does not require backtracking.

One aspect of the CA model is that transitions are labelled with the collection of nodes that synchronously succeed in a given step, at the exclusion of all other nodes present in the

connector being modelled. Calculating this set based on the configuration of a connector is precisely what connector colouring achieves. That is, the 2-colouring model of a connector produces a set of colourings which can be compared with the transitions in the corresponding CA. This set comprises a so-called *flow* and a *no-flow* colours. The former marks places in the connector where data flows, and the latter marks the absence of data flow.

Using 3-colouring model, the *no-flow* colour of 2-colouring model is replaced with two different *no-flow* colours to trace the exclusion constraints responsible for the no-flow back to their origins, *i.e.* provide information about the reason for the absence of data flow, allowing to properly model context-sensitive connectors.

The set of the 3-colouring model is $Colour = \{\text{————}, - - \triangleleft - -, - - \triangleright - -\}$.

Graphically, the arrow indicates the direction of exclusion, *i.e.* it points away from the exclusion reason and in the direction in which the exclusion propagates.

Connector colouring permits to capture context-dependent behaviour and therefore provide semantics to context-dependent connectors, by considering the context (the presence or absence) of pending I/O operations at the ends of a connector. Thus, channels and other primitive connectors determine the actual data flow based on the colouring of their ends.

Each colouring of a connector is a solution to the synchronisation and exclusion constraints imposed by its channels and nodes.

In order to understand it better, let us present some primitive Reo channels using the 3-colouring model.

- *sync channel*

  i) ———— *ii*) $- - \triangleright - -$ *iii*) $- - \triangleleft - -$

  In configuration *i*) data flows through the channel. In *ii*) data does not flow through the channel and the reason is given that a problem occurred in the input operation. Finally, in *iii*) data does not flow through the channel and the reason to delay relates to a problem occurred in the output operation.

- *drain channel*

  i) ———— *ii*) $- - \triangleright - -$ *iii*) $- - \triangleleft - -$

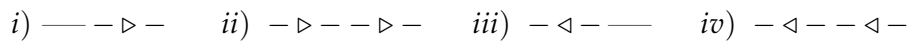
  The configurations are the same as for the previous channel, however with a slightly different meaning on *ii*) and *iii*). In the former data flow is delayed due to a problem in the input operation at the left source end, and in the latter, the data flow is delayed due to a problem in the source end at the right.

- *lossy channel*

  i) ———— *ii*) $—\triangleleft - -$ *iii*) $- - \triangleright - -$

Configuration *i*) occurs when both source and sink end perform their I/O operations (*write* and *take*) synchronously. In *ii*), the input operation is performed, but the data is lost because there is no output operation at the sink node. The arrow pointing to the left propagates this reason to the point where data is lost. In *iii*) the data does not flow because the input operation failed. The reason propagates into the direction of the sink node as if it was a signal to explain what happened at the other side of the channel.

- *fifo channel*



This channel has four configurations: two for empty state plus two for full state. In *i*) the input operation occurs, but the output operation does not, since the buffer is empty, and thus no output operation may occur. In *ii*) the input operation fails and the reason is propagated through the channel. In *iii*) the input operation is not performed since the buffer is full. Nevertheless the output operation is performed. In *iv*) the output operation is not performed, thus the buffer is still full and consequently the reason is propagated to inform the source end of the channel that an input operation should be delayed.

3

STATE OF THE ART REVIEW: SOFTWARE RECONFIGURATIONS

Nowadays SOA is in focus mainly due to the services interoperability that this architectural style enables, regardless of the language in which each service was implemented or even the hardware in which it runs. The emergence of cloud-computing also had an important role to make SOA the paradigm underlying modern software systems.

A major concern in these systems is to maintain QoS levels, in particular, a continuous availability of services. For this, and in order to adapt a system to new requirements, environments or failures [Per97, Wol97], their underlying architecture can be changed or reconfigured, either by dynamically updating service functionalities, substituting, adding or removing services; or, by changing the behaviour or structure of software connectors.

This chapter reviews the notion of reconfiguration in architectural design and introduces the specific reconfiguration model implemented in this MSc dissertation.

3.1 ARCHITECTURAL RECONFIGURATIONS

The architecture of a software system consists, basically, of a set of interconnected components and protocols that determine their interaction. In architecture design emerges the concept of architectural style [SG96], *i.e.*, some set of rules indicating which components can be part of the architecture and how they can be properly interconnected.

Architectures usually are designed and changed in a static way by refining abstract components or assembling subsystem architectures. However, in some architectures namely in critical software systems, this might not be possible once stoping or even restarting a system for changes or updates leads to reducing QoS levels and increasing costs and risks. Changes in the configuration of the architecture of a running system are called dynamic reconfigurations. Often, however, the facilities for runtime modification found in current operating systems, distributed object technologies, and programming languages, do not ensure the consistency, correctness, or desired properties of dynamic change [OMT98].

Dynamic software reconfiguration consists in changing the configuration of an architecture at runtime. Some distributed systems use functional redundancy or clustering in order to avoid the need for runtime change. Upgrades to web servers, for instance, are held by for-

warding incoming network traffic to a redundant host while the original host is reconfigured. However, due to the cost and even the risk that this approach implies, this is not always feasible.

Software Architectures (SAs) have the potential to provide a foundation for systematic runtime software modification and can support different types of software evolution, such as corrective, perfective, and adaptive evolution [GJM02].

Oreizy et al. [OMT98] present an architecture-based approach to runtime software evolution that operates at the architectural level. The main purpose of this approach is to reduce the costs and risks typically associated with dynamic reconfiguration. Other approaches to runtime software evolution have been proposed [GR91, GJB96, PHL97].

Gorlick and Razouk [GR91] present a data flow based approach to runtime change called *Weaves*. A weave is an arbitrary network of tool fragments connected together by transport services. In turn, each tool fragment is a small software component that performs a well-defined function and may retain state. However, *Weaves* does not currently provide a mechanism to check the consistency of runtime changes neither explicit support for representing change policies.

Grupta et al. [GJB96] describe an approach to modelling changes at the statement level for a simple abstract imperative programming language. In this approach, applications must be written entirely in the dynamic language to benefit from dynamism, which leads to a performance overhead since every function invocation must be bound during runtime. In addition, application behaviour and dynamism are not explicitly separated.

Peterson et al. [PHL97] present an approach to module-level runtime change based on Haskell, a purely-functional programming language. This approach allows refined control over runtime change, since software architects can implement change policies tailored to the application. However, managing change in large systems can be complex since change policies are not isolated in the application source code.

On the other hand, the approach in [OMT98] provide several benefits over previous approaches, such as a common representation for describing software systems and managing runtime change, separation of computation from communication, and encapsulation of change application policies and scope within connectors. In this approach, the design of dynamically reconfigurable software systems is focused on the behaviour of individual application components.

Gomaa and Hussein [GH04] present an approach for dynamic reconfigurations based on software reconfiguration patterns, which can be used with the design of self-adaptive component-based software systems.

A software reconfiguration pattern defines how a set of components, forming an architecture or design pattern, cooperate to change the current software configuration. A Reconfiguration Framework for change management, presented in the mentioned work, initiates and controls

the automatic reconfiguration of the software system from one dynamic configuration to another.

Designing runtime reconfigurations at a software pattern level provides better organisation and deeper understanding since it operates at a larger level of abstraction than the component level does. For runtime evolution of a SA, it is important to consider how components can coordinate their communication during the reconfiguration and then design such inter-component behaviour as a reconfiguration pattern.

Bruni et al. [BLLMT08] discuss different reconfiguration mechanisms such as graph rewrite rules (reconfigurations as graph transformations) and inductive reconfigurations. Since it represents architectures by graphs, reconfigurations can also be seen as graph transformations, and to formalize them, graph rewrite rules are used. The use of graphs and graphs transformations to model architectural styles has been previously proposed by several authors [Roz97, LM98, Tae04, BHTV06, Gad07].

The approach presented in [BLLMT08] has some advantages over similar previous approaches namely [LM98], such as allowing for representing complex reconfiguration rules because it is a hierarchical and inductively based approach. This approach also uses graph rewriting as a unifying model to represent architectural design, behaviour and reconfiguration while other approaches use separated formalisms for these issues.

Wermelinger and Fiadeiro [WF99] propose a uniform algebraic approach to address problems manifested in other languages such as low-level of specification. Thus, in this approach, components are written in a high-level program design language and two existing frameworks, for specifying architectures and rewriting labelled graphs respectively, are combined. Furthermore, this approach shows the relationships between reconfigurations and computations while keeping them separate, because it provides a semantics to a given architecture through the algebraic construction of an equivalent program. The authors introduce, as well, a program design language that incorporates some of the usual programming constructs while keeping a simple syntax to be formally manageable.

Bravetti et al. [BDGPZ12] introduce a core calculus of adaptable processes that allows for expressing a wide range of patterns of process evolution and runtime adaptation. This improves the kind of reconfiguration that can be expressed in existing (higher-order) process calculi. The authors also study structural and behavioural characterisations and show that such a distinction may make a difference in terms of process expressiveness and decidability of reachability-like properties.

Grassi et al. [GMS07] present a model-driven approach to automatically transform a design model into an analysis model, to support the model-based analysis of the effectiveness of reconfigurable component-based applications. In particular, this approach relies on the existence of intermediate languages and extends one of them to capture core features of a dynamically reconfigurable component-based system.

Bruni et al. [BLLMT08] propose Architectural Design Rewriting (ADR) as a declarative rule-based approach for modeling reconfigurable SAs. ADR is a suitable and expressive framework based on an algebraic presentation of graph-based structures and conditional rewrite rules. In particular, these features enable the modelling of, for instance, hierarchical designs, and inductively defined reconfigurations, besides ordinary computations. 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.

3.2 COORDINATION BASED RECONFIGURATIONS

A reconfiguration of a service-based application may become necessary if one of the services in use suddenly becomes unavailable, its Quality of Service (QoS) level becomes unacceptable or its behavioural interfaces change. However, when a reconfiguration of an application is needed, in particular at runtime, it is usually not possible to stop or to restart the intended services since they can be supplied as third-party services.

While in some domains switching between a finite set of configurations to accommodate the new requirements is sufficient, in others a rule-based approach for reconfiguration is more suitable to perform structural changes [Kra11]. Krause proposes a rule-based rewriting approach for modelling distributed component connectors and their (dynamic) reconfigurations based on distributed graph transformation concepts.

Due to the graph structure inherent to Reo networks, methods from graph transformation presented in [Kra11] are suitable to model and implement reconfigurations in Reo. Important aspects of this approach include rule-based definition and atomic execution of complex reconfigurations.

In [Kra11, KMLA11b], a graph transformation rule is defined as a pattern which must be matched, and a template which describes the changes to be performed to the system. Additional application conditions may further restrict the applicability of a transformation rule. This approach allows to specify concretely in which situations and how a system should be changed, including possible dependencies that must be updated. Furthermore, these rules can be applied either locally or globally, *i.e.*, wherever patterns match. With this approach complex structural reconfigurations can also be achieved in an atomic step, instead of sequentially performing low-level modifications on primitives.

The AGG [ABJ+10] tool and the Henshin [Tae04] framework are used to formally analyse the reconfiguration approach in [Kra11]. Furthermore, [Kra11] refers that the AGG system can be used to detect conflicting reconfiguration rules using critical pair analysis and the Henshin can be used to generate state spaces and to do qualitative and quantitative model

checking. The analysis tools in AGG and Henshin can also help to circumvent design errors that occur when dealing with inaccurate implementations of reconfigurations.

As a complementary approach to critical pair analysis, model checking can be used as well for verifying reconfiguration behaviour. In [Kra11] it is shown that using a model checking approach, it is possible to analyse dynamic reconfigurations. One way of analysing dynamic reconfigurations and the interplay of execution is by encoding both behavioural aspects in the same formalism and then apply, for instance, model checking.

Krause et al. [Kra11, KAV09] follow the same approach of graph transformation to provide a formal framework for modelling Reo connectors and their reconfigurations in a distributed setting. For this, it considers connectors which are distributed over a network, encapsulated (their internals are hidden from their surroundings) and linked together only via their published interfaces. Reconfiguration of a network is achieved by reconfiguring its constituent connectors (*e.g.*, through a change in its interfaces) and is defined and performed locally. However, it can be either triggered internally or invoked from the outside. Reconfiguring a connector may also require connectors in its neighbourhood to reconfigure, which implies a need for synchronising local reconfigurations into a consistent reconfiguration of the connector as a whole [Kra11]. Nevertheless, in a distributed setting, it can not be assumed the existence of a third-party that coordinates local reconfigurations. Thus, to ensure the consistency of a reconfigured network other mechanisms should be used. In Krause's thesis [Kra11] some of them are used, presented and proposed: a framework of distributed graph transformation [Tae99, EOP06]; a synchronisation mechanism based on the notion of amalgamation [BFH87, CMR+96, TB94]; and a distributed strategy to organize the stepwise reconfiguration of large networks.

Rooting on his initial approach for coordination reconfiguration [Kra11], Krause et al. [KLA08] propose a framework for dynamic reconfigurations triggered by data flow, which relies on connector colouring semantics [Cos10, CCA07]. This framework allows to define such dynamic reconfigurations by annotating transformation rules with colourings, which leads to a notion of dynamic connectors.

Transformations are automatically applied depending on the structure, the state and the context of a connector. Connectors are reconfigured at runtime based on this information, which leads to a powerful notion of dynamic connectors and dynamic reconfigurations.

Bozga et al. [BJMS12] propose Dy-BIP, which is a dynamic extension of the BIP language (see Subsection 2.1.1) rooted in rigorous operational semantics for modelling dynamic architectures. This framework supports a powerful and high-level set of primitives for describing dynamic interactions, which are (i) expressed as symbolic constraints offered by interacting components and (ii) computed efficiently by an execution Engine.

Dy-BIP allows the construction of composite hierarchically structured components from atomic components and relies on a clear distinction between behaviour and architecture. Each

atomic component provides its own interaction constraints at each computation step. In turn, interactions at some state may depend on interactions in the past and thus it is necessary to parametrize interaction constraints by *history variables*. These variables keep track of the interactions already executed, which avoid state explosion and duplication of ports.

An atomic component is an automaton extended with history variables. Each transition of it is labeled by a port, an interaction constraint and a set of history variables to be updated.

The composition semantics has been implemented by using a centralised execution Engine, which gathers current state interaction constraints from all atomic components. Thus, atomic components interact and coordinate their execution through the Engine. Hereinafter, the Engine builds – at every state reached at execution – the overall system of constraints, finds the set of maximally satisfying interactions and then select and execute (atomically by all involved components) one of them.

3.3 LANGUAGES FOR RECONFIGURATION

The Service Component Architecture (SCA) defines a framework for describing the composition and the implementation of services using components. Each component requires and provides services. However, SCA does not provide support for context awareness [PBD09]. On the other hand, Fractal [BCL+06] 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. Several controllers have been defined in the Fractal specification such as the binding controller, that allows the dynamic binding and unbinding of component interfaces.

FPath and FScript notations [DLLC09] 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.

ADLs provide a formal foundation for describing SAs by specifying the syntax and semantics for modelling and describe components, connectors, and their configurations. Numerous ADLs have been developed, each providing complementary capabilities for architectural development and analysis. The use of ADLs has been limited to static analysis and system generation focused on design issues and, therefore, do not provide support for specifying runtime architectural changes. However, a few ADLs, such as Darwin [MK96] or Rapide [LV95] can express runtime modification to architectures providing that the modifications are specified during the design of the application.

Darwin [MDK92] is an ADL which allows distributed and parallel programs to be constructed from hierarchically structured configuration descriptions of the set of component instances (which communicate by message passing) and their interconnections.

In this language, components are characterised by the services they provide (to allow other components to interact with them) and the services they require (to interact with other components). The Darwin compiler checks that bindings are only made between required and provided services with compatible interaction classes and datatypes. The interaction and datatype information specified at the component interface is optional in this model since if this information is necessary, the compiler infers the type of interface objects which are not explicitly typed. Thus, Darwin descriptions are reusable and concise.

Magee and Kramer [MK96] describe two techniques used in Darwin to capture dynamic structures: lazy instantiation and direct dynamic instantiation. The former allows a structure to evolve according to a fixed pattern. These structures must have acyclic bindings and have been found to be mainly useful in the domain of parallel processing. The latter, on the other hand, allows the definition of structures which can evolve in an arbitrary way. It can be used in a way which balances flexibility at runtime with the advantages of retaining a structural description. In [MK96], support for dynamic binding both inside and outside Darwin is also discussed.

Rapide [LV95] is a concurrent event-based simulation language for defining and simulating the (dynamic) behaviour of system architectures and the properties such as synchronisation and timing. It is one of the few architectural languages which also supports dynamic architectures, providing facilities for both dynamic connections and instantiation of components.

The approach in [OMT98], in contrast, can accommodate unplanned modifications of an architecture and incorporate behaviour which was not anticipated initially; it augments current ADLs with runtime change support. This approach consists of several interrelated mechanisms that apply, for instance, architectural changes to a system model. An accurate model of a system architecture must be available during runtime in order to modify a system. Thus, an architectural model – which describe the interconnections between components and connectors, and their mappings to implementation modules – is deployed.

The modifications are expressed in terms of this architectural model. A modification description uses operations for adding, removing or replacing components and connectors as well as changing the architectural topology. They are provided by multiple organisations and applied by end–users based on their particular needs. The modifications that compromise system integrity are restricted through the use of constraints.

The runtime architecture infrastructure maintains the consistency between the architectural model and implementation as modifications are applied. It also reifies changes in the architectural model to the implementation and prevents runtime changes from violating ar-

chitectural constraints. In addition, it can support different component addition, removal, and replacement policies and can be tailored for particular application domains.

There are still other approaches that provide a formal basis for architectural specification such as Wright [All97] and Acme [GMW97].

Wright defines a set of standard consistency checks and it focuses on the concept of explicit connector types, on the use of automated checking of architectural properties, and on the formalisation of architectural styles [All97]. Wright was also extended to handle dynamic aspects of architecture [ADG98].

Dynamic Wright allows to analyse statically the kinds of dynamic architectural changes that can occur in a running system since it provides the localisation of reconfiguration behaviour. It provides also a uniform representation of reconfiguration behaviour and steady state behaviour and clearly delimits interactions between these two kinds of behaviour by using control events. These control events are also used in a separate view of the architecture which describes how they trigger reconfigurations.

The semantic foundations of Dynamic Wright are based on CSP [Hoa78] and thus it can exploit traditional tools and analytic techniques based on process algebras. Dynamic Wright is restricted to dynamic architectures that have a finite number of possible reconfigurations in order to be able to provide strong support for static reasoning and automated analysis.

Acme is an extensible generic ADL for representing architectures and an annotation mechanism for describing additional semantics. This scheme allows subsets of ADL tools to share architectural information that is understood by all, and tolerate the presence of information that is not.

The software architect, in Acme, may state constraints using an extension Armani [Mon01]. The languages has also built-in support for dynamic reconfiguration, using Plastik [BJC05].

Armani extends Acme with a language for expressing architectural constraints over architectures. Thus, it can be used, for instance, to express constraints on system composition, behaviour, and properties. These constraints are specified in first-order logic; predicates are referred to as functions.

Acme may also be used to represent reconfigurable architectures by expressing the possible reconfigurations in terms of Acme structures. This means that dynamic reconfiguration is not originally addressed by Acme but it can be handled using the extensible mechanism of the language. Thus, Plastik defines Acme extensions to represent different types of reconfigurations at the architecture level.

Plastik is a meta-framework that arises from the integration of an ADL (an extension of Acme/Armani enhanced with new constructs for dynamic reconfiguration) with a reflective component runtime (OpenCOM [CBG+04]). In turn, the OpenCOM component runtime is used to build reconfigurable systems software elements such as middleware and programmable networking environments.

Plastik supports both programmed and ad-hoc reconfiguration. The former is supported at the ADL level and is related to changes that can be predicted at system design time. The latter, on the other hand, is intended for changes that are not and cannot be predicted at system design time. Nevertheless, it is intended to build general invariants into the specification of the system and to accept any change as long as the invariants are not violated.

While ADLs focus on describing SAs for the purposes of analysis and system generation, 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 [Med96], and Clipper [AHP94] are examples of such languages.

3.4 Tool support for reconfigurations

Over the last years, tools to support (dynamic) reconfiguration of software architectures have been developed. FraSCAti [SMF+09] is one of them.

FraSCAti is an open source Fractal-based SCA platform (see Section 3.3), endowed with dynamic properties that enable reconfiguration of components at runtime. 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 [DLLC09] – presented in Section 3.3 – can be used.

Oreizy et al. [OMT98] present ArchStudio, which is a tool suite that implements several interrelated mechanisms for supporting the runtime reconfiguration of software architectures. The tools comprising ArchStudio are implemented in the Java programming language, and can modify C2-style applications written using the Java-C2 class framework [MOT97]. C2 is a component- and message-based style designed to support of applications with a significant Graphical User Interface (GUI) aspect [TMA+95]. Thus, this framework provides a set of extensible Java classes for C2 concepts such as components and connectors. ArchStudio includes three tools: Argo [RHR96], ArchShell [Ore96], and the Extension Wizard. Argo and ArchShell are design tools for software architects to describe architectures and architectural reconfigurations. Argo provides a graphical depiction of the architectural model that the architect may manipulate directly. On the other hand, ArchShell is an alternative to Argo that provides a command-driven interface for specifying runtime reconfigurations. Extension Wizard is used to deploy a reconfiguration, after it has been specified and thus provides a simple end-user interface for enacting runtime reconfigurations.

The Extensible Coordination Tools (ECT) is a powerful tool suite for Reo [AKM+08]. It consists of several development tools, implemented as plug-ins for the Eclipse platform, for modelling, simulation, animation and verification of Reo connectors. In particular, ECT contains, among others, a graphical automata editor and a plugin for converting Reo connectors

to CA, and allows for reconfigurations, *i.e.*, changes in the topology of Reo connectors, using algebraic graph transformation, via a basic reconfiguration view.

To support reconfigurations in ECT, Krause [Kra11] extends the Reo model with the possibility of associating reconfiguration actions with primitives and nodes. Through this, it is possible to define reconfiguration rules in ordinary Reo files. A reconfiguration rule is a high-level description of a set of primitive reconfiguration steps. They are used inside the editor, for dynamic creation of connectors. Executable instances of connectors can be derived by generating code from Reo specifications or by interpreting these specifications. The graphical Reo editor automatically highlights parts of a connector that are augmented with reconfiguration actions. Reconfiguration rules can be applied in two different modes: local or global. A local rule is applied exactly where it is defined. On the other hand, since global rules are defined externally, they can not be applied where they were defined and thus they are applied to the complete content of another Reo file. Reconfiguration rules are applied locally only in specific regions of the connector. These regions are the reconfigurable parts of the connector that can be formally viewed as disjoint sub-graphs that restrict the domain of the transformations. Each of these regions has a number of reconfiguration rules attached [KCPA08]. Reconfiguration rules can also be translated into Henshin transformation rules [ABJ+10] since the Reo model is defined using EMF.

The Reo engine [KCPA08] includes two independent components to execute dynamic connectors: one for computing colouring tables and to perform the data-flow, and one for computing reconfiguration matches and executing the transformations.

ReoLive is a centralised implementation of dynamic reconfigurations that uses the CA based interpreter engine of Reo. The deployment of connectors in ReoLive is achieved by uploading Reo files. A constraint automaton is automatically generates from the Reo model by the application, and then it can be executed on the ReoLive server.

3.5 A RECONFIGURATION FRAMEWORK

This section provides an informal account of the reconfiguration model, introduced and formalised in [OB13a, OB13b]. In particular, it presents the coordination protocols and coordination-based reconfiguration notions, which are used later in the design of a reconfiguration language and its supporting engine.

3.5.1 *Coordination protocols*

In the context of this framework, a coordination protocol abstracts the *glue-code* that defines and constrains the interaction between components or services of a system. In the underlying model, a coordination protocol is named coordination pattern and is seen as a reusable and

composable architectural element. It is defined as a graph where nodes are interaction points and edges are channels formally structured as follows

$$\mathcal{C} \subseteq 2^{\mathcal{E}} \times \mathcal{I} \times \mathcal{T} \times 2^{\mathcal{E}},$$

where $\mathcal{E}$ is the set of channel ends, $\mathcal{I}$ is the set of channel identifiers and $\mathcal{T}$ is a channel typing system, for instance based on Reo channels [Arb04].

The formal structure of a coordination pattern is consequently

$$\rho \subseteq 2^{\mathcal{C}} \times 2^{\mathcal{N}},$$

where $2^{\mathcal{C}}$ is a set of channels and the $2^{\mathcal{N}}$ is a set of nodes, *i.e.*, a partition on the union of all ends of the channels composing the pattern. It comprises all input and output nodes of the coordination pattern as well as their internal nodes; and it holds information about the connections of channels. The set of all coordination patterns is referred to as $\mathcal{P}$.

In the following, a node with channel ends `{a,b,c}` is denoted as `a.b.c`.

Figure 5 presents two coordination patterns. The coordination pattern (`cp1`), for instance, comprises two channels (a sync channel `x1` and a lossy channel `x2`). The `x1` channel has an input end `a` and an output end `b`. The `x2` channel has an input end `c`, and an output end `d`. Since the channels `x1` and `x2` are connected through the ends `b` and `c`, in the nodes partition a node `b.c` arises.

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

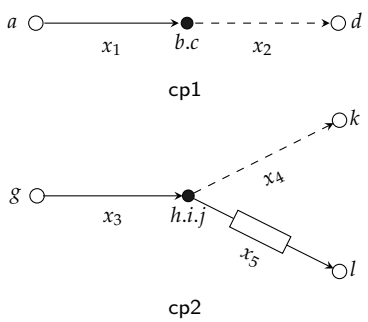


Figure 5: Two simple coordination patterns.

3.5.2 *Coordination-based reconfigurations*

A reconfiguration is a modification of the original structure of a coordination pattern obtained through a sequence of parameterised 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 as is, and the latter replaces the original coordination pattern by ρ.

The `par(`ρ`)` primitive, where ρ is a coordination pattern, sets the original coordination pattern in parallel with ρ, without creating any connection between them. Without loss of generality, nodes and channel identifiers in both patterns are disjoint. Figure 6 presents the resulting coordination pattern, after applying `par(cp2)` to `cp1`.

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

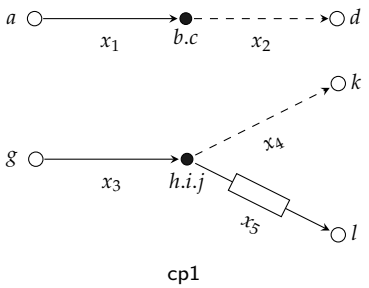


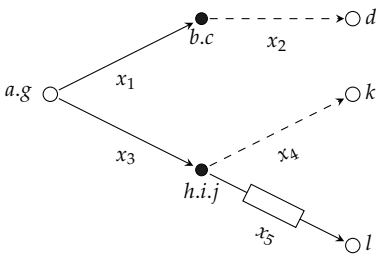
Figure 6: Resulting coordination pattern after applying the `par` primitive.

The `join(`E`)` primitive, where E is a set of ends, creates a new node by merging all ends within E, into a single node in the nodes partition. The created node is a new input or output port if all the ends of the given set are, respectively, input or output ports of the coordination pattern. For instance, applying `join(a,g)` to `cp1` (*c.f.*, Figure 6) creates node `a.g`, as presented in Figure 7.

The `split(`n`)` primitive, where n ia a node, is the opposite of `join` primitive because it breaks connections within a coordination pattern by separating all channel ends coincident in n. Figure 8 presents the resulting coordination pattern, after applying `split(h.i.j)` to the `cp1` from Figure 7. Notice that this reconfiguration primitive only affects – similarly to previous primitives – the nodes partition of the coordination pattern. In particular, it separates the node `h.i.j` into three nodes (`h`, `i` and `j`).

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 at the nodes partition as much as it happens with the `split`. Figure 9 presents the resulting coordination pattern, after applying `remove(x2)` to `cp1` from

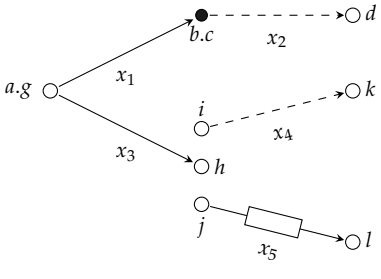
```
cp1=(
    {(a, x1, sync, b),
    (c, x2, lossy, d),
    (g, x3, sync, h),
    (i, x4, lossy, k),
    (j, x5, fifo, l)},
    {a.g, b.c, d, h.i.j, k, l}
)
```



cp1

Figure 7: Resulting coordination pattern after applying the `join` primitive.

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

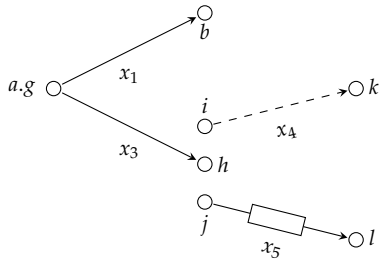


cp1

Figure 8: Resulting coordination pattern after applying the `split` primitive.

Figure 8. Notice how node `b.c` was split and its composing end `c`, was removed with channel `x2`.

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



cp1

Figure 9: Resulting coordination pattern after applying the `remove` primitive.

The application of these reconfiguration primitives upon a coordination pattern does not leave isolated nodes. By applying two reconfigurations in sequence, the structure assumed to exist in the coordination pattern for the second reconfiguration primitive may completely disappear after applying the first, what would lead into reference problems. Theoretically, if the arguments of each reconfiguration primitive are unrelated to the nodes or channel names in the coordination pattern, its application should fail. However, the semantics of such primitives, do not express any kind of restriction on the parameters and thus in these cases, reconfiguration primitives may behave as the identity.

To define more complex reconfigurations, the primitives can be composed. These reconfigurations are called reconfiguration patterns. A reconfiguration pattern can be seen as a sequence of elementary reconfigurations, *i.e.* primitives, which affect significant parts of a connector, are generic and reusable and focused on the interaction protocols. Figure 10 shows a set of reconfiguration patterns that were first introduced in [OB13b].

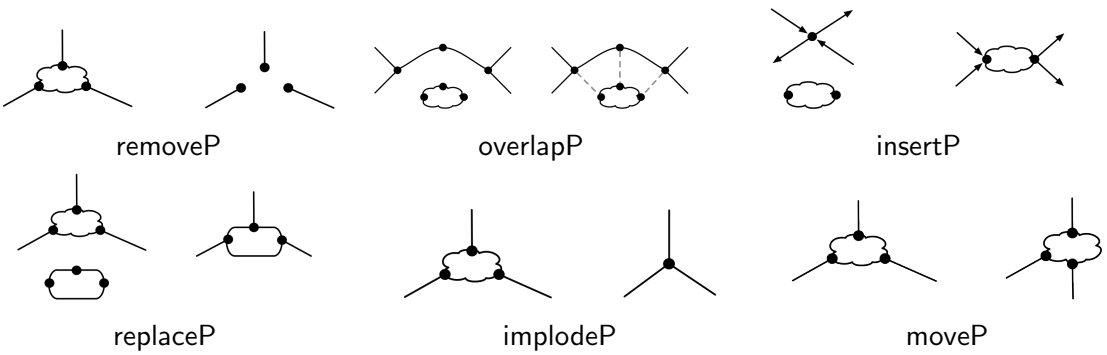


Figure 10: Reconfiguration patterns.

The `removeP(`C`)` pattern, where C is a set of channel identifiers, removes from a coordination pattern each one of channel identifier within C, by successively applying the remove primitive.

The `overlapP(`ρ, X`)` pattern, where ρ is a coordination pattern and X is a set of pairs of nodes, overlaps two coordination patterns by joining nodes from both of them. The ρ is set in parallel with the original coordination pattern. Then, the nodes of each pair in X, are joined. Each pair of X is composed of a node of the original coordination pattern and another of ρ.

The `insertP(`ρ, n, m_i, m_o`)` pattern, where ρ is a coordination pattern and n, m_i and m_o are nodes, puts both coordination patterns side by side, *i.e.*, ρ in parallel with the original coordination pattern. Then, it uses `split` to make room for a new coordination pattern to be added. Finally, it uses `join` to re-build connections. In particular, after splitting n of the original coordination pattern, all the resulting output ports are joined with the m_i, which is an input node of ρ, and the resulting input ports are joined with the m_o, which is an output node of ρ.

The `moveP(`c, n_i, n_f`)` pattern, where c is a channel identifier and n_i and n_f are nodes, moves a single end of c, corresponding to n_i, to a different node, n_f, in the coordination pattern.

The `replaceP(`ρ, C, X`)` pattern, where ρ is a coordination pattern, C is a set of channel identifiers and X is a set of pairs of nodes, replaces a sub-structure of the original coordination pattern by a new one. For this, the sub-structure corresponding to C, is firstly removed and then, the new coordination pattern is overlapped, *i.e.*, `overlap(`ρ, X`)` is applied.

In its turn, the `implodeP(`C, N`)` pattern, where C is a set of channel identifiers and N is a set of nodes, collapses a sub-structure corresponding to C, into a new single node. For this, firstly, the channels are removed, invoking `removeP(`C`)` pattern, and then the nodes are joined, applying `join(`N`)`.

4

RECOOPLA: THE RECONFIGURATION LANGUAGE

This chapter presents the conceptual and formal description of a domain-specific language — ReCooPLa — to design coordination-based reconfigurations. Moreover, it discusses the relevant technologies to support its development.

The main objective of ReCooPLa is to support the formal model presented in Section 3.5 and make it a suitable tool for the working software architect.

Domain Specific Languages (DSLs) [vDKV00, MHS05, OPHdC09] are languages focused on a particular application domain, used to bridge the gap between programming and specific universes of discourse. Their level of abstraction is tailored to the specific domain, allowing for embedding high-level concepts in the language constructs, and hiding low-level specificities under their processors. Moreover, 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 [KOM+10].

In this way, the language described in this chapter provides a formal and high-level interface for the envisaged reconfiguration engine, abstracting away reconfiguration-based details.

4.1 CONCEPTUAL DESCRIPTION

ReCooPLa was designed to describe the coordination-based reconfigurations referred in Chapter 3.5. Thus, the main concept of this language is that of a *reconfiguration*. ReCooPLa is comparable to other programming languages. We assume reconfiguration to be a first call concept in the language, as much as the function concept is in other programming languages. In fact, they share characteristics: both have a signature (identifier and arguments) and a body which assigns a specific behaviour. But in particular, a reconfiguration is always applied to, and always returns a, coordination pattern (as defined in Section 3.5).

Reconfigurations accept arguments of the following data types: *Name*, *Node*, *XOR*, *Set*, *Pair*, *Triple*, *Pattern* and *Channel*, each one with its specificities.

The reconfiguration body is a list of different sorts of instructions. We devote special attention 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.

To support the application of reconfigurations, the language counts on other constructs that mainly manipulate the parameters of each reconfiguration. In concrete, it provides means to declare and assign local variables. Field selectors, specific operations for structured data types and common Set operations (union, intersection and subtraction). Moreover, a control structure is provided 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 FORMAL DESCRIPTION

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

RECONFIGUATION

A reconfiguration (formally presented in Listing 4.1) is expressed by: the reserved word `reconfiguration`; an identifier representing the name of the reconfiguration that matches the usual regular expression for this kind of terminals; a list of arguments, which may be empty; and a reconfiguration body, which is a list of instructions as explained later in this section. In particular, each argument is aggregated by data type (*i.e.*, data types are factored), unlike conventional languages, where data types are replicated for every different argument.

```
reconfiguration
  : 'reconfiguration' ID '(' args* ')' '{' instruction+ '}'
```

Listing 4.1: EBNF notation for the *reconfiguration* production.

The construct for a reconfiguration is given by: $rcfg(n, t_1, a_1, \ldots, t_k, a_n, b)$, 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: primitives (*Name*, *Node* and *Xor*), 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 a primitive data type, is internally seen as a set of names, to maintain compatibility with its definition in Section 3.5. *XOR* is a particular case of *Node*, which has at least one input end and two (mutual exclusive) output ends. The generic data types are based on the Java generics, therefore it is necessary to give a type to their contents, as can be seen in Listing 4.2.

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

Listing 4.2: EBNF notation for the *datatype* production.

The structured data types have an internal representation as shown in Figure 11.

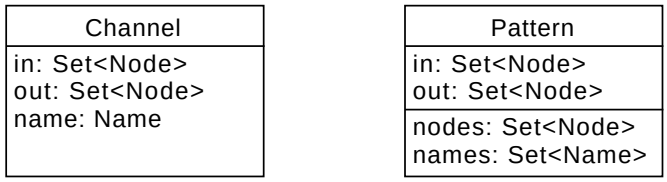


Figure 11: Internal representation of structured data types.

This notation follows the traditional approach of UML class diagrams, *i.e.* the top part contains the name of the structured data type, the middle part contains its attributes and the bottom part mentions its operations. 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 inclosed between curly brackets. An 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 marked by the reserved word `forall`, is used to iterate over a set of elements. Adopting a notation similar to Java, it requires a variable (of data type *Set*) to iterate over, and the declaration of a local variable that in each iteration assumes the value of each element in the provided set. Clearly, the data type of this variable shall be compatible with the data type of the elements inside the set. Again, a list of instructions defines the behaviour of this control structure.

In Listing 4.3 it can be seen the corresponding production rule. For a concrete account of how this production derives, the reader is referred to the example given in Listing 4.8.

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

Listing 4.3: EBNF notation for the *forall* production.

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 4.4), is expressed by an identifier (to which a reconfiguration is applied) followed by '`@`' operator and a reconfiguration name. The latter may be a primitive reconfiguration or another reconfiguration previously declared.

The `'@'` operator stands for *application*. A reconfiguration is applied to a variable of type *Pattern*. In particular, we can omit the variable (optional identifier in the production `reconfiguration_apply`) when we want to refer to the pattern to which the reconfiguration being defined is applied. This typical usage can be seen in Listing 4.8.

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

Listing 4.4: EBNF notation for the *reconfiguration_apply* production.

Application is used either as $@(c)$ or $@(p, c)$, where p is a *Pattern* and c a reconfiguration call. Each reconfiguration call also has its own construct: $r(a_1, \ldots, a_n)$, for r a reconfiguration name, and each a_i one of its arguments.

OPERATIONS

An expression is composed of one or more operations. These can be specific constructors for generic data types, including nodes, or operations over generic and structured data types. Listing 4.5 shows these types of operations.

Each constructor is defined as a reserved word (`S` stands for *Set*, `P` for *Pair* and `T` for *Triple*; and a list of values that shall agree to the data type. The corresponding production is given in Listing 4.5 and exemplified in Listing 4.6.

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

Listing 4.5: EBNF notation for the *constructor* production.

```
    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);
```

Listing 4.6: *Constructors* input example.

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

For the Set data type, ReCooPLa provides the usual binary set operators: '+' for union, '-' for subtraction and '&' for intersection. For the remaining data types (except *Node*, *XOR* and *Name*), selectors are used to apply the operation, as shown in Listing 4.7 (production rule `operation`). Symbol `#` is used to access a specific channel from the internal structure of a pattern.

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

Listing 4.7: EBNF notation for the *operation* and *attribute_call* productions

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*. They are 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 a 0-indexed 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.
- `nodes`: returns all input and output ports plus all the internal nodes of a *Pattern* variable.
- `names`: returns all channel identifiers associated to a *Pattern* variable.
- `fst`, `snd`, `trd`: 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 constructs. The field selection construct is $\bullet(v, c)$, where v is a variable and c a call to an operation. The construct of the '#' operator is $\#(p, n)$, where p is a pattern and n is a channel identifier. The constructs for the set operators follow a similar definition: $+(s_1, s_2)$, $-(s_1, s_2)$ and $\&(s_1, s_2)$, for union, difference and intersection, respectively, with s_1, s_2 being variables of the sort *Set*. The constructs for the other operators are generalised as either $oper(a)$ or $oper()$, depending whether the operation with name *oper* has an argument a or not.

Listing 4.8 shows an example of valid ReCooPLa sentences. Therein, two reconfigurations are declared: `removeP` and `overlapP`. The former removes from a coordination pattern an entire set of channels by applying the `remove` primitive repeatedly. The latter sets a coordination pattern in parallel with the original one, using the `par` primitive, and performs connections between the two patterns by applying the `join` primitive with suitable arguments. These reconfiguration patterns are the implementation of the informal definitions presented in Section 3.5.2.

```
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);
  }
}
```

Listing 4.8: ReCooPLa input example.

MAIN

Finally, ReCooPLa allows for the specification of the actual application of reconfigurations to coordination patterns. This is expressed in a special reconfiguration marked with the reserved word `main`.

The main reconfiguration accepts a (possibly empty) list of arguments aggregated by data type, as in a normal reconfiguration. The difference is that in the main reconfiguration, data types are only references to available coordination patterns expressed in imported CooPLa files. The arguments are assumed as new instances of the given patterns that are to be reconfigured. These instances are defined with default stochastic information; and can not

match an existing identifier of a stochastic instance declared in the imported CooPLa files. An exception is the `Empty` pattern, which is a structureless pattern whose instance(s) can be used in the body of the main to construct new patterns from nothing.

Listing 4.9 presents a partial grammar for the syntax of these main reconfigurations.

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

Listing 4.9: EBNF notation for the *main* production.

The construct for the main reconfiguration is (a special case of the construct for the normal reconfiguration) given as $main(cp_1, a_1, \ldots, cp_k, a_n, b)$, where each a_i is an argument (instance of a coordination pattern) of type cp_i; and b is the body of the main.

The body of the main reconfiguration is a list of specific instructions, where an instruction is either an assignment or an isolated application of a reconfiguration. Listing 4.10 presents the grammar for the syntax of the body of the main.

```
main_instruction
  : main_assignment | reconf_apply
main_assignment
  : t=ID v=ID '=' p=ID '@' reconfiguration_call
reconf_apply
  : ID '@' reconfiguration_call
```

Listing 4.10: EBNF notation for the *main_instruction* production.

An assignment in the main is a declaration, expressed as usually, with a data type and a variable identifier, followed by a concrete application of a reconfiguration to the coordination pattern instances (which are either passed as arguments, imported, or freshly declared). The data type corresponds to a coordination pattern name, which may or may not exist in the imported ones. If the pattern name does not exist, it becomes a new coordination pattern and the variable identifier is added as its instance; otherwise the structure assigned to the new instances is verified to ensure that it matches the expected coordination pattern. If the structures are similar, then the identifiers are added as instances of the pattern; otherwise they become instances of the multi-purpose pattern `Reconfigured`. The construct for this specific instructions is $assign_m(t, v, p, r, e_1, \ldots, e_n)$ where t is a data type (coordination pattern), v a variable name, p another coordination pattern (usually an argument of the main reconfiguration), r a reconfiguration call with each e_i one of its suitable argument.

An isolated application of a reconfiguration, directly changes the target instance of coordination patter and associates it to the generic type `Reconfigured`, and remove it from its original pattern. The `@` symbol is again used to express the application of reconfigurations.

The arguments of a reconfiguration are obtained from the arguments of the main and freshly declared pattern instances, using the operations explained above to access nodes, channels and alike. The construct for this specific instructions is $reconf_m(p, r, e_1, \ldots, e_n)$, where p is a coordination pattern (an argument of the main reconfiguration or a freshly declared pattern instance), r a reconfiguration call with each e_i one of its suitable argument.

A simple example of a main reconfiguration is presented in Listing 4.11. Therein, a `Sequencer` coordination pattern is reshaped through the application of the `removeP` reconfiguration (previously presented in Listing 4.8). Notice both the import of the files where patterns and reconfigurations were previously defined.

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

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

Listing 4.11: Main reconfiguration in ReCooPLa

4.3 RECOOPLA PROCESSOR

This section presents the development of the language processor. It follows the traditional approach in compiler construction [ASU86]: the parser, the semantic analyser and the translator. Suitable language engineering technologies (*c.f.*, ANTLR and StringTemplate) were applied to this end, which are briefly discussed next.

4.3.1 *Technologies*

ANother Tool for Language Recognition (ANTLR) [Par07] is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.[1] It is widely used in academia and industry to build all sorts of languages, tools, and frameworks.[2] The following are some representative examples of companies using ANTLR: *Apple* and *IBM* use ANTLR for their high-profile projects; *Twitter search* uses ANTLR for query parsing, with over 2 billion queries a day; *Checkmarx* uses ANTLR to parse a large set of languages, which are the object of queries to check for security properties of the code; *Oracle* uses ANTLR within SQL Developer IDE and their migration tools; *NetBeans IDE* parses C++ with ANTLR; the *HQL language* in the *Hibernate* object-relational mapping framework is built with ANTLR.

1 http://www.antlr.org/about.html
2 http://www.antlr3.org/showcase/list.html

Moreover ANTLR is powerful, flexible, well documented and actively supported. It supports the development of traditional compilers allowing a deep separation of concerns. A plus is the possibility of using attribute grammars that formalise all the development process. Competitor tools [Joh75, Kod04, HVM+05] fail to provide the features and the easiness of use that AntLR provides. Based on this, we chose AntLR for developing ReCooPLa and its processor.

Attribute grammars were first developed by Donald Knuth in 1968 to formalize the semantics of a context-free language [Knu90]. Thus, an attribute grammar is a Context-Free Grammar (CFG) that has been extended to provide context-sensitive information by adding attributes, computation and translation rules, and contextual conditions. Context-sensitive aspects of syntax of a language, such as check if an item has been previously declared, can be specified using an attribute grammar, in particular, by using contextual conditions that check the semantic validity of the concrete sentence. Attribute grammars can also be used to translate – by using translation rules – the syntax tree directly into code for some specific machine, or into some intermediate language, only if the semantics is valid. Each distinct symbol in the grammar has associated with it a (possibly empty) set of attributes. In turn, each attribute has a domain of possible values and can be evaluated in assignments or conditions.

Attributes can be classified as inherited or synthesised. The former transports contextual information down the derivation tree and its value at a node in a tree is defined in terms of attributes at the parent and/or sibling of that node. Thus, terminal symbols and the axiom do not have inherited attributes. On the other hand, the latter synthesises information from the "leafs" in the derivation tree and transports it up the tree. Thus, its value at a node in a tree is defined in terms of attributes at the child of that node. The terminal symbols attributes, for instance, which are intrinsic and pre-established, are seen as synthesised attributes. In short, attribute grammars can pass values from a node to its parent, using a synthesised attribute, or from the current node to a child, using an inherited attribute.

4.3.2 *Development*

After the design of the language, it is necessary to create a compiler for it that is efficient and modular. A compiler is needed for the translation of the language into a form in which it can be executed. To achieve this objective we followed the traditional approach for compiler construction [ASU86], by developing the necessary components (Lexer, Parser, Semantic Analyser and Translator) to obtain the necessary executable format. These components were developed with ANTLR. In Figure 12 the compiler scheme, comprising all of these components, is depicted.

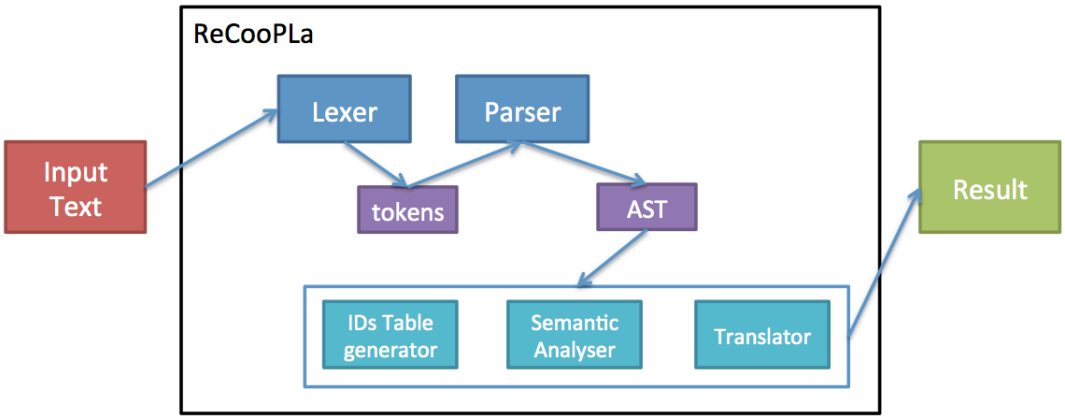


Figure 12: The compiler scheme.

Lexer

The *Lexer*, the first component of the compiler, reads the stream of characters of "Input Text" and groups them in meaningful sequences called *lexemes*. For each recognised lexeme, the *Lexer* produces a token, by adding extra information like the line and the position in which the lexeme appears in the text, among other useful data. Its output is a sequence of tokens. This phase is traditionally called the lexical analysis.

Parser

The *Parser*, reads the sequence of tokens produced by the *Lexer* and check if they are in the right order by using the `LL(k)` algorithm, which is a top-down approach for recognising the input starting from the axiom of the grammar with the possibility of looking to `k` tokens ahead so it can decide which grammar production to follow. It generates a tree-like representation that depicts the grammatical structure of the token stream (*i.e.*, the syntax tree). In order to enhance the work and separate concerns, the parser was instructed to output an intermediate representation: the Abstract Syntax Tree (AST).

Conceptually, an AST is obtained by removing the syntactic sugar that does not affect the information retrieval from the input, increasing the efficiency and the ability to separate concerns, through the construction of tree walkers increasing also the semantic legibility of the language.

Formally, to obtain a AST, first is required to define in `options` section of the parser that it will outputs an AST, as shown in Listing 4.12. Thus, the parser will create also `CommonTree` tokens that can have parent and child tokens instead of the default tokens of type `CommonToken`.

```
parser grammar RecParser;

options{
  tokenVocab=RecLexer;
  output=AST;
}
```

Listing 4.12: Parser `options` section.

After add the output option, an AST can be created by using tree operators or rewrite rules. The former are used to define a certain node as root or to exclude a certain node from the tree by using the "`^`" and "`!`" symbols, respectively. Listing 4.13 presents an example where a production `datatype` is annotated using tree operators to change the output tree. In this example, the `SEP_START` and `SEP_END` are separators matching "<" and ">" symbols, respectively. They are both exclude from the tree since they are syntactic sugar that does not affect the information retrieval from the input. Moreover, `other_type` becomes the root of the sub-tree referring to the parser rule/production.

```
datatype
  : ...
  | other_type^ SEP_START! subtype SEP_END!
```

Listing 4.13: Tree operators example.

On the other hand, rewrite rules can be placed at the end of each derivation in a production. A rewrite rule is denoted by "`->`" followed by `^(...)`, where the first token inside the parenthesis will become the root of a sub-tree. In this approach, all tokens that are not included in the rewrite rule will be removed from the AST. The rewrite rules are more flexible since they allow, for instance, the rearranging of tokens, children of root, in a parser rule, which is not possible using tree operators. Listing 4.14 presents an example where a same production `datatype` is annotated using rewrite rules to create the same output tree that would be created using tree operators in Listing 4.13.

```
datatype
  : ...
  | other_type SEP_START subtype SEP_END -> ^(other_type subtype)
```

Listing 4.14: Rewrite rules example.

If a parser rule has not any node suitable to promote root, a new token can be created for this task. Thus, is possible to maintain the coherence and readability of the output AST. First it is necessary to add a `tokens` section on the parser, right below the `options` section and above the `header` section.

```
tokens {
  RECONFIGS;
  IMPORT;
  RECONFIGURATION;
  ARGUMENTS;
  INSTRUCTIONS;
  DECLARATION;
  ASSIGNMENT;
  FORALL;
  ...
}
```

Listing 4.15: Tokens section in parser.

Then, the tokens previously defined can be used in rewrite syntax as shown in Listing 4.16.

```
declaration
  : datatype var_def (',' var_def)*
  -> ^(DECLARATION datatype var_def+)
  ;

assignment
  : ID '=' assignment_member
  -> ^(ASSIGNMENT ID assignment_member)
```

Listing 4.16: Rewrite rules using created tokens.

In Figure 13, an AST for the `removeP` reconfiguration pattern (*c.f.* Listing 4.8) is presented.

The generated AST can be then used by tree grammars in ANTLR. Essentially, a tree grammar defines a walker that travels the glsast. These walkers were used in the construction of the ReCooPLa compiler to separate code of multiple components like the Semantic Analyser, or the Translator. With this approach, the compiler can be further extended with extra functionality without the need for understanding or changing code within other components.

In a tree grammar, first should be declared the location of the tokens to be used and the type of tree-tokens that it can expect on the `options` section, as shown in Listing 4.17. The former is achieved by the set of the option `tokenVocab = RecParser`. This tokens match the ones presented in Listing 4.15, in `tokens` section in parser. The latter is achieved by the set of the option `ASTLabelType = CommonTree`.

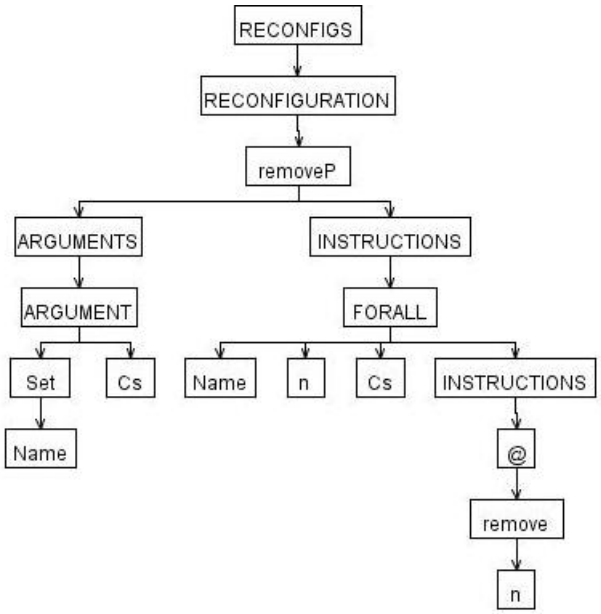


Figure 13: The *removeP* AST.

```
tree grammar RecIDTable;

options{
  tokenVocab = RecParser;
  ASTLabelType = CommonTree;
}
```

Listing 4.17: Tree grammar `options` section.

Three walkers were defined in the project which correspond to three components depicted in Figure 12 and detailed next.

Ids Table Generator

The *Ids Table Generator* component was developed to obtain a table of identifiers found in the input text. This component was separated from the semantic analysis for a proper separation of concerns. Thus, the semantic analysis itself –presented later in this section– is simplified.

The natural approach to an identifiers table is to use a stack to save symbols. However, this approach would not be feasible since should also be kept a hierarchical structure between the scopes were the symbols are saved, in order to, for instance, ascertain if a variable has already been declared.

We designed an Identifiers Table as a Map of identifiers to symbols. In this context, we consider a symbol to be one of reconfiguration name, argument of the reconfiguration or declared variable. Each *symbol* has some attributes to store the identifier information such

as, for instance, the respective identifier name, data type (*e.g.*, Pattern, argument, variable, etc), as well as the line and position on the input file. If the identifier corresponds to a reconfiguration name, as in Listing 4.1, the *symbol* has also an attribute which corresponds to a list of tables where all the symbols within the reconfiguration are stored. The first element of this list of tables stores the arguments and the declared variables, in the top-most scope. The remaining elements stores the symbols of the iterative control structures (a table for each control structure) within the reconfiguration. Each one of these (internal) tables comes together with a pair of numbers: a current scope identifier, which is incremental for keeping an order between tables; and the parent scope identifier, kept for ensuring a hierarchical structure between identifiers tables and scopes. It goes without saying that these identifiers are reset at each new reconfiguration symbol since this information is only useful in its context. Listing 4.18 presents the identifier table for the `removeP` reconfiguration, as generated by the Ids Table Generator component.

```
id_table = {
    [removeP -> id: removeP,
        datatype: [PATTERN],
        classType: RECONFIG,
        line: 1,
        position: 16,
        file: reconfig.rpla,
        scopes: {
            [Cs -> id: Cs,
                datatype: [SET, NAME],
                classType: ARG,
                line: 1,
                position: 35
            ],(0,0);
            [n -> id: n,
                datatype: [NAME],
                classType: VAR,
                line: 2,
                position: 14
            ],(1,0)
        }
    ]
}
```

Listing 4.18: Identifiers table for *removeP* reconfiguration.

Semantic Analyser

The semantic analyser component uses the AST generated by the parser and the information in the the identifiers table to check the input text for semantic consistency with the desired

semantics for ReCooPLa. If something is not as expected, an error is created and added to a list of errors that is to be used further in the development of the ReCooPLa supporting tools.

The walker associated to this component implements an attribute grammar so that contextual conditions more explicitly define the meaning of the language. Contextual conditions define constraints on the language that the syntax, by itself, can not recognise. The context is given by the location in the AST where the walker is at a given processing phase, and by the accessible resources therein. Such resources can be incoming attributes (synthesised or inherited) or other constants, used in computation rules of the underlying attribute grammar.

However, in some semantic analysis as in this case where multiple attributes are needed, the complexity that comes from passing the attributes by the tree quickly increases. To overcome this issue, in some productions a section `scope` was added. This makes the work easier since the attributes declared in such scope are accessible to all children of that production, without having to be pushed down the tree to the place where the computation rule or contextual condition is defined.

In Listing 4.19 is presented an example where an attribute `current_scope` is defined in the `scope` of `content` production and computed using a given computation rule only at `instruction` production. The typical usage is `$name_of_production::attribute`, as can be seen in Listing 4.19.

```
content
scope{ SymbolsTable current_scope; }
  : reconfiguration* main?
  ;

(...)
instruction
@init{ $content::current_scope = this.getScope($reclang::scopes.get(id)); }
  : declaration | assignment | reconfiguration_apply | forall
  ;
```

Listing 4.19: Example of the scope of a production.

The value of attribute `$current_scope` is obtained from the identifiers table, generated in the first tree walker, which is an input of the semantic analyser. This is accomplished through the use of a method (`getScope(Integer id)`), wherein `id` is an element of the list of scope identifiers filled in while the tree is traversed. The attribute is then used in some contextual conditions to check the semantic validity of the concrete sentences of the input text. Listing 4.20 presents an example where this happens, using attribute `$current_scope`. In this example, the semantic analyser checks, for each declared symbol in that location, if it is found in the current scope, has expected. Then, another contextual condition checks if its position is the same as the stored in the identifiers table. If it is not, it means that a variable

has been declared with the same name as another previously declared. Therefore, an error is reported and added to the list of errors.

```
declaration returns[ArrayList<Error> errors]
@init{ ArrayList<Error> local_errors = new ArrayList<Error>(); }
  : ^(DECLARATION datatype (ID
  {
    if ($content::current_scope.containsSymbol($ID.text)) {
      Symbol s = $content::current_scope.getSymbols().get($ID.text);
      if ( !($ID.line == s.getLine() && $ID.pos == s.getPosition()) ) {
        local_errors.add( Error.report(ErrorType.ERROR,
            Error.nameAlreadyDefined( ... ) );
      }
    }
  )+
  )
  { $declaration.errors = local_errors; }
  ;
```

Listing 4.20: Error reporting example.

The example in Listing 4.20 reflects part of the approach adopted to handle errors: Report, Recover and Resume (RRR). By using this strategy, at first the error is identified and reported conveniently. Then, an attempt is made to retrieve the processing in order to avoid, for instance, cascading errors. This reduces the size of the list of errors and focus the developer on the key errors. Finally, the analysis and processing of the input text are safely continued.

The error messages should be as clear as possible, in order to clearly identify the problem and its severity (error, warning, ...), what is causing the problem and also its location, as accurately as possible. In the example in Listing 4.20, the reported error would give rise to a message such the one in Listing 4.21

```
"<TIME> ERROR file.rpla 5:3 >> "Name 'var' is already defined at line 2:9!"
```

Listing 4.21: Error message example.

There are several types of semantic errors identified by this walker in the context of the semantic analysis, such as a variable/reconfiguration name already defined, an assignment of a variable that is not yet defined, an element of a set with a wrong data type and an attribute used by a variable of an incorrect type. The semantic errors found are not all related to types of variables. Examples of this are an incorrect number of arguments of a reconfiguration, use of a reconfiguration that was not previously defined and an import of a file that does not exist or is not valid.

The syntactic erros are also collected in a similar way, by the parser, using the same data structure than the semantic analyser. These errors correspond to the existence of chains of

tokens that are not recognised by the language. For instance, if a semicolon is missing at the end of an instruction, a syntactic error will occur.

By itself, ANTLR provides a powerful mechanism for detecting and recovering from an error, whether lexical or syntactic, so that all text input is consumed. By default, ANTLR throws the error messages to *stderr*. For that, it provides a method (`emitErrorMessage()`) that originally receives an error message and throws it to the *stderr*. However, since it is intended to collect these errors to a data structure and merge it with the errors coming from the semantic analyser (for further usage), this method was overridden.

To achieve this, within the action `@members` on the parser, the `emitErrorMessage` method is rewritten, as presented on the Listing4.22.

```
@members{
  private ArrayList<Error> syntax_errors = new ArrayList<Error>();

  @Override
  public void emitErrorMessage(String msg) {
        syntax_errors.add(Error.report(ErrorType.ERROR, msg, file_path));
  }

  public ArrayList<Error> getErrors() {
      return this.syntax_errors;
  }
}
```

Listing 4.22: `emitErrorMessage` method rewriting.

In the Listing 4.22, first a list of `syntax_errors` is declared as a private attribute of the class (the parser of the language). Then, between lines 4 and 7 the rewritten `emitErrorMessage` method takes a `String` as an argument corresponding to the error message and add it to the list of errors, in the proper format, also used in the semantic analysis. Finally, between lines 9 and 11 a method to access the `syntax_errors` attribute is declared, keeping the encapsulation of the generated class. Thereby, a list of syntactic errors, which is then accessible outside of parser (for instance, in the reconfiguration engine) and compatible with the list of semantic errors, is obtained.

If there are no errors during the syntactic and semantic analysis, the ReCooPLa processor should proceed to the language translation and code generation.

5

THE RECONFIGURATION ENGINE

This chapter introduces the reconfiguration engine, which executes reconfigurations specified in ReCooPLa. In particular, it goes through a conceptual description of both the model of the engine, and the associated ReCooPLa translation schema. The actual implementation of the engine is delivered along with the architecture and details on the implementation of an integrated editor for ReCooPLa.

5.1 TECHNOLOGIES

Due to the possibility of integrating this work with Reo Tools [AKM+08] the Eclipse platform was chosen as the IDE on which the reconfiguration engine was to be developed. Consequently, Java was the chosen programming language because the Eclipse plugin API is supported only for Java. But being forced to use both Eclipse and Java is not a bad thing. Their multi-platform availability is actually a good characteristic that opens possibilities for more people using our system. Moreover, Java is an object-oriented programming language, fast, secure and reliable. It is largely adopted both in academia and industry. Documentation and support are available everywhere. In turn, Eclipse originally created by IBM in November 2001, is a project focused on providing an open development platform. It is an Open Source IDE, mostly provided in Java, but the development language is independent and can be extended by tools and plugins. The Eclipse Foundation, created in January 2004, is a member supported corporation that hosts the Eclipse Project and helps cultivating both an open source community and an ecosystem of complementary products and services.[1]

StringTemplate is a code generator that emits text using templates. In turn, a template is basically a "document with holes", where values called attributes can be set. An attribute can be an object, a variable, a template instance or a sequence of attributes, and it is, by default, enclosed in angle brackets. The remaining text outside of attribute expressions is ignored by *StringTemplate* and assumed as regular text to print out.

1 http://www.eclipse.org/org/

StringTemplate is designed to be embedded inside other applications and is distributed as a small library with no external dependencies apart ANTLR. A template can be directly created in code, loaded from a directory or loaded from a file containing a collection of templates called *template group file* (`stg`). The latter was the chosen approach and thus the templates were all specified on a single template group file and then given as input of the translator, to streamline the translation process.

Templates definitions are similar to functions with untyped arguments. Then, the content of the template is defined through a kind of attribution, denoted by "`::=`" and enclosed in double angle brackets once it is a multi-line template. If it was a single line template it would be enclosed in double quotes. All content of the template is processed as regular text except that enclosed in angle brackets, corresponding to the attributes of the template.

5.2 THE ENGINE MODEL

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 [OB13b], 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 14 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 Chapter 3.5. Both CoordinationPattern and Channel classes provide attributes and methods that match the attributes and operations of the *Pattern* and *Channel* types in ReCooPLa. In Table 1 can be seen the complete mapping of the attributes and operation of ReCooPLa structured data types to the corresponding methods of the Reconfiguration Engine classes.

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 Chapter 3.5. 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 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

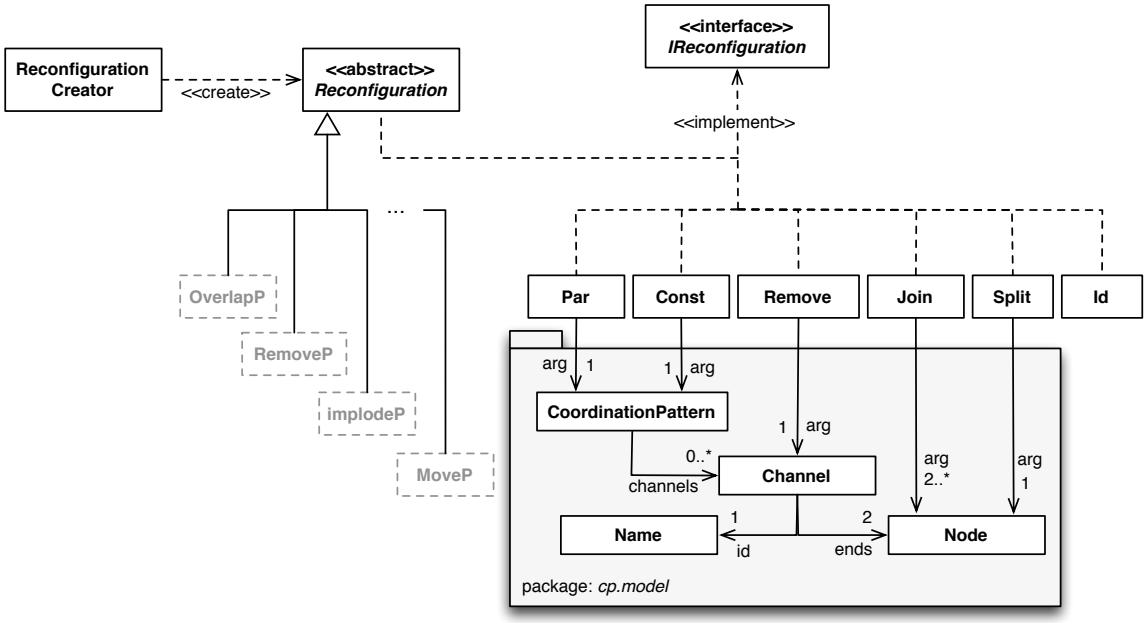


Figure 14: The Reconfiguration Engine model

Table 1: Mapping between the ReCooPLa structured data types and the Reconfiguration Engine classes

ReCooPLa structured data types		**Reconfiguration Engine classes**
Pattern.in	→	CoordinationPattern.getIn()
Pattern.in[i]	→	CoordinationPattern.getIn(int i)
Pattern.out	→	CoordinationPattern.getOut()
Pattern.out[i]	→	CoordinationPattern.getOut(int i)
Pattern.nodes	→	CoordinationPattern.getNodes()
Pattern.names	→	CoordinationPattern.getNames()
Channel.in	→	Channel.getIn()
Channel.in[i]	→	Channel.getIn(int i)
Channel.out	→	Channel.getOut()
Channel.out[i]	→	Channel.getOut(int i)
Channel.name	→	Channel.getId()

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 runtime, by the ReconfigurationCreator class, taking advantage of reflection in Java Virtual Machine (JVM). 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 class with an apply(CoordinationPattern p) method. The content of such method is derived from the content of the ReCooPLa reconfiguration, taking advantage of the translation schema presented in Section 5.3. This creates a Java file for each reconfiguration in the ReCooPLa specification. These files are compiled in runtime, into Java class files and, via reflection, loaded into the running JVM. Each reconfiguration in a ReCooPLa specification yields a new class, which extends the abstract class `Reconfiguration`, whereas the *main* reconfiguration yields a single class `Run`, which uses, via reflection, the reconfigurations previously generated. Thus, the application of reconfigurations becomes as simple as calling, in the the `Run` class, 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 5.3 goes through the details of such a translation.

5.3 RECOOPLA TRANSLATION

This section presents the translation of ReCooPLa into the model of the Reconfiguration Engine. First, an overview of the translation process is provided, where a formal translation scheme is defined for the ReCooPLa constructs. Then details on the translation implementation are given, along with examples of obtained results.

5.3.1 *Translation overview*

Throughout this section, it is assumed the existence of Java classes to match the types in ReCooPLa. This means that, besides the classes already mentioned in Figure 14, 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.

In order to keep exposition simple, some minutiae like imports, semicolons, annotations, auxiliary variables, control or try-catch structures and efficiency concerns are not taken into account, for simplicity sake. Moreover, abstractions are used to wrap complex constructions; for instance, method mkRecfg(n, t_1, a_1, ..., t_k, a_n, b) abstracts details of the creation of a Reconfiguration class with name n; attributes a_1, ..., a_n of type t_1, ..., t_k; and method apply with body b, which always ends with a return p instruction, where p is the argument of apply.

This being said, the translation of ReCooPLa constructors into the Reconfiguration Engine is given by the rule-based function $\mathcal{T}(C)$, where C is a constructor of ReCooPLa as presented in Section 4.2. The definition of $\mathcal{T}()^2$ goes in Table 2.

Table 2: Translation rules for ReCooPLa constructs

$\mathcal{T}(rcfg(n,t_1,a_1,...t_k,a_n,b))$	$\rightarrow$	mkRcfg(n, $\mathcal{T}(t_1)$, a_1, ... $\mathcal{T}(t_k)$, 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($\mathcal{T}(e_n)$); }} [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)
$\mathcal{T}(\bullet(v,c))$	$\rightarrow$	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()
$\mathcal{T}(main(cp_1,a_1,...cp_k,a_n,b))$	$\rightarrow$	mkMain(Run, $\mathcal{T}(cp_1,a_1)$, ... $\mathcal{T}(cp_k,a_n)$, $\mathcal{T}(b)$)
$\mathcal{T}(cp,a)$	$\rightarrow$	CoordPatt a = new CoordPatt(patterns.get(cp)) [4]
$\mathcal{T}(assign_m(t,v,p,r,e_1,\ldots,e_n))$	$\rightarrow$	CoordPatt v; v = $\mathcal{T}(@(r(p,e_1,\ldots,e_n)))$; $S(v,t)$ [5]
$\mathcal{T}(reconf_m(p,r,e_1,\ldots,e_n))$	$\rightarrow$	$\mathcal{T}(@(r(p,e_1,\ldots,e_n)))$; $S(v$,Reconfigured)

A translation can only occur when the ReCooPLa specification is syntactically and semantically correct. The ReCooPLa parser ensures syntactic correctness; on the other hand, the semantic analyser detailed in Section 4.3.2 reports errors concerning structure, behaviour and data types.

2 By convention n is used for identifiers; t, t_i for data types; a_i for arguments; b for set of instructions; T for non-generic data type; T_G for generic data type, except *Set*; v, v_i for local variables; e, e_i for expressions; p for patterns; s_i for sets; c for channel names; i for numbers; and finally *oper* for the operations enumerated in Section 4.2.

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

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

5 $S(v,t)$ abstracts the action of storing variable v as an instance of coordination pattern named t.

5.3.2 *Translation implementation*

The *translator* –the last component of the compiler– uses the AST generated by the parser presented in Section 4.3.2 to translate ReCooPLa specification into Java code. This is done in accordance with the translation schema presented in Table 2 and taking advantage of the identifiers table, generated in previous steps of the ReCooPLa processor. In order to ease the translation, the *StringTemplate* [6] technology (*c.f.*, Section 5.1) is applied.

Listing 5.1 presents the template `mkclass`, which is responsible for defining the complete structure of a Reconfiguration class as already anticipated.

```
mkclass(name, fields, constructor, method) ::= <<
public class <name> extends Reconfiguration {
<fields>
<constructor>
<method>
}
>>
```

Listing 5.1: Template for the Reconfiguration classes.

After the templates are defined, they are used in the translator, in the context of rewrite rules. Listing 5.2 shows how the `mkclass` template is used in the translator component. The arguments of `mkclass` template are of different types: variables (*e.g.*, `class_name`), lists of values (*e.g.*, `$args.values`) and others string templates (*e.g.*, `$instructions.st`). Each argument is obtained in the context of the production reconfiguration, taking advantage of the underlying attribute grammar. Lines 4 and 5 define the name of the class (forcing it to start with capital letter, as required by Java); Line 8 actually directly uses the `mkclass` template with the suitable parameters.

```
reconfiguration
  : ^(ID
  {
    String class_name = Character.toUpperCase($ID.text.charAt(0))
        + $ID.text.substring(1);
  }
  args? instructions )
  -> mkclass(name={class_name}, fields={$args.values},
  constructor={$args.st}, method={$instructions.st})
  ;
```

Listing 5.2: Template usage.

6 http://www.stringtemplate.org/index.html

Finally, a map is created and filled for be able to handle the translation result on the reconfiguration engine. This map has as key the name of the reconfiguration, and as value the content of the translation of that same reconfiguration. Listing 5.3 presents an excerpt from the code written to save the translation result.

```
@members{
  private HashMap<String,String> reconfigurations;
  public HashMap<String,String> getReconfigurations(){
    return this.reconfigurations;
  }
}
(...)
content
  : ( element { reconfigurations.put($element.name, $element.st); } )*
    ( main    { reconfigurations.put($main.name, $main.st); } )?
  ;
```

Listing 5.3: Collection to save the result of the translation.

At the action `@members`, a private attribute `reconfigurations` is added. Then a public method to access this attribute is declared, keeping also the encapsulation of the generated class.

The translation result is further treated by the Reconfiguration Engine, in particular, by a single class `Run`, via reflection. Figure 15 shows the result of applying the translation rules to the *OverlapP* ReCooPLa reconfiguration documented in Listing 4.8.

```
public class OverlapP extends Reconfiguration {

    private CoordinationPattern2 p;
    private LinkedHashSet<Pair<Node,Node>> X;

    public OverlapP(CoordinationPattern2 arg1, LinkedHashSet <Pair<Node,Node>> arg2 ) {
        this.p = arg1 ;
        this.X = arg2 ;
    }

    @Override
    public CoordinationPattern2 apply(CoordinationPattern2 $cp) {
        Par par;
        Join join ;
        par = new Par(this.p);
        par.apply($cp);
        for(Pair<Node,Node> n : this.X) {
            Node n1, n2;
            n1 = n.fst();
            n2 = n.snd();

            LinkedHashSet<Node> E = new LinkedHashSet<Node>();
            E.add(n1);
            E.add(n2);
            join = new Join(E);
            join.apply($cp);
        }
        return $cp ;
    }
```

Figure 15: Example of a ReCooPLa reconfiguration translated.

5.4 A PRACTICAL 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 16. 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. nodes.

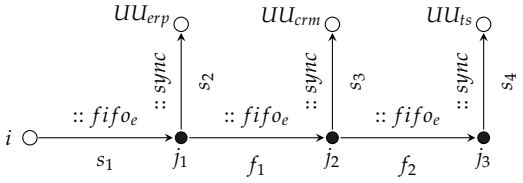


Figure 16: The User update coordination pattern.

However, other configurations were 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 17. 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 5.4, resorting to removeP already defined in Listing 4.8.

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

Listing 5.4: `implodeP` reconfiguration pattern.

This reconfiguration pattern takes as parameters the set of nodes and channels relative to the structure 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 5.5.

```
public class ImplodeP extends Reconfiguration {
    private LHSet<Node> X;
    private LHSet<Name> Cs;
    public ImplodeP(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;
    }
}
```

Listing 5.5: ImplodeP class generated.

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 17, where (for reading purposes) node k is used to represent the union of j_1 and j_2. The coordination pattern presented defines an instance of a parallel pattern, where UU_{erp}, UU_{crm} and UU_{ts} execute in parallel with data entering in port i.

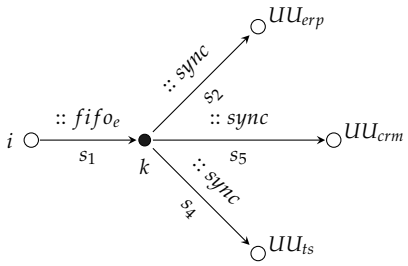


Figure 17: The User update coordination pattern reconfigured.

5.5 Architecture and Development

The Reconfiguration Engine is part of a broader system implemented as an eclipse plugin, comprising language processors, editors and other associated tools.

As expected, one of the processors is the ReCooPLa 's. The other is for processing CooPLa specifications, in order to provide the coordination patterns up on which reconfigurations are applied. Figure 18 presents the high-level system architecture with such components.

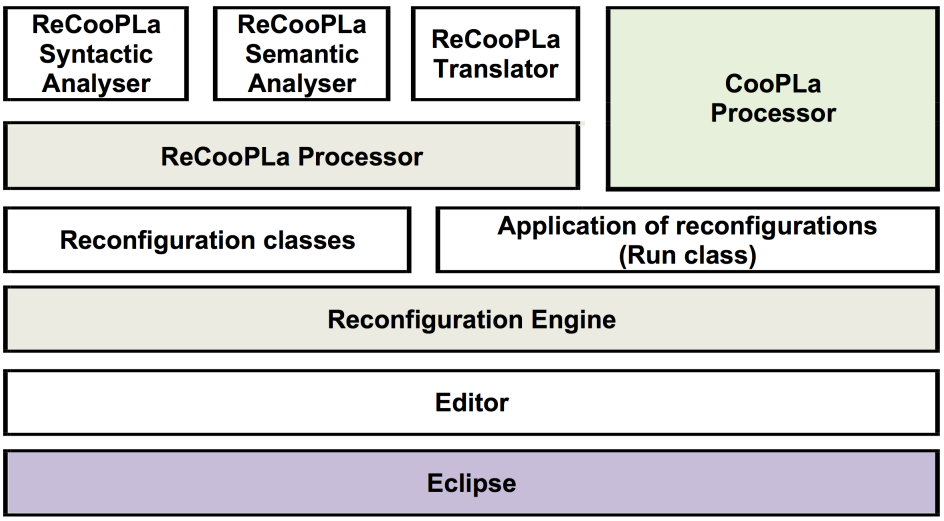


Figure 18: The System high-level architecture.

Let's now focus on the Editor component of the system. It is an Eclipse plugin developed in Java as a plug-in to allow the creation of CooPLa and ReCooPLa files specifications. The ReCooPLa part of the editor borrows features of the ReCooPLa processor to analyse the code, dynamically, and then annotate it with markers pointing to errors detected in the syntactic and semantic analysis. This is accomplished by adopting the reconciling techniques fomented in editor development within Eclipse. The reconciler entity allows for running the ReCooPLa processor in the editor background, in a separated thread, so that the editor does not freeze. The ReCooPLa processor is invoked periodically (*e.g.*, when the user stops coding) and it takes the input text with all the changes performed. The captured (syntactic and semantic) errors are then used by the reconciler to create annotations and problem marks, and add them to the editor view. Figure 19 presents how these are presented by the editor. The errors are pointed out by the annotations and problem markers provides some more details about the error. The annotations can provide also the error message when the user hovers the mouse over the red marks on both sides of the editor.

If there are no errors, the reconfigurations can be executed, according to the main reconfiguration, specified in ReCooPLa. Once this feature of the editor is started, a group of procedures are performed. First of all, before translating the reconfigurations, it is ensured that the specifications are syntactic and semantic valid. Then, the Java files, corresponding to the reconfigurations, are created by taking advantage of the translation obtained through the process presented in Section 5.3. These files are afterwards properly compiled taking into account the required dependencies. After compiling them, the created coordination patterns

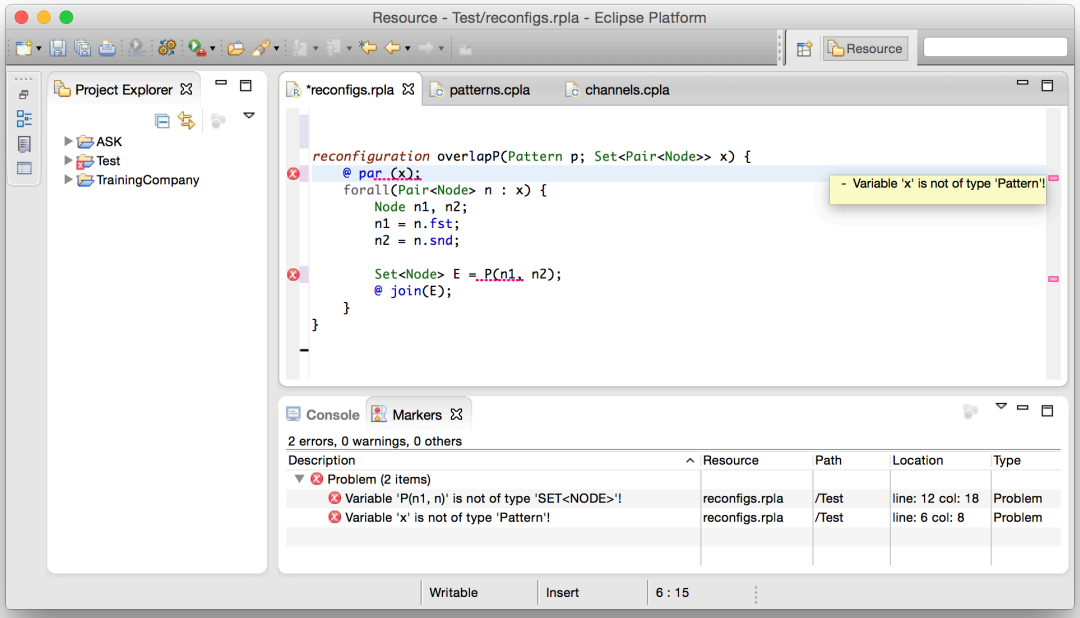


Figure 19: Annotations and problem markers on the editor.

specified in the `main` of ReCooPLa (*i.e.*, the results of the application of the reconfigurations) are obtained. This is achieved using Java Reflection, which is a powerful and useful technique that allows the inspection and modification of the structure and behaviour of an application at runtime as well as the instantiation of a new object, invocation of methods, and some other features.

The generated coordination patterns have a particular data structure that besides their structural information, also contains information about their stochastic instances, the environment and nodes. Each one of these data structures is converted into a graph structure and then sent to an Eclipse View, called "*Patterns Created*". Figure 20 shows the result after a simple reconfiguration, where a simple `Pattern2` with only one `fifo` channel is inserted into an existing one (`Pattern1` which has two `sync` channels).

Additionally, it is given to the user the possibility to save the internal representation of the generated coordination patterns to a CooPLa file in the file system. Figure 21 shows the dialog presented to the user to save the created patterns.

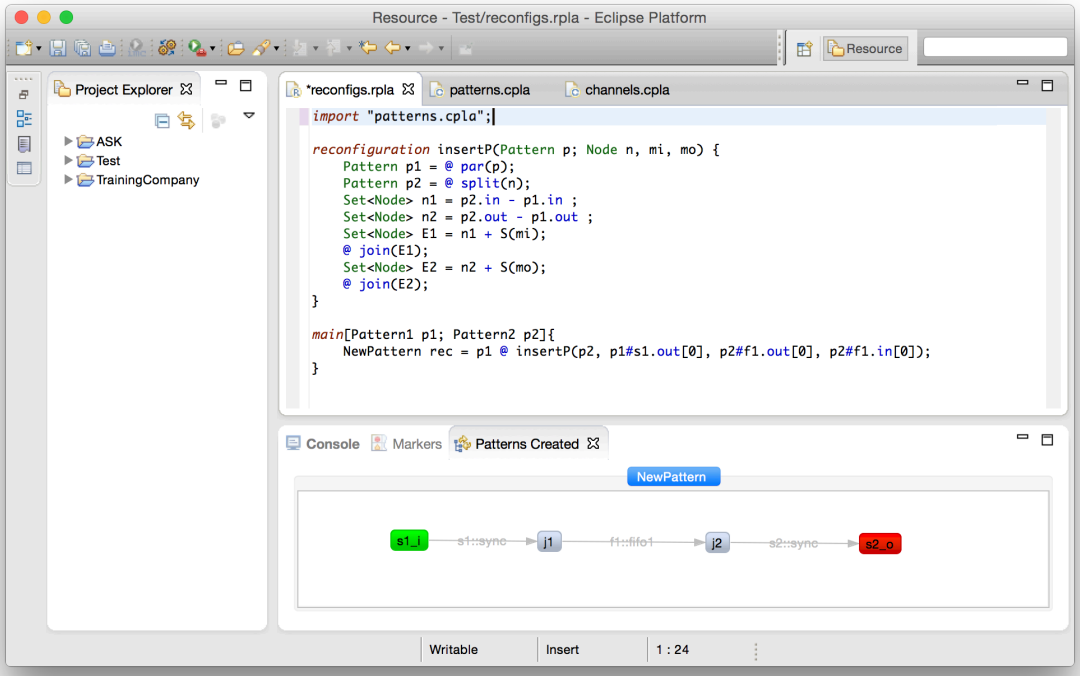


Figure 20: Patterns Created View as part of the editor.

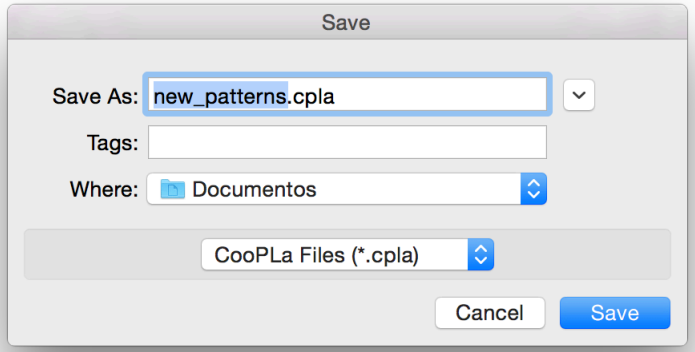


Figure 21: File dialog to save created patterns.

6

CASE STUDY

This chapter presents a case study that shows how the ReCooPLa engine can be used in the context of self-adaptive systems and integrated in an approach that deals with both the design and runtime monitoring of these demanding systems. Such an approach is proposed by Oliveira and Barbosa [OB14] and has roots on the reconfiguration framework presented in Section 3.5 and supporting tools. Essentially, it follows the feedback control loop MAPE(-K) model [KC03], where an active transition system of reconfigurations, referred to as the Reconfiguration Transition System (RTS), is integrated and plays a central role.

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

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

More details on this approach can be found in [OB14]. In the referred paper, an application case was developed to simulate the runtime adaptation of the Access Society's Knowledge (ASK) system. The case study presented in this chapter is, in fact, the design-time necessary part of that work, where the RTS is constructed with the help of the ReCooPLa engine.

6.1 THE ASK SYSTEM

ASK is a communication software developed by a Dutch company – Almende – to mediate consumers and services provides. It serves as a bridge between the interveners of the system according to their needs and profiles, in order to achieve the best consumer-provider match in the lower time possible, while keeping the entailing costs low.

The architecture of ASK comprises a web-based front-end (for user interaction), a database (for storing business data) and a contact engine (for matching the contacting interveners). The contact engine is the core of the architecture. It collects the requests, from the users of the system, and converts them into tasks. These are then processed to generate requests to a component called `Executor`, where they are placed in a queue – `Execution-Queue (EQ)` –, until a web service – `HandleRequestExecution (HRE)` – is ready to take and convert them into connections between service providers and consumers. The HRE service runs on a server separated from the EQ, but which is not dedicated; having the limit of spawning 20 HRE service instances in parallel.

The ASK system was previously studied regarding performance and resource allocation concerns from a static point of view [MAS+11, Moo11, OSB15]. These studies revealed bottlenecks and performance decay when environment changed with time, for example, due to the the increasing of the number of user requests at a given time of day or the downtime of a server. Such environment changes may contribute, for instance, to undesired financial losses for the company.

This leads to the proposal, design and implementation of a system whose architecture is able to adapt to the environment changes, referred to the Adaptable-ASK system.

6.2 ADAPTABLE-ASK DESIGN

The Alemende's solution was to evolve ASK into an adaptive system, which is able to change its architecture far adaptation to the environmental settings, in order to acquire the right amount of resources and continuing performing within desirable levels of QoS.

According to [OB14], by taking into account the system requirements and foreseen environment changes, it can be defined a suitable set of configurations and respective reconfigurations with the objective of creating a graph-like structure relating the two – the RTS.

In order to design the control loop, following the RTS-based approach briefly presented above, the ReCooPLa engine was used. The objective of this case study is exactly to show how the reconfiguration engine, along with ReCooPLa, can be used to design suitable configurations and reconfigurations and derive, from there, such an RTS to be used in the context of adaptable systems. Its focus is set only on the executor component and its architectural adaptation, bypassing the conversion of user requests into tasks, which, in fact, does not in-

sert any performance disturbance on the system. The initial work was to design (in CooPLa) the basic coordination layer for this component. Communication between the users and the HRE service is asynchronous, with requests being enqueued in a FIFO-like structure. This coordination pattern in Figure 22 (named `Original`) became then a building block that could be used in reconfigurations to produce more sustainable configurations of the ASK system.

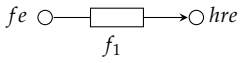


Figure 22: `Original` coordination pattern.

By creating a stochastic instance of mentioned coordination pattern and using the CooPLa processors, analysable assets were generated for quantitative analysis. Such an exercise was done, taking user request rates as the relevant environment change. When the number of requests increased, this architecture, with only one server hosting limited instances of the HRE service, has shown not to be enough. In this context, it may be necessary to add a second server to increase production. The coordination pattern in Figure 23 (named `ExRouter`) is a useful building block that can be used to achieve the addition of a new server to the architecture[1], while rearranging the coordination between such architectural elements.

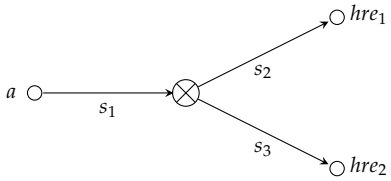


Figure 23: `ExRouter` coordination pattern.

At that moment, one could use (one stochastic instance of) this coordination pattern to design a reconfiguration that leads the architecture into the desired configuration of two HRE servers. Figure 24 shows how this was implemented with the ReCooPLa engine. In particular, it shows the ReCooPLa implementation of a reconfiguration to scale-out the Adaptable-ASK system from the original configuration.

The resulting coordination pattern was saved in a file named `scaledout.cpla`. It was then processed with CooPLa tools for being analysed with user request fluctuations retrieved from the logs of the ASK system. This has shown to be a suitable configuration for most of the request demands, but when the requests decrease, the company loses money from the rental of the second server. Thus it may be necessary to go back to the original configuration when such is the context. However, when the requests reach a peak, this solution was still not the best fit. Figure 25 shows a reconfiguration solution for these two problems. This solution is a ReCooPLa implementation of three reconfigurations where: the first is an auxiliary one to

1 Symbol ⊗ represents a node that routes data for one of its outgoing channels in a mutual-exclusive fashion.

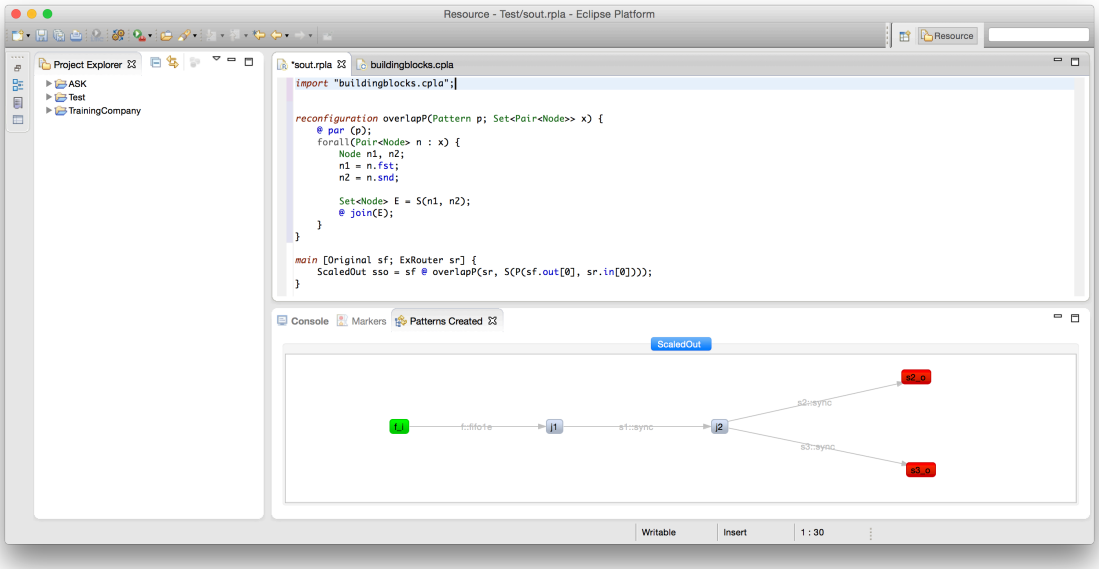


Figure 24: ReCooPLa implementation of the `scaled-out` reconfiguration.

preserve the scaled-out configurations; the second scales-in the Adaptable-ASK system from the scaled-out configuration; and the third takes the scaled-out configuration to the original one. The solution resulted in three exported coordination patterns, where only the second one was saved in a `scaledinout.cpla` file (the other patterns already exist in CooPLa files).

The reconfigurations must be well managed to increase the efficiency. However, the costs management is out of the scope of this case study. This work of designing reconfigurations ends up in the construction of the RTS as desired in the approach for self-adaptive systems described above. Figure 26 shows the RTS that is constructed from the partial study herein presented. The backwards reconfigurations are omitted for readability sake.

It includes also a state for a configuration where a log service is added to record information related to the user requests, using the reconfiguration presented in `log_sout.rpla`. From this reconfiguration, results another exported coordination pattern (`log_scaledout.cpla`). Figure 27 shows how this was implemented with the ReCooPLa engine.

Certainly, the complete RTS for the Adaptable-ASK has more states and transitions. For instance, a state representing a configuration where a certain amount of requests is acceptable to be lost, may make sense for some environmental conjuncture. The beauty of this approach is that the RTS may be changed at runtime with new or revised reconfigurations. If something goes wrong, the implemented reconfigurations should be revised and therefore, this is a cyclical process.

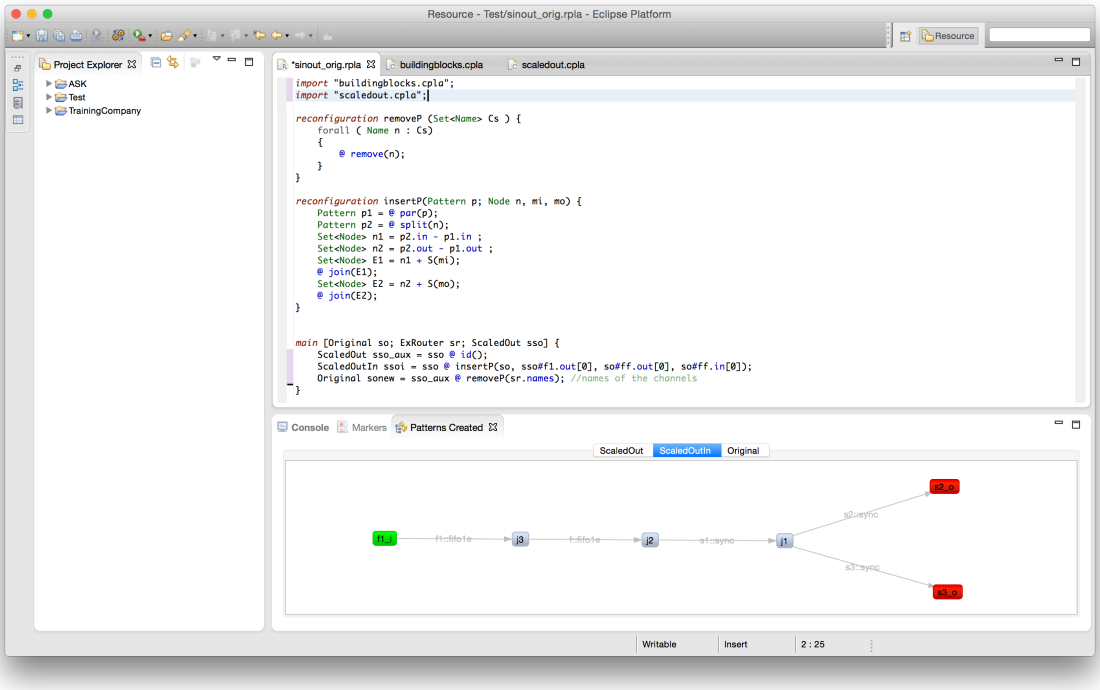


Figure 25: ReCooPLa implementation of three reconfigurations.

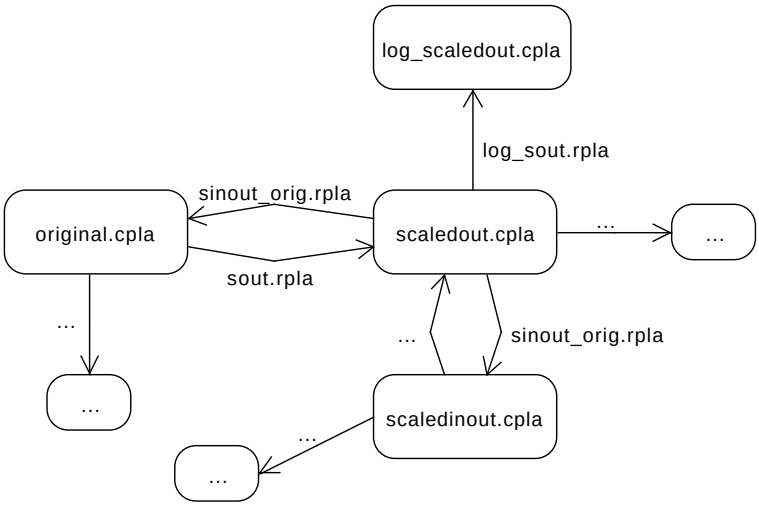


Figure 26: Partial RTS for the design of the Adaptable-ASK system.

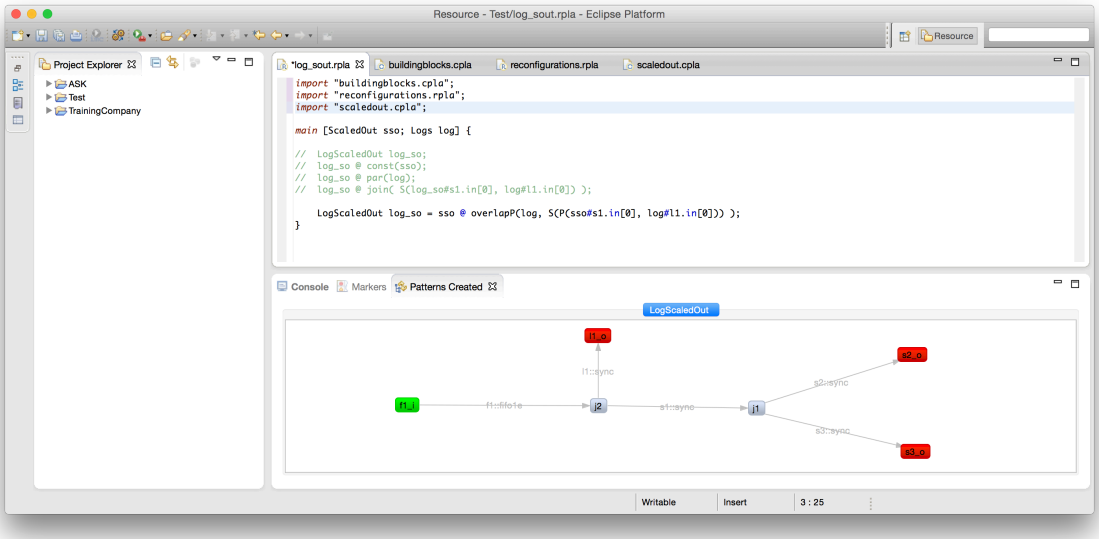


Figure 27: ReCooPLa implementation of a reconfiguration to add a log.

7

CONCLUSIONS AND FUTURE WORK

Throughout this dissertation it was shown that the adaptation of a system to new requirements or environments, is possible via reconfiguration of their architecture, notably, their coordination protocols.

To the time of writing of this dissertation, and apart from Krause's approach [Kra11] embedded in the ECT [AKM+08], there are no other approaches that directly target the specific reconfiguration of software coordination layer. With effect, the main objective of this master work, an engine for coordination-based reconfigurations, extends the state of the art, hopefully, as a suitable alternative to [Kra11].

To meet the objectives initially defined, a survey and study of adjacent concepts was made. To guide the research execution, a plan was also defined, with the main objective of developing a tool for rapid prototyping of coordination-based architectural reconfigurations. This involves the following tasks:

- Design of a formal language to express reconfigurations, through combination of primitive operations, based on the formal model presented in [OB13a].
- Development of a processor for the language that should report errors from syntactic and semantic analysis.
- Development of the engine for reconfigurations (based on the language) as a plugin for eclipse.
- Evaluation of the tool by means of a suitable case-study.

The first research topic (design of a formal language to express reconfigurations) was completed with success. To achieve this, a DSL named ReCooPLa was designed and also documented in this dissertation. ReCooPLa supports the model presented in [OB13a] and make it a suitable tool for the software architect. This language differs from other architectural languages by focussing on reconfigurations rather than on the definition of architectural elements like components, connectors and their interconnections. This work also led to a paper [ROB14a], invited to a special issue of a indexed journal with impact factor.

The second topic (development of a processor for the language) was completed with success. This processor is comprised of a few steps which ensure that the specifications are syntactically and semantically correct. The processor is the element that fills the gap between the ReCooPLa and its engine.

The third topic (development of the engine for reconfigurations as a plugin for eclipse) was also completed with success. As it often happens with DSLs, 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 a plugin of the CooPLa Editor[1] (an eclipse plugin itself). It is developed in Java to execute reconfigurations specified in ReCooPLa over graph-based structures – coordination patterns – which are defined in CooPLa [OB13b], a lightweight language to define the graph-like structure of coordination patterns. In addition, the reconfiguration primitives described in [OB13a] as elementary operations, were also implemented in Java, to support the reconfiguration engine. The ReCooPLa engine takes advantage of reflection features, associated to the Java programming language, to compile and dynamically load the generated Java files into the running ReCooPLa engine. It differs from other architectural reconfiguration tools due to its focus on the coordination layer rather than on the high-level architecture.

It can be argued that instead of generating Java files, compile and load them into the engine, one could have resorted to reflection packages such as Javassist[2] to dynamically create the classes, bypassing the external compilation stage. However these packages still fall short in some aspects of recent versions of the Java language. We stepped into that direction at a first attempt, but coping with Java generics in Javassist has shown to be a cumbersome, time consuming task.

In order to fulfil the last task mentioned above, a case study was described. The case study answers the research question initially presented (*How to simulate a reconfiguration in their design?*). In particular, it showed that despite ReCooPLa engine be targeted to the early stages of software development; *i.e.*, the design of reconfigurations and their analysis against requirements; it may play an important role in the design and maintenance of adaptive systems.

In addition to these objectives, another one existed that was not completely reached:

- Engine integration with the Reo coordination system (ReoTools).

This integration depended on the cooperation of third parties and, unfortunately, it has not been possible to perform it in due time. However, the CooPLa Editor is able to import and export the Reo visual language (to and from equivalent CooPLa specifications). This integrates, somehow, these tools, since the reconfigured patterns may be stored into CooPLa sentences.

1 coopla.di.uminho.pt
2 http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

This project also has a capacity of evolution insofar as some additional features can be developed. The physical integration of the engine with the ReoTools is one of such features.

For future work is also planned the import of architectural configurations specified in Wright, ACME, or similar ADLs, whenever their connectors are suitable instances of coordination patterns expressed in CooPLa. This would allow for taking advantage of ReCooPLa and its engine to define reconfigurations of the coordination layer, while having a full specification of a system architecture. This will make possible for the ReCooPLa engine to be part of a control loop for adaptive systems.

In the future, the ReCooPLa editor could also be extend with a feature for visualisation of the stages of reconfigurations, *i.e.*, with a visual debug. This would allow a better understanding of the reconfiguration process as well as a faster and accurate identification of a possible error in the specification.

BIBLIOGRAPHY

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, pages 121–135, Berlin, Heidelberg, 2010. Springer-Verlag.

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[AHP94] 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.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, 5(1):23–70, February 1993.

[AKM+08] Farhad Arbab, Christian Krause, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, testing and executing reo connectors with the eclipse coordination tools. In *proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2008)*, Salamanca, Spain, September 2008.

[All97] 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.

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Formal methods for distributed processing. chapter PICCOLA—a Small Composition Language, pages 403–426. Cambridge University Press, New York, NY, USA, 2001.

[Arb96a] Farhad Arbab. The iwim model for coordination of concurrent activities. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 34–56, London, UK, UK, 1996. Springer-Verlag.

[Arb96b] Farhad Arbab. Manifold version 2: Language reference manual. Technical report, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996.

[Arb98] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, pages 11–22, 1998.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BA96] Pascal Bouvry and Farhad Arbab. Visifold: A visual environment for a coordination language. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 403–406, London, UK, UK, 1996. Springer-Verlag.

[BBB+11] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3):41–48, 2011.

[BBG11] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCL+06] 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.

[BCS12] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. A model of context-dependent component connectors. *Science of Computer Programming*, 77(6):685–706, June 2012.

[BDGPZ12] Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4):1–71, November 2012.

[BFH87] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of graph transformations: A synchronization mechanism. *J. Comput. Syst. Sci.*, 34(2-3):377–408, June 1987.

[BHTV06] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and D´aniel Varr´o. Style-based modeling and refinement of service-oriented architectures. *Software & Systems Modeling*, 5(2):187–207, 2006.

[BJC05] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2Nd European Conference on Software Architecture*, EWSA'05, pages 1–17, Berlin, Heidelberg, 2005. Springer-Verlag.

[BJMS12] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using Dy-BIP. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2012.

[BLLMT08] 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.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, July 2006.

[CBG+04] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In *In Proc. IASTED Software Engineering and Applications (SEA'04*, 2004.

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring i: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, May 2007.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[CGZ95] Nicholas Carriero, David Gelernter, and Lenore D. Zuck. Bauhaus linda. In *Selected Papers from the ECOOP'94 Workshop on Models and Languages for*

Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems, ECOOP '94, pages 66–76, London, UK, UK, 1995. Springer-Verlag.

[CH05] Alan Colman and Jun Han. Coordination systems in role-based adaptive software. In Jean-Marie Jacquet and GianPietro Picco, editors, *Coordination Models and Languages*, volume 3454 of *Lecture Notes in Computer Science*, pages 63–78. Springer Berlin Heidelberg, 2005.

[Cia96] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys (CSUR)*, 28(2):300–302, 1996.

[CMR+96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe. Algebraic approaches to graph transformation, part i: Basic concepts and double pushout approach. Technical report, 1996.

[Cos10] D.F. de Oliveira Costa. *Formal models for component connectors*. PhD thesis, Vrije Universiteit, Amsterdam, 2010.

[DLLC09] 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.

[EOP06] Hartmut Ehrig, Fernando Orejas, and Ulrike Prange. Categorical foundations of distributed graph transformation. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2006.

[Erl09] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1st edition, 1999.

[FL13] José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software and System Modeling*, 12(2):349–367, 2013.

[Gad07] Fabio Gadducci. Graph rewriting for the π-calculus. *Mathematical. Structures in Comp. Sci.*, 17(3):407–437, June 2007.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–, February 1992.

[GH04] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 79–88, 2004.

[GJB96] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for online software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, February 1996.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[GMS07] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, pages 103–114, New York, NY, USA, 2007. ACM.

[GMW97] 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.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pages 23–34, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HP06] 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.

[HVM+05] Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

[JA12] Sung-Shik T. Q. Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comp. Sci.*, 22(1):201–251, 2012.

[Joh75] Stephen C. Johnson. YACC yet another compiler compiler. Computing Science Technical Report CSTR32, Bell Laboratories – Murray Hill, New Jersey, 1975.

[KAV09] Christian Koehler, Farhad Arbab, and Erik Vink. Reconfiguring distributed reo connectors. In Andrea Corradini and Ugo Montanari, editors, *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pages 221–235. Springer Berlin Heidelberg, 2009.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[KCPA08] Christian Koehler, David Costa, José Proença, and Farhad Arbab. Reconfiguration of reo connectors triggered by dataflow. *Electronic Communications of the EASST*, 10, 2008.

[Ken00] Elizabeth A. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, April 2000.

[KLA08] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. *Electron. Notes Theor. Comput. Sci.*, 194(4):77–92, April 2008.

[KMLA11a] 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.

[KMLA11b] 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.

[Knu90] Donald E. Knuth. The genesis of attribute grammars. In *Proceedings of the International Conference on Attribute Grammars and Their Applications*, WAGA, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[KO96] Bent Bruun Kristensen and Kasper Osterbye. Roles: Conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, December 1996.

[Kod04] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.

[KOM+10] 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.

[Kra11] Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, Amsterdam, The Netherlands, 2011.

[LM98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Softw. Eng.*, 24(7):521–533, July 1998.

[Lum99] Markus Lumpe. *A pi-calculus based approach to software composition*. PhD thesis, Universit¨at Bern, January 1999.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, September 1995.

[MAS+11] Y. Moon, F. Arbab, A. Silva, A. Stam, and C. Verhoef. Stochastic reo: a case study. In *Proceedings of the TTSS'11*, 2011. To appear.

[MB08] M. Malohlava and T. Bures. Language for reconfiguring runtime infrastructure of component-based systems. In *Proceedings of MEMICS 2008*, Znojmo, Czech Republic, November 2008.

[MDK92] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Configurable Distributed Systems, 1992., International Workshop on*, pages 102–117, Mar 1992.

[Med96] 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.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, April 1992.

[MHS05] 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.

[Mis05] Jayadev Misra. Computation orchestration. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 285–330. Springer Netherlands, 2005.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, October 1996.

[Mon01] Robert Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, January 2001. Version 2.3.

[Moo11] Young-Joo. Moon. *Stochastic Models for Quality of Service of Component Connectors*. PhD thesis, Universiteit Leiden, October 2011.

[MOT97] Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of off-the-shelf components in c2-style architectures. *SIGSOFT Softw. Eng. Notes*, 22(3):190–198, May 1997.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

[MT02] José Meseguer and Carolyn L. Talcott. Semantic models for distributed object reflection. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 1–36, London, UK, UK, 2002. Springer-Verlag.

[OB13a] 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.

[OB13b] 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.

[OB14] Nuno Oliveira and Luís S. Barbosa. A self-adaptation strategy for service-based architectures. Sep 2014. To appear.

[OMT98] 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.

[OPHdC09] 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.

[Ore96] Peyman Oreizy. *Issues in the runtime modification of software architectures*. Information and Computer Science, University of California, Irvine, 1996.

[OSB15] Nuno Oliveira, Alexandra Silva, and Luís S. Barbosa. $\mathsf{IMC}_{\mathsf{Reo}}$: interactive Markov chains for stochastic Reo. *Journal of Internet Services and Information Security*, 2015. To appear.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, Amsterdam, The Netherlands, The Netherlands, 1998.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[PBD09] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context awareness for dynamic service-oriented product lines. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 131–140, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[PC08] José Proença and Dave Clarke. Coordination models orc and reo compared. *Electron. Notes Theor. Comput. Sci.*, 194(4):57–76, April 2008.

[Per97] 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.

[PHL97] John Peterson, Paul Hudak, and Gary Shu Ling. Principled dynamic code improvement. Research Report YALEU/DCS/RR-1135, Yale University, Department of Computer Science, July 1997.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 368–377, New York, NY, USA, 1999. ACM.

[PSHA12] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Symbolic execution of reo circuits using constraint automata. *Sci. Comput. Program.*, 77(7-8):848–869, July 2012.

[RC90] Gruia-Catalin Roman and H. Conrad Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Trans. Softw. Eng.*, 16(12):1361–1373, December 1990.

[RC10] 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.

[RHR96] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending design environments to software architecture design. In *Proceedings of The 11th Knowledge-Based Software Engineering Conference*, KBSE '96, pages 63–, Washington, DC, USA, 1996. IEEE Computer Society.

[ROB14a] Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa. ReCooPLa: a DSL for Coordination-based Reconfiguration of Software Architectures. In Maria João Varanda Pereira, João Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 61–76, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[ROB14b] Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa. Towards an engine for coordination-based architectural reconfigurations. ComSIS, 2014.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[RW97] A. I. T. Rowstron and A. M. Wood. Bonita: A set of tuple space primitives for distributed coordination. In *Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture - Volume 1*, HICSS '97, pages 379–, Washington, DC, USA, 1997. IEEE Computer Society.

[RYC+06] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, roles and coordinators — a coordination model for open distributed and embedded systems. In *Proceedings of the 8th International Conference on Coordination Models and Languages*, COORDINATION'06, pages 247–265, Berlin, Heidelberg, 2006. Springer-Verlag.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing*, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[SMR+11] 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.

[SR11] Miao Song and Shangping Ren. Coordination operators and their composition under the actor-role-coordinator (arc) model. *SIGBED Rev.*, 8(1):14–21, March 2011.

[Tae99] Gabriele Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, 7(4):431–462, 1999.

[Tae04] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In JohnL. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin Heidelberg, 2004.

[Tal06] Carolyn L. Talcott. Coordination models based on a formal model of distributed object reflection. *Electron. Notes Theor. Comput. Sci.*, 150(1):143–157, March 2006.

[TB94] Gabriele Taentzer and Martin Beyer. Amalgamated graph transformations and their use for specifying agg — an algebraic graph grammar system. In Hans-Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 380–394. Springer Berlin Heidelberg, 1994.

[TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[Tol96] Robert Tolksdorf. Coordinating services in open distributed systems with laura. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 386–402, London, UK, UK, 1996. Springer-Verlag.

[TSR11] Carolyn Talcott, Marjan Sirjani, and Shangping Ren. Comparing three coordination models: Reo, arc, and pbrd. *Sci. Comput. Program.*, 76(1):3–22, January 2011.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[WF99] Michel Wermelinger and José Luiz Fiadeiro. Algebraic software architecture reconfiguration. In *Software Engineering—ESEC/FSE'99*, page 393–409. Springer, 1999.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Syst. J.*, 37(3):454–474, July 1998.

[Wol97] Alexander L. Wolf. Succeedings of the second international software architecture workshop (isaw-2). *SIGSOFT Softw. Eng. Notes*, 22(1):42–56, January 1997.