
Atlantis Studies in Computing
*Series Editors:* J. A. Bergstra · M. W. Mislove

Jacopo Mauro

# Constraints Meet Concurrency

# Atlantis Studies in Computing

Volume 5



For further volumes:
http://www.atlantis-press.com

## Aims and Scope of the Series

The series aims at publishing books in the areas of computer science, computer and network technology, IT management, information technology and informatics from the technological, managerial, theoretical/fundamental, social or historical perspective.

We welcome books in the following categories:

Technical monographs: these will be reviewed as to timeliness, usefulness, relevance, completeness and clarity of presentation.
Textbooks.

Books of a more speculative nature: these will be reviewed as to relevance and clarity of presentation.

For more information on this series and our other book series, please visit our website at:

www.atlantis-press.com/publications/books
Atlantis Press
29, avenue Laumière
75019 Paris, France

Jacopo Mauro

# Constraints Meet Concurrency

Jacopo Mauro
University of Bologna
Bologna
Italy

ISSN 2212-8557 ISSN 2212-8565 (electronic)
ISBN 978-94-6239-066-9 ISBN 978-94-6239-067-6 (eBook)
DOI 10.2991/978-94-6239-067-6

Library of Congress Control Number: 2014930346

*I dedicate this to my family and friends;*
*Nullius boni sine socio iucunda possessio est*

# Foreword

The Italian Chapter of the European Association for Theoretical Computer Science (EATCS) was founded in 1988, and aims at facilitating the exchange of ideas and results among Italian theoretical computer scientists, and at stimulating cooperation between the theoretical and the applied communities in Italy.

One of the major activities of this Chapter is to promote research in theoretical computer science, stimulating scientific excellence by supporting and encouraging the very best and creative young Italian theoretical computer scientists. This is done also by sponsoring a prize for the best Ph.D. thesis. An interdisciplinary committee selects the best two Ph.D. theses, among those defended in the previous year, one on the themes of Algorithms, Automata, Complexity, and Game Theory, and the other one on the themes of Logics, Semantics, and Programming Theory.

In 2012, we started a cooperation with Atlantis Press so that the selected Ph.D. theses will be published as volumes in the Atlantis Studies in Computing.

The present volume contains one of the two theses selected for publication in 2013:

*Constraints Meet Concurrency* by Jacopo Mauro (supervisor: Prof. Maurizio Gabbrielli, University of Bologna, Italy)

and

*Secure Computation Under Network and Physical Attacks* by Alessandra Scafuro (supervisor: Prof. Ivan Visconti, University of Salerno, Italy).

The scientific committee which selected these theses was composed of Profs. Andrea Clementi (University of Rome Tor Vergata), Mimmo Parente (University of Salerno), and Elena Zucca (University of Genova).

They gave the following reasons to justify the assignment of the award to the thesis by Jacopo Mauro:

*The Ph.D. thesis of Jacopo Mauro provides impressive and important results on the use of constraints in concurrency. He investigates the expressive power of the Constraint Handling Rules (CHR) language, finding syntactic characterizations which make the language non Turing-equivalent, and showing that the addition of priority mechanisms augments the expressive power. These are very strong theoretical results on language expressivity, technically difficult and very interesting.*

*Moreover, he considers more applicative issues about the design and construction of architectures for software services. He develops a system for efficiently*

*solving constraint problems by several solvers, integrated in a cloud-based architecture implemented in Jolie, a language for Service Oriented Computing. To enhance the efficiency of the system he also uses learning techniques and failure-handling capabilities.*

*In summary, the thesis is a laudable combination of theoretical and practical contributions, also supported by a very good record of publications, and it constitutes a very valuable addition to the research on CHR and on service-oriented computing.*

I would like to thank the members of the scientific committee, and I hope that this initiative will further contribute to strengthen the sense of belonging to the same community of all the young researchers who have accepted the challenges posed by any branch of theoretical computer science.

Tiziana Calamoneri

# Preface

It is a great pleasure for me for to write this Preface for the thesis of Jacopo Mauro, for at least three reasons.

The first one is the occasion for this Preface. Publishing a Ph.D. thesis in the Atlantis Studies in Computing, after having received a prize from the Italian Chapter of EATCS, is certainly a relevant achievement. Our Ph.D. school in Bologna is proud of such a result, which encourages us to continue on this path toward excellence.

The second reason is related to the content and the results of Jacopo's thesis. This is not the canonical Ph.D. thesis which deals with a single problem in a specific field. In fact the thesis considered two quite different research topics, constraint programming and concurrency theory, trying to identify the synergies and the cross fertilization deriving from their interaction and addressing both theoretical problems and practical applications. This was a quite challenging objective, which required a relevant effort to master many different technical notions. Nevertheless, the results obtained by Jacopo were excellent, as witnessed by the external reviewers of the thesis who wrote:

> This excellent thesis involves several important scientific contributions to the fields of declarative programming and concurrent programming. . . . In conclusion, I consider this thesis an important and impressive work.

Finally, I am happy to write this Preface also for a personal reason. Having been the advisor of Jacopo has been a pleasure for me, because I have seen (and still see) a continuous improvement in his technical skills and in his ability of doing research. I am very well aware that the advisor has no merit in such a process, having him mainly in a maieutic role. Nevertheless, it is pleasant to think that even such an indirect role has somehow contributed to the growth of a person.

Bologna, December 2013 Maurizio Gabbrielli

# Acknowledgments

Writing acknowledgments is one of the most difficult tasks. No matter how hard you try, you will always end up forgetting somebody whose support has been very important. For this reason my first thanks go to all the professors, assistants, colleagues, and friends with whom I have had the pleasure to interact with in these years.

I am truly indebted and thankful to my mentor/advisor/supervisor, Prof. Maurizio Gabbrielli, who in these last 3 years has been a source of wisdom and support, sometimes refraining my enthusiasms, and at other times inciting me to follow different directions. Thank you for your patience, your constant presence, and guidance.

Needless to say, I am also indebted to my coauthors Mila Dalla Preda, Claudio Guidi, Ivan Lanese, Zeynep Kiziltan, Luca Mandrioli, Maria Chiara Meo, Fabrizio Montesi, Barry O'Sullivan, Jon Sneyers, and Gianluigi Zavattaro.

I also want to thank all the friends with whom I have spent all these years in Bologna. Among them I must in particular thank all the office colleagues with whom I have spent endless hours of discussion and the people who were harassed by the lunch calls. Ordered by the distance of their working position from my desk, a non-comprehensive list of these stoic people is: Paolo Parisen Toldin, Marco di Felice, Ornela Dardha, Sara Zuppiroli, Michael Lienhardt, Tudor Alexandru Lascu, Silvio Peroni, Giulio Pellitta, Gustavo Marfia, Wilmer Ricciotti, Elena Giachino, Dominic Mulligan, Ugo Dal Lago, Claudio Sacerdoti Coen, Matteo Magnani, Jaap Boender, Ludovico Antonio Muratori, Francesco Turroni, Ferdinanda Camporesi, Cinzia Di Giusto, Jorge Perez, Claudio Mezzina, Gaboardi Marco, and Micaela Spigarolo.

During my visit to Amsterdam I met a lot of precious friends. I would like to thank them all, especially all the people of the SEN3 group. I am really grateful for the wonderful time I spent there with you.

Last, but absolutely not least, deep thanks go to my parents and to my sisters. Even though they probably understand very little of it, this thesis is also the result of their work.

# Contents

# Chapter 1
# Introduction

In this thesis we explore the interactions between constraints and concurrency, two well-known areas in computer science.

Constraint is a ubiquitous concept: in everyday life there are a lot of rules (physical, chemical, economical, and legal) that restrict, limit, or regulate the way we operate and what decisions we take. In computer science, constraints can be very useful not only to model the world but also to discover or verify if instances satisfy a model. For these reasons the notion of constraints gave birth to a new field called Constraint Programming that has attracted wide attention since it provides a concise and elegant way to describe problems and also efficient tools to compute solutions.

Concurrency is also a universal concept. In every second of our life, there are thousands of events or tasks occurring simultaneously and interacting with each other. With the evolution of the networks a lot of connected computers are available and nowadays more and more people think that we are inevitably going towards a world full of interconnected devices. This network of devices is a concurrent system and has peculiarities and characteristics that an environment constituted by only one processing unit does not have. On one hand, a concurrent system is usually hard to use since, in such a system, problems like deadlocks, resources conflicts, and security emerge. On the other hand, a concurrent system can be the only means to solve problems requiring huge amounts of resources or modeling complex scenarios in a simple and clear way.

Starting from the concepts of constraints and concurrency, natural questions arise: What happens when these two ideas meet? Is the constraint notion useful in the field of the concurrency theory? Can Constraint Programming exploit concurrent system to solve or model new and more complex problems?

The goal of this thesis is to provide a positive answer to these questions investigating the possible connections between the constraint and concurrency fields. To do so, in the first part of the thesis, we concentrate on some of the benefits that the notion of constraint can bring to the field of concurrency theory. Specifically, we focus on Constraint Handling Rules (CHR), i.e., a concurrent language that supports constraints as first-class primitives, and we study the expressive power of constraints and priorities

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_1, © Atlantis Press and the authors 2014

in such a language. We prove the non-Turing completeness of some CHR fragments, and a comparison between different variants of CHR languages using technical tools like well-structured transition systems and language encodings.

In the second part of the thesis, we investigate how concurrent systems can be used to solve problems modeled by using constraints in a more efficient way. We define a framework called "Constraints In Clouds" (CiC) that, deployed on a concurrent system, solves constraint problems in a faster, cheaper, and more efficient way. The CiC framework exploits the fact that different constraint solvers are better at solving different problem instances, even within the same problem class. It uses machine learning techniques to predict the best solvers to use for every problem instance and, following these predictions, decides which solver needs to be used. Its goal is to employ strategies that minimize the time needed to solve, a set of problem instances, the consumption of computing resources, and the risk of failures.

To develop the CiC framework we adopted a service-oriented approach using the concurrent language Jolie. We chose to follow the service- oriented paradigm because nowadays, it is the most used one for programming large, complex, distributed, or cloud-based applications, thanks to its modularity, flexibility, extensibility, and reliability. By using Jolie, however, we observed some of its limitations like the lack of the broadcast primitives or the impossibility/inefficiency of some transaction compensations. To overcome these limitations, we provide an extension of the Jolie language that not only contributes to the improvement of service- oriented languages, but also allows us a better and more efficient implementation of the CiC framework.

## 1.1 Thesis Outline and Contributions

In this section we briefly describe the overview of the contents of the thesis. Essentially, the thesis is divided into three main parts. The first part (Chaps. 2, 3) gives an overview of the constraint and concurrency fields, the second (Chaps. 4–6) presents the studies of the expressive power of the concurrent language CHR. The third part (Chaps. 7–10) describes instead the CiC framework with the above-mentioned Jolie extension. In more detail, in

Chapter 2 we give an overview of the Constraint Programming field, starting with a historical background, and then focusing on the key Constraint Programming concepts: propagation and search;

Chapter 3 we give an overview of some topics faced by the concurrency community. In particular, we will define what is a concurrent system and what are the main approaches used to model and describe it from a formal but also practical point of view;

Chapter 4 we recall the Constraint Handling Rules language with three of its semantics;

Chapter 5 we present our first contribution. In this chapter we prove that two fragments of CHR are not Turing powerful using the theory of well-structured transition

systems, and a direct approach since classical techniques like encoding into Petri nets or finite state automata could not be used;

Chapter 6 we present a novel comparison between Turing powerful CHR languages. By using language encodings, we show that priorities in CHR augment the expressive power of the language while instead the use of dynamic priorities, i.e., priorities that can vary at runtime, do not increase the expressive power obtained using static priorities only;

Chapter 7 we describe the "Constraints In Clouds" (CiC) framework, its main features, the development of a first prototype using Jolie and some preliminary tests;

Chapter 8 we investigate the use of machine learning techniques, specifically, classification techniques, to enhance the performance of the CiC framework. We study the approaches for classifying the runtime of constraint solvers, and we simulate strategies that could be adopted within the CiC framework to empirically evaluate their performances;

Chapter 9 we add the support of broadcast message to Jolie. In particular, we describe a novel approach based on radix trees that handles broadcast messages with optimal complexity. Broadcast messages can therefore be used to obtain a more concise, and efficient implementation of the CiC framework;

Chapter 10 we extend the Jolie language considering, a new mechanism for handling, and compensating ongoing transactions, that for external or internal reasons have failed. This mechanism allows, a more reliable implementation of the CiC framework, and as a secondary effect, a clearer, and efficient handling of timeouts;

Chapter 11 contains concluding remarks, and future directions

Apart from Chaps. 5 and 6 that assume the description of the CHR language provided in Chap. 4, all the remaining chapters are self-contained.

All the original contributions of the thesis have been published. In particular the work presented in Chap. 5 has been published in [49] while Chaps. 6–10 have been published in [48, 72, 73, 85] and [89] respectively. Moreover, the works presented in Chaps. 9 and 10 were carried out within the Focus Research Project: a joint effort between the Institut national de recherche en informatique et automatique (INRIA) and the University of Bologna.

# Part I
# Constraint and Concurrency Overview

# Chapter 2
# Constraints

A constraint is something that restricts, limits, or regulates. From this notion of constraint, a new field of computer science has been created: Constraint Programming (CP).

CP has attracted high attention among experts from many areas because of its potential for solving hard real-life problems and because it is based on a strong theoretical foundation. The success of CP derives from the fact that on one hand it allows to model a problem in a simple way and on the other hand it provides efficient problem solving algorithms.

There are a lot of surveys on CP. One of the best and complete works on CP is [108]. To be short, however, we will follow [10]. After giving in Sect. 2.1, a brief history of CP we will first define what a Constraint Satisfaction Problem (CSP) is in Sect. 2.2 and how these problems are solved in Sects. 2.3, 2.4, 2.5, and 2.6. In Sects. 2.8 and 2.7 we will describe some extensions of CP while in Sects. 2.9 and 2.10 we will present some of its applications and limitations.

## 2.1 Short History

The earliest ideas leading to CP may be found in the Artificial Intelligence (AI) dating back to the 1960s and 1970s. The scene labeling problem [129] where the goal is to recognize the objects in a 3D scene by interpreting lines in the 2D drawings is probably the first CSP that was formalized. The main algorithms developed in those years were related to achieving some form of consistency. Another application for constraints is interactive graphics where Ivan Sutherland's Sketchpad [120], developed in the 1960s, was the pioneering system. Sketchpad and its follower were interactive graphics applications that allowed the user to draw and manipulate constrained geometric figures on the computer's display. These systems contribute to developing local propagation methods and constraint compiling. The main step toward CP was achieved when Gallaire [51] and Jaffar and Lassez [66] noted that

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_2, © Atlantis Press and the authors 2014

logic programming was just a particular kind of CP. The basic idea behind Logic Programming is that the user states what has to be solved instead of how to solve it, which is very close to the idea of constraints. Therefore, the combination of constraints and logic programming is very natural and Constraint Logic Programming (CLP) makes a nice declarative environment for solving problems by means of constraints. However, it does not mean that CP is restricted to CLP. Constraints were integrated to typical imperative languages like c++ [111] or Java [77] as well.

CP has an inner interdisciplinary nature. It combines and exploits ideas from a number of fields including AI, Combinatorial Algorithms, Computational Logic, Discrete Mathematics, Neural Networks, Operations Research, Programming Languages, and Symbolic Computation.

## 2.2 Constraint Satisfaction Problems

CSPs have been a subject of research in AI for many years. A CSP is defined as:

- a set of variables $X = x_1, \ldots, x_n$
- for each variable $x_i$ a finite set $D_i$ of possible values like naturals, reals or strings. $D_i$ is called domain
- a set of constraints restricting the values that the variables can simultaneously take.

**Example 2.1** *Let us show how the famous "send more money" mathematical game published in the July 1924 issue of Strand Magazine by Henry Dudeney can be modeled. This game is an alphametics, i.e., a type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters. To solve the game, we need to associate to every letter in "send more money" a different number from 0 to 9 in a way that the sum of "send" and "more" is equal to "money." We also require that the leading digit of a multidigit number is not zero.*

*Since we need to find what numbers are associated to every letter we can model these numbers with variables which domain is the set* $\{0, \ldots, 9\}$. *Let be* $c_v$ *the variable associated to the letter* $c$.

*The set of constraint to consider are:*

- $s_v e_v n_v d_v + m_v o_v r_v e_v = m_v o_v n_v e_v y_v$
- $c_v \neq c'_v$ *if* $c \neq c'$
- $s_v \neq 0$
- $m_v \neq 0$.

A solution to a CSP is a labeling, i.e., an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied at once. We may want to find:

- just one solution, with no preference
- all solutions

- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

Solutions to a CSP can be found by searching (systematically) through the possible assignments of values to variables. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments), and those that explore the space of complete value assignments (to all variables) stochastically.

## 2.3 CSP Solving: Systematic Search

From the theoretical point of view, solving CSP is trivial using systematic exploration of the solution space. Even if systematic search methods without additional improvements look very simple and nonefficient, they are important and worth mentioning because they make the foundation of more advanced and efficient algorithms.

The basic constraint satisfaction algorithm that searches the space of complete labelings, is called generate-and-test. The idea is simple: a complete labeling of variables is generated and, consequently, if this labeling satisfies all the constraints then the solution is found, otherwise, another labeling is generated. The generate-and -test algorithm is a weak generic algorithm that is used if everything else failed. Its efficiency is poor because of noninformed generator and late discovery of inconsistencies. Consequently, there are two ways to improve its efficiency:

- the generator of valuations is smart, i.e., it generates the complete valuation in such a way that the conflict found by the test phase is minimized
- the generator is merged with the tester, i.e., the validity of the constraint is tested as soon as its respective variables are instantiated. This method is used by the backtracking approach. Backtracking [87] is a method of solving CSP by incrementally extending a partial solution that specifies consistent values for some of the variables, toward a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test. However, its running complexity for most nontrivial problems is still NP-hard.

There are three major drawbacks of the standard backtracking:

1. thrashing, i.e., repeated failure due to the same reason
2. redundant work, i.e., conflicting values of variables are not remembered
3. late detection of the conflict, i.e., conflict is not detected before it really occurs.

We will now present some of the improvements of backtracking studied in the literature.

## 2.4 CSP Solving: Consistency Techniques

One alternative approach for solving CSP is based on removing inconsistent values from variables' domains till the solution has been founded. These methods are called consistency techniques. There are several consistency techniques [76, 83] but most of them are not complete, i.e., they can not be used alone to solve a CSP completely. The names of basic consistency techniques are derived from the graph notions. The CSP is usually represented as a constraint graph or hypergraph (sometimes called constraint network) where nodes correspond to variables and edges/hyperedges are labeled by constraints.

The simplest consistency technique is referred to as a node consistency. It removes values from variable domains that are inconsistent with unary constraints on respective variables. The most widely used consistency technique is called arc consistency. This technique removes values from variables domains that are inconsistent with binary constraints. There exist several arc consistency algorithms starting from AC-1 based on repeated revisions of arcs till a consistent state is reached or some domain become empty. The most popular among them are AC-3 and AC-4.

Even more inconsistent values can be removed by path consistency techniques. Path consistency is a property similar to arc consistency, but considers pairs of variables instead of only one. A pair of variables is path-consistent with a third variable if each consistent evaluation of the pair can be extended to the other variable in such a way that all binary constraints are satisfied. There exist several path consistency algorithms like PC-1 and PC-2 but, compared to algorithms for arc consistency, they need an extensive representation of constraints that is memory consuming.

All above-mentioned consistency techniques are covered by a general notion of $k$-consistency [39] and strong $k$-consistency. A constraint graph is $k$-consistent if for every system of values for $k-1$ variables satisfying all the constraints among these variables, there exist a value for arbitrary $k$th variable such that the constraints among all $k$ variables are satisfied. A constraint graph is strongly $k$-consistent if it is j-consistent for all $j \leq k$. We have that:

- node consistency is equivalent to strong 1-consistency
- arc consistency is equivalent to strong 2-consistency
- path consistency is equivalent to strong 3-consistency.

Algorithms exist for making a constraint graph strongly $k$-consistent for $k > 2$ but in practice they are rarely used because of efficiency issues.

Although these algorithms remove more inconsistent values than any arc-consistency algorithm they do not eliminate the need for search in general. Restricted forms of these algorithms removing a similar amount of inconsistencies with a greater efficiency have been proposed. For example, directional arc consistency revises each arc only once, requires less computation than AC-3 and less space than AC-4 but is still able to achieve full arc consistency in some problems. It is also possible to weaken the path consistency in a similar way.

## 2.5 Constraint Propagation

Both systematic search and consistency techniques can be used alone to completely solve the CSP but this is rarely done in practice. A combination of both approaches is more commonly used. To avoid some problems of backtracking like thrashing or redundant work, look back schemes were proposed. Backjumping [52], for instance, is a method to avoid thrashing. The control of backjumping is exactly the same as backtracking except when backtracking takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed so far. However, if backjumping finds an inconsistency, it analyses the situation in order to identify the source of inconsistency. It uses the violated constraints as guidance to find out the conflicting variable. If all the values in the domain are explored then the backjumping algorithm backtracks to the most recent conflicting variable. This is a main difference from the backtracking algorithm that backtracks to the immediate past variable.

Another look back schema called backchecking [63] avoids redundant work. Backchecking and its evolution backmarking are useful algorithms for reducing the number of compatibility checks. If the algorithm finds that some label $Y/b$ is incompatible with any recent label $X/a$ then it remembers this incompatibility. As long as $X/a$ is still committed, the $Y/b$ will not be considered again. Backmarking is an improvement over backchecking since it reduces the number of compatibility checks by remembering for every label the incompatible recent labels and avoids repeating compatibility checks which have already been performed.

All look back schemes share the disadvantage of late detection of the conflict. Indeed, they solve the inconsistency when it occurs but they do not prevent the inconsistency to occur. For this reason, look ahead schemes were proposed. For instance forward checking, the simplest example of look ahead strategy, performs arc-consistency between pairs of a non instantiated variable and an instantiated one removing temporarily the values that the noninstantiated variable can not assume. It maintains the invariance that for every unlabeled variable there exists at least one value in its domain that is compatible with the values of instantiated/labeled variables. Even though forward checking does more work than backtracking when each assignment is added to the current partial solution, it is almost always a better choice than chronological backtracking.

Further, future inconsistencies are removed by the partial look ahead method. While forward checking performs only the checks of constraints between the current variable and the not defined variables, the partial look ahead extends this consistency checking even to variables that have not direct connection with labeled variables, using directional arc consistency. The approach that uses full arc-consistency after each labeling step is called (full) look ahead.

## 2.6 Stochastic and Heuristic Algorithms

In the last few years, greedy local search strategies have become popular. These algorithms alter incrementally inconsistent value assignments to all the variables. They use a "repair" or "hill-climbing" metaphor to move toward more and more complete solutions. To avoid getting stuck at "local minimum," they are equipped with various heuristics to randomize the search but this stochastic nature generally avoids the guarantee of "completeness" provided by the systematic search methods.

Hill-climbing is probably the most famous algorithm of local search [87]. It starts from a randomly generated labeling of variables and, at each step, it changes a value of some variable in such a way that the resulting labeling satisfies more constraints. If a strict local minimum is reached then the algorithm restarts using other randomly generated states. The algorithm stops as soon as a global minimum has been found, i.e., all constraints are satisfied, or some resource (e.g., time, memory) is exhausted.

To avoid exploring the whole neighborhood of a state, the min-conflicts heuristic was proposed [92]. This heuristic chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints. Because the pure min-conflicts algorithm cannot go beyond a local-minimum, some noise strategies were introduced. Among them worth presenting is the random walk heuristics [112] that for a given conflicting variable picks randomly a value with probability $p$ and applies the min-conflicts heuristic with probability $1 - p$. The hill-climbing algorithm using the random-walk heuristic is also known as Steepest-Descent-Random-Walk.

Tabu search is another method to avoid cycling and getting trapped in local minimum [54, 55]. It is based on the notion of tabu list, a special short-term memory that maintains a selective history, composed of previously encountered configurations or, more generally, pertinent attributes of such configurations. A simple tabu search strategy consists in preventing configurations of tabu list from being recognized for the next k iterations. Such a strategy prevents algorithm from being trapped in short-term cycling and allows the search process to go beyond local optima.

## 2.7 Constraint Optimization

In many real-life applications, we do not want to find any solution but a good solution. The quality of solutions is usually measured by an application dependent function called objective function. The goal is to find a solution that satisfies all the constraints and minimize or maximize the objective function. Such problems are referred to as Constraint Optimization Problems (COP). A COP consists of a standard CSP and an optimization function that maps every solution to a numerical value.

The most used algorithm for finding optimal solutions is called branch and bound [81]. It needs a heuristic function mapping a partial labeling to a numerical value that represents an under estimate (in case of minimization) of the objective function for the best complete labeling obtained from the partial labeling. The branch and bound algorithm searches for solutions in a depth first manner and behaves like chronological backtracking except that, as soon as a value is assigned to the variable, the heuristic function is applied to the partial labeling. If case the under estimate obtained exceeds the bound of the best solution encountered so far the subtree under the partial labeling is pruned.

The efficiency of branch and bound is determined by two factors: the quality of the heuristic function and whether a good bound is found early. Observations of real-life problems show improving a good solution are usually the most computationally expensive part of the solving process.

Note that the branch and bound algorithm can be used to find suboptimal solutions too. For instance, the algorithm can compute all the solution and return a solution that reaches an acceptable bound even though this solution is not proved to be optimal.

## 2.8 Constraint Programming Extensions

Problems in which it is not possible to satisfy all the constraints are called over-constrained. Several approaches were proposed to handle these problems. Here we present two of these approaches, namely Partial Constraint Satisfaction and Constraint Hierarchies.

Partial Constraint Satisfaction [40] involves finding a solution to a CSP problem where some constraints are "weaken" to permit additional acceptable value combinations. Formally, a Partial Constraint Satisfaction problem is defined as a standard CSP with some evaluation function that maps every labeling of variables to a numerical value. The goal is to find a labeling with the best value of the evaluation function. A Partial Constraint Satisfaction problem looks like a COP with the difference that the satisfaction of all the constraint is not required. In fact, the global satisfaction is described by the evaluation function and constraints are used as a guide to find an optimal value of the evaluation function. Many standard algorithms like backjumping, backmarking, arc-consistency, forward checking, and branch and bound were customized to work with Partial Constraint Satisfaction problems.

Constraint hierarchies [24] is another approach of handling over-constrained problems. The constraint is weakened explicitly here by specifying its strength or preference. It allows one to specify not only the constraints that are required to hold, but also weaker constraints (usually called soft constraints) . Intuitively, the hierarchy does not permit to the weakest constraints to influence the result at the expense of dissatisfaction of a stronger constraint. Currently, two groups of constraint hierarchy solvers can be identified, namely refining method and local propagation. While the refining methods solve the constraints starting from the strongest level and continuing to

weaker levels, the local propagation algorithms gradually solve constraint hierarchies by repeatedly selecting uniquely satisfiable constraints.

## 2.9 Applications

CP has been successfully applied to many areas as diverse as DNA structure analysis, time-tabling for hospitals or industry scheduling. It proved to be well adapted for solving real-life problems because many application domains evoke constraint descriptions naturally.

Assignment problems were perhaps the first type of industrial application that were solved with the constraint tools. A typical example is the stand allocation for airports, where aircraft must be parked on the available stand during the stay at airport or counter allocation for departure halls. Another example is berth allocation to ships in the harbor or refinery berth allocation.

Another typical constraint application area is personnel assignment where work rules and regulations impose difficult constraints. The important aspect in these problems is the requirement to balance work among different persons. Systems like Gymnaste were developed for production of rosters for nurses in hospitals, for crew assignment to flights or stuff assignment in railways companies.

Successful applications for finite domain constraint are the once that solve scheduling problems, where, again, constraints express naturally the real-life limitations. Constraint-based software is used for well-activity scheduling, forest treatment scheduling, production scheduling in plastic industry or for planning production of military and business jets. The usage of constraints in Advanced Planning and Scheduling systems is increasing due to current trends of on-demand manufacturing.

Another large area of constraint application is network management and configuration. These problems include planning of cabling of the telecommunication networks in the building or electric power network reconfiguration for maintenance scheduling without disrupting customer services. Another example is optimal placement of base stations in wireless indoor telecommunication networks [47]. There are many other areas that have been tackled using constraints. Recent applications of CP were used in computer graphics, natural language processing, database systems, molecular biology, business applications, electrical engineering, and transport problems.

## 2.10 Limitations

Since many problems solved by CP are NP-hard problems, the identification of restrictions that make the problem tractable is very important for both the theoretical and the practical points of view. Unfortunately, the efficiency of constraint programs

is still unpredictable and the intuition is usually the most important part of deciding when and how to use constraints. A common problem for CP users is the stability of the constraint model. Even small changes in a program or in the data can lead to a dramatic change in performance. The process of performance debugging for a stable execution over a variety of input data is currently not well understood.

Another problem is choosing the right constraint satisfaction technique for a particular problem. Sometimes fast blind search like chronological backtracking is more efficient than more expensive constraint propagation and vice versa.

A particular problem in many constraint models is the cost optimization. Sometimes, it is very difficult to improve an initial solution, and a small improvement takes much more time than finding the initial solution.

Finally, constraint programs can add constraints dynamically but they do not support the on-line constraint solving required, for instance, in a changing environment. For instance, the possibility of deleting a constraint at runtime has been considered by some extensions like the ones described in [124] but this kind of operation is yet too costly to be performed.

# Chapter 3
# Concurrency

Concurrency in computer science is a property of systems in which more than one execution context can be active at the same time. We can think of a concurrent system as a system where computations are logically or physically executed at the same time and are potentially interacting with each other.

Concurrency can be divided into two broad classes: physical or logical. We talk about physical concurrency when there is a real simultaneous execution (two or more computation units are required), logical when instead the concurrent execution is physically executed sequentially but the users perceive the executions as concurrent.

In Sects. 3.1 and 3.2, we overview different types of concurrent systems and what problems arise when we are dealing with them. In Sect. 3.3, we will introduce two of the most famous mathematical models that have been developed for modeling concurrent computation while in Sect. 3.4 we will define what languages are currently used for developing concurrent applications. Finally in Sect. 3.5, we will introduce Service-Oriented Computing, a new promising approach for concurrency emerged in the last few years following the expansion of the Web.

## 3.1 Concurrent System

We can distinguish concurrent system into three big families: single cores, parallel, and distributed.

### *3.1.1 Single Core Systems*

Systems with only one computation unit can be considered concurrent systems if they allow the sequentialization of concurrent activities in a way that the developer, the application or the user can consider the activities as they are running simultaneously.

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_3, © Atlantis Press and the authors 2014

In these days, basically every single core computer can be considered a concurrent system since the interrupt mechanism and the operating system scheduler allow the user to consider the system "logically" concurrent.

A single core system is the simplest kind of concurrent system. However, since they have only one computational unit they do not support physical concurrency.

### *3.1.2 Parallel Systems*

Parallel systems are composed by tightly coupled computation units that work and are perceived as a single computer. Usually computation units share a common memory and the network that connects them is reliable, fast, and with a large bandwidth.

While in the past the improvements of the performances were obtained increasing the frequencies of the microprocessors now the current trend of semiconductor chip makers is to integrate in the same chip more than one CPU. These are the simplest parallel systems that can be think of. The two most known chip maker factories (Intel and Amd) are competing to increase the number of CPU on the same package and in the future we will have even 100 processing units per package.

More powerful parallel systems are instead the supercomputers used in research laboratories. They are at the front line of current processing capacity and they are obviously used to solve problems that require a huge computation power such as protein folding predictions, weather simulations, aerodynamic research, and nuclear test simulations.

The architecture of a supercomputer has varied through time. Starting from systems in which a few number of the same processors where connected together (symmetric multiprocessor or SMP) now the most powerful supercomputers have more than 100 thousands computing units. They usually cost millions of dollars to deploy and million of dollars to run (you can imagine what can be the power consumption of 100 thousand computation units).[1]

### *3.1.3 Distributed System*

We usually talk about a distributed system when a system is composed by a bunch of computational units that cannot be perceived as a unique computer. In this case, we do not require a fast and reliable net connecting the computing units, nor their tightly coupling or theirs presence in one site (e.g., a distributed system can be spread all over the world).

In the recent years, a distributed system is often been called "cloud," world that is used in the buzzword "cloud computing." There is no precise definition of what cloud or cloud computing is. One of the possible simplest definition describes a cloud computing as a group of technologies that allow the use of hardware and software

[1] A list of the most powerful supercomputers can be found at http://www.top500.org/

distributed over a network. Hence, the cloud is hardware and software distributed over a network. Another possible definition is that cloud computing is a paradigm shift whereby details are abstracted from the users who no longer have need of control over the technology infrastructure.

Since in the computer science world there is an ongoing interest in cloud and cloud computing we would like to spend some more time presenting this concept reporting one of the more precise definition of cloud computing that we have found. The following definitions have been provided by the National Institute of Standards and Technology (http://csrc.nist.gov).

**Definition 3.1 (Cloud computing).** Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.

Essential Characteristics:

- On-demand self-service. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.
- Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs).
- Resource pooling. The provider's computing resources are pooled to serve multiple consumers using a multitenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.
- Rapid elasticity. Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.
- Measured service. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

Service Models:

- Cloud Software as a Service (SaaS). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications

are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.
- Cloud Platform as a Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.
- Cloud Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

Deployment Models:

- Private cloud. The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.
- Community cloud. The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.
- Public cloud. The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
- Hybrid cloud. The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).

## 3.2 Concurrent System Problems

When we are dealing with a concurrent system, we face problems that have no match in the sequential world. The main challenge is solving race conditions that arise when there is a common resource that is required simultaneously by two or more processes but can be obtained by only one of them.

Race conditions cause

- poor performances. If a task gets the resource, then the others tasks should wait until the resource becomes available.
- nondeterminism. Sometimes the status of a system depends on the order in which the tasks have used the resource.

A good practice for a concurrent system developer is to avoid race conditions whenever this is possible. Unfortunately, some race conditions can not be avoided like, for instance, in parallel system with shared memory where it is impossible to avoid race conditions on the memory resource in the most general case. Even when the dual approach of shared memory, namely message passing, the problem cannot be avoided since there will be race conditions over the network resource.

The problem of race conditions has been widely studied in the literature and a lot of techniques have been developed to deal with it. Usually we use the term mutual exclusion to describe algorithms that are used to avoid the simultaneous use of a common resource. Starting from the 1960s, even hardware was design to allow the use of mutual exclusion algorithms that can be divided into two categories:

- busy-wait solutions in which a process repeatedly checks to see if the resource is available. Examples of these solutions are Dekker's algorithm and Peterson's algorithm.
- hardware supported solutions like locks, mutex, semaphores and monitors.

Mutual exclusion algorithms should be used carefully since when many forms of mutual exclusion are used it is possible to have negative side effects.

One of the most encountered problems is the deadlock that happens when two or more processors are waiting for the other to finish, and thus neither of them ever finish. A simple deadlock between two processes $p_1$ and $p_2$ happens for instance when $p_1$ needs a resource possessed by $p_2$ which in turns need a resource held by $p_1$.

Other side effects of the use of mutual exclusion algorithms is starvation and priority inversion. The former happens when a process never gets sufficient resources to run to completion while the latter happens when a process with higher priority waits for a lower-priority process.

Minimizing race condition is a very important task for a concurrent system developer. This however is not easy and sometimes a lot of knowledge of the system need to be used to obtain good performances out of a concurrent system. For example, concurrent algorithms have been sometimes optimized paying attention to the distribution and the latency of the memories of a concurrent system.

## 3.3 Mathematical Models

Concurrency theory has been an active field of research in theoretical computer science and a lot of models have been proposed for modeling and understanding concurrent systems. We will describe two of the most studied approach, namely Petri nets and Process algebras.

### 3.3.1 Petri Net

Petri net is one of the most popular and old formal model for the representation and analysis of concurrent system. It is due to C.A. Petri, who introduced it in his doctoral dissertation in 1962.[2]

A Petri net is a directed graph with two types of nodes, namely places and transitions. The arcs run from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition. The places to which arcs run from a transition are called the output places of the transition. Places may contain a natural number of tokens.

We say that a transition may fire whenever there is a token in all its input arcs. When a transition is fired, it consumes those tokens and places a token at the end of all output arcs. Execution of Petri nets is nondeterministic, i.e., when multiple transitions are enabled at the same time, any one of them may fire.

Petri nets are still used today in a lot of different fields of human knowledge. Their simple formal definition allows the modeling of concurrent systems. Moreover, a lot of interesting proprieties like the possibility of reaching a certain state were proven to be decidable. Petri nets can therefore be very useful not only for modeling a system but also to prove that some propriety of the system is decidable.

### 3.3.2 Process Calculus

Process calculi (the plural of process calculus) or process algebras are a family of languages for modeling concurrent systems. The history of process algebra traces back to the early 1970s when it was found that it was difficult to use the methods of denotational, operational, or axiomatic semantics to describe the semantics of programs containing a parallel operator. To solve this problem new innovative languages, the process calculi, have been created. The first and the most famous process calculus is CCS introduced by Milner in 1980. Among the languages proposed after CCS we should mention CSP and the more recent $\pi$-calculus and ambient calculus.

It is quite impossible to describe every process algebra presented so far. We can therefore only focus on some features that, as pointed out in [104], all process calculi have in common. These features are:

- the primitive elements of the language are process that describe the behavior of the system.
- the representation of interactions between independent processes is described as communications rather than modification of shared variables.

[2] Actually Petri nets were originally invented for describing chemical processes in August 1939 when Petri was only 13- year old.

- a small collection of primitives and operators are used to describe processes and systems. The basic operators, always present in some form are:
  - parallel composition
  - operator for sending and receiving data
  - sequentialization
  - recursion or process replication
- algebraic laws are defined for the process operators. These laws allow process expressions to be manipulated using equational reasoning.

## 3.4 Concurrent Programming

Concurrent programming encompasses the programming languages and algorithms used in concurrent systems. Concurrent programming is usually considered to be more general than parallel programming because it can involve arbitrary and dynamic patterns of communication and interaction, whereas parallel systems generally have a predefined and well-structured communication patterns. The basic goals of concurrent programming include correctness, performance, and robustness. Concurrent systems such as operating systems are generally designed to operate indefinitely and not terminate unexpectedly.

As mentioned in [12], the main challenge for designing a concurrent programs is to ensure the correct sequencing of the interactions between different computational processes and coordinate the access to resources that are shared among processes.

For some particular and simple concurrent systems, it is possible to have a developing framework that allows the programmer to write a program like in the sequential case. The developing framework can distribute the work on the concurrent system to improve the performances (see, for instance, the OpenMP project at http://openmp.org/wp/). Having this kind of developing tools on one hand facilitates the writing of the code but, on the other hand, the lack of flexibility makes sometimes impossible to reach optimal performances. In these cases, in order to improve the performances, programming languages that use language constructs for concurrency need to be used.

Today, the most commonly used programming languages that have specific constructs for concurrency are Java and C#. Both of these languages fundamentally use a shared-memory concurrency model, with locking provided by monitors or message-passage. Of the languages that use a message-passing concurrency model, Erlang is probably the most widely used in industry at present.

Unfortunately, only few of other concurrent languages have been used in industries. Indeed, the majority of concurrent languages have been developed as research languages. A nonexhaustive list of languages with concurrency operators is: ActorScript, Ada, Concurrent Haskell, Concurrent ML, Concurrent Pascal, Go, MultiLisp, Linda, occam, occam-$\pi$, Oz, and Scala.

Note that concurrent programs can also be executed sequentially on a single processor by interleaving the execution steps of each computational process.

## 3.5 Service-Oriented Computing

According to the W3C Working Group, a service is "an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent."[3]

Service-oriented computing (SoC) [61] is an emerging paradigm for programming distributed systems in which services are first class entities that can be composed to obtain more complex services for developing massively distributed applications.

In SoC interactions are no longer based on fixed or programmed exchanges of products with specific parties but on the provisioning of services by external providers that are procured on the fly. The processes of discovery and selection of services are not coded (at design time) but performed by the middleware according to some user functional and nonfunctional requirements. The process of binding the client application and the selected service is not performed by skilled software developers but by the middleware.

In this setting, there is a need to rethink the way we engineer software applications, moving from the typical static scenario in which components are assembled to build a (more or less complex) system that is delivered to a customer, to a more dynamic scenario in which (smaller) applications are developed to run on such global computers and respond to business needs by interacting with services and resources that are globally available.

SoC brings to the front many aspects that have already been tackled in component-based development (see for instance [34]). However, differently from the component-based view that encompass a fixed system of components, SoC considers an evolving universe of software applications that service providers publish so that they can be discovered by (and bound to) business activities as they execute. For instance, if documents need to be exchanged as part of a loan application, the bank may rely on an external courier service instead of imposing a fixed one. In this case, a courier service would be discovered for each loan application that is processed taking into account the address to which the documents need to be sent, speed of delivery, reliability, and so on.

The added flexibility provided by SoC comes at a price: dynamic interactions impose the overhead of selecting the co-party at each invocation since the choice between invoking a service and calling a component is a decision that needs to be taken according to a given goal.

To develop a service-oriented architecture following the SoC paradigm, two different approaches can be used:

[3] This definition is available at http://www.w3.org/TR/ws-gloss/

- orchestration. When a new functionality is required, a new service is created. Like the conductor of an orchestra controls and conducts the musicians this new service controls the other services to obtain the desired functionality;
- choreography. In this context, we can compare the services as dancers. Like the dancers move all together to create a choreography the services work together to obtain the desired features of the system.

Some languages have been proposed for the orchestration approach. The most used and wildly known orchestrating language is Business Process Execution Language (BPEL), an OASIS standard executable language for specifying interactions with Web Services (see http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf for the last specification).

Another language worth mentioning is Jolie [94], the first full-fledged programming language based upon the service-oriented programming paradigm. In Jolie, everything is a service. It can be used to create new services from scratch and/or compose existing ones using ad hoc primitives.

Few languages that follow the choreography approach have been proposed instead. Among them the most famous one is certainly Web Services Choreography Description Language (WS-CDL) which is, still today, a W3C candidate recommendation (see http://www.w3.org/2002/ws/chor/ for more informations).

More information regarding SoC can be retrieved consulting the results of the Sensoria Project (http://www.sensoria-ist.eu/) which has studied this topic in detail from a practical and theoretical point of view.

# Part II
# Constraints in Concurrent Languages

# Chapter 4
# Constraint Handling Rules

The idea of approaching hard combinatorial optimization problems through a combination of search and constraint solving appeared first in logic programming. Despite the continued support of logic programming for constraint programmers, research efforts were initiated to import constraint technologies into other paradigms. Nowadays, constraints can for instance be easily used in imperative languages. However, constraints are equally well suited for modeling and supporting concurrency. In particular, concurrent computation can be seen for instance as agents that communicate and coordinate through a shared constraint store [50].

Importing constraint in existing languages raises some concerns:

- how easy is to use the new language ?
- how expressive is the new language ?
- how extensible is the new language ?

Each concern is intrinsically linked to the host language and has a direct impact on potential end-users. It is desirable to obtain a declarative reading of a high-level model statement that exploits the facilities of the host language. Extensibility is also crucial since it is important to support the addition of user-defined constraints and user-defined search procedures.

In this chapter, we first provide in Sect. 4.1 a brief overview of different concurrent languages having constraints as primitive building blocks. Among all the concurrent languages with constraints, we present in detail the language Constraint Handling Rule (CHR). In particular in Sects. 4.3 and 4.5 we recall the CHR syntax and two of its semantics that will be considered in Chaps. 5 and 6 . In Sect. 4.6, we describe an extension of CHR that increase the expressive power of CHR adding rules priorities.

## 4.1 Brief Background

At the end of the 1980s, concurrent constraint programming integrated ideas from Concurrent Logic Programming [113] and constraint logic programming (CLP) [67]:

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_4, © Atlantis Press and the authors 2014

- Maher [84] proposed the ALPS class of committed-choice languages
- a concurrent logic language based on Ueda's GHC [122] was used in the Japanese Fifth-Generation Computing Project
- Saraswat [110] introduced the ask-and-tell metaphor for constraint operations and the concurrent constraints (CC) language framework that permits both don't-care and don't-know non-determinism
- Smolka proposed a concurrent programming model Oz that subsumes functional and object-oriented programming [115].

Implemented concurrent constraint logic programming languages include AKL, CIAO, CHR, and Mozart (successor of Oz).

In concurrent constraint programming (CCP), the processes communicate via a shared constraint store. The main difference with respect to imperative programming languages concerns the notion of store, which represents the state of a system. In CCP, rather than containing variable instantiations, the store is a constraint that specifies partial information about the possible values the variables can take at any stage of the computation. Processes can interact with each other by adding a constraint if it is consistent with the store (tell action). Alternatively, a process can check if the store entails (implies) a given constraint (ask action) and, if this is not the case, it remains blocked until some concurrent process adds enough information to the store. Hence, as computation proceeds, more and more information is accumulated and the store is monotonically refined.

Several extensions of the pure CCP paradigm have been proposed. In timed CCP [19, 103, 109], processes cannot wait indefinitely for an event and, in case a timeout occurs, they must take an alternative action. The Soft CCP model [16] generalizes CCP to handle soft constraints: the novel idea is to parametrize tell and ask primitives with a preference level that is used to determine their success, failure, or suspension.

Recently, some efforts have been made to enrich nominal process calculi like the $\pi$-calculus [90, 91] with primitives for constraint handling. An example of this extension is the concurrent constraint $\pi$-calculus [25].

## 4.2 Constraint Handling Rules: Notation

In this thesis, we focus our attention to one of the concurrent constraint language: (CHR). CHR [15, 41, 42, 43] is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language.

We chose to study this language because it has constraints as first class primitives, it is simple and it has been implemented over logic, imperative, and functional languages (see http://www.cs.kuleuven.be/~dtai/projects/CHR/ for more details on the implementations).

In this chapter, we will give an overview of CHR syntax and its operational semantics following [33, 43].

We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called user-defined (or CHR) constraints. Therefore, we assume a signature $\Sigma$ on which program terms are defined and two disjoint sets of predicate symbols $\Pi_b$ for built-in and $\Pi_u$ for user-defined constraints.

**Definition 4.1** (*Built-in constraint*). A *built-in constraint* $p(t_1, \ldots, t_n)$ is an atomic predicate where $p$ is a predicate symbol from $\Pi_b$ and $t_1, \ldots, t_n$ are terms over the signature $\Sigma$.

For built-in constraints, we assume a (first order) theory $\mathcal{CT}$ which describes their meaning.

**Definition 4.2** (*User-defined constraint*). A *user-defined (or CHR) constraint* $p(t_1, \ldots, t_n)$ is an atomic predicate where $p$ is a predicate symbol from $\Pi_u$ and $t_1, \ldots, t_n$ are terms over the signature $\Sigma$.

We use $c, d$ to denote built-in constraints, $h, k$ to denote CHR constraints and $a, b, f, g$ to denote both built-in and user-defined constraints (we will call these generally constraints). The capital versions of these notations will be used to denote multisets of constraints. We also denote by *false* any inconsistent conjunction of constraints and with *true* the empty multiset of built-in constraints.

We will use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic constraints. We prefer to use multisets rather than sequences (as in the original CHR papers) because our results do not depend on the order of atoms in the rules. In particular, we will use this notation based on multisets in the syntax of CHR.

The notation $\exists_V \phi$, where $V$ is a set of variables, denotes the existential closure of a formula $\phi$ w.r.t. the variables in $V$, while the notation $\exists_{-V} \phi$ denotes the existential closure of a formula $\phi$ with the exception of the variables in $V$ which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in $\phi$. Finally, we denote by $\bar{t}$ and $\bar{X}$ a sequence of terms and of distinct variables, respectively.

In the following, if $\bar{t} = t_1, \ldots t_m$ and $\bar{t'} = t'_1, \ldots t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t'})$ represents the set of equalities $t_1 = t'_1, \ldots, t_m = t'_m$ if $p = p'$, and it is undefined otherwise. This notation is extended in the expected way to multiset of constraints. Moreover, we use $+\!+$ to denote sequence concatenation and $\uplus$ for multiset union.

We follow the logic programming tradition and indicate the application of a substitution $\sigma$ to a syntactic object $t$ by $\sigma t$.

To distinguish between different occurrences of syntactically equal constraints, a CHR constraint can be labeled by a unique identifier. The resulting syntactic object is called identified CHR constraint and is denoted by $k\#i$, where $k$ is a CHR constraint and $i$ is the identifier. We also use the functions defined as $\mathtt{chr}\,(k\#i) = k$ and $\mathtt{id}(k\#i) = i$, possibly extended to sets and sequences of identified CHR constraints in the obvious way.

$$\begin{array}{ll} \textit{reflexivity} & \texttt{leq}(X,Y) \iff X = Y \mid \textit{true} \\ \textit{antisymmetry} & \texttt{leq}(X,Y), \texttt{leq}(Y,X) \iff X = Y \\ \textit{transitivity} & \texttt{leq}(X,Y), \texttt{leq}(Y,Z) \Rightarrow \texttt{leq}(X,Z) \end{array}$$

**Fig. 4.1** A program for defining $\leq$ in CHR

## 4.3 CHR Program

A CHR program is defined as a sequence of three kinds of rules: simplification, propagation, and simpagation rules. Intuitively, simplification rewrites constraints into simpler ones, propagation adds new constraints which are logically redundant but may trigger further simplifications, simpagation combines in one rule the effects of both propagation and simplification rules. For simplicity we consider simplification and propagation rules as special cases of a simpagation rule. The general form of a simpagation rule is:

$$r \ @ \ H^k \backslash H^h \iff D \mid B$$

where $r$ is a unique identifier of a rule, $H^k$ and $H^h$ (the heads) are multisets of CHR constraints, $D$ (the guard) is a possibly empty multiset of built-in constraints and $B$ is a possibly empty multiset of (built-in and user-defined) constraints. If $H^k$ is empty then the rule is a simplification rule. If $H^h$ is empty then the rule is a propagation rule. At least one of $H^k$ and $H^h$ must be non empty.

In the following, when the guard $D$ is empty or *true* we omit $D \mid$. Also the names of rules are omitted when not needed. For a simplification rule, we omit $H^k \backslash$ while we write a propagation rule as $H^k \Rightarrow D \mid B$. A CHR *goal* is a multiset of (both user-defined and built-in) constraints. An example of a CHR program is shown in Fig. 4.1. This program implements the less or equal predicate, assuming that we have only the equality predicate in the available built-in constraints. The first rule, a simplification, deletes the constraint leq($X$, $Y$) if $X = Y$. Analogously the second rule deletes the constraints leq($X$, $Y$) and leq($Y$, $X$) adding the built-in constraint $X = Y$. The third rule of the program is a propagation rule and it is used to add a constraint leq($X$, $Z$) when the two constraints leq($X$, $Y$) and leq($Y$, $Z$) are found.

## 4.4 Traditional Operational Semantics

The theoretical operational semantics of CHR, denoted by $\omega_t$, is given in [33] as a state transition system $T = (\text{Conf}, \overset{\omega_t}{\rightarrow}_P)$: Configurations in Conf are tuples of the form $\langle G, S, B, T \rangle_n$, where $G$ is the goal (a multiset of constraints that remain to be solved), $S$ is the CHR store (a set of identified CHR constraints), $B$ is the built-in store (a conjunction of built-in constraints), $T$ is the propagation history (a set of sequence of identifiers used to store the rule instances that have fired) and $n$ is the next free identifier (it is used to identify new CHR constraints). The propagation

**Table 4.1** Transitions of $\omega_t$

**Solve** $\langle (c, G), S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle G, S, c \wedge C, T\rangle_n$ where $c$ is a built-in constraint

**Introduce** $\langle (k, G), S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle G, \{c\#n\} \cup S, C, T\rangle_{n+1}$ where $k$ is a CHR constraint

**Apply** $\langle G, H_1 \cup H_2 \cup S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle (B, G), H_1 \cup S, \theta \wedge D \wedge C, T \cup \{t\}\rangle_n$ where $P$ contains a (renamed apart) rule

$$r \ @ \ H_1' \backslash H_2' \iff D \mid B$$

and there exists a matching substitution $\theta$ s.t. $\mathtt{chr}(H_1) = \theta H_1', \mathtt{chr}(H_2) = \theta H_2', \mathcal{CT} \models C \rightarrow \exists_{-Fv(C)}(\theta \wedge D)$ and $t = \mathtt{id}(H_1) +\!\!+ \mathtt{id}(H_2) +\!\!+ [r] \notin T$

history is used to avoid trivial nontermination that could be introduced by repeated application of the same propagation rule. The transitions of $\omega_t$ are shown in Table 4.1.

Given a program $P$, the transition relation $\xrightarrow{\omega_t}_P \subseteq \text{Conf} \times \text{Conf}$ is the least relation satisfying the rules in Table 4.1. The **Solve** transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. The **Introduce** transition is used to move a user-defined constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The **Apply** transition allows to rewrite user-defined constraints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable name clashes, this transition assumes that all variables appearing in a program clause are fresh ones. The **Apply** transition is applicable when the current store ($B$) is strong enough to entail the guard of the rule ($D$), once the parameter passing has been performed. Note also that, as previously mentioned, the condition $\mathtt{id}(H_1) +\!\!+ \mathtt{id}(H_2) +\!\!+ [r] \notin T$ avoids repeated application of the same propagation rule and therefore trivial nontermination.

An *initial configuration* has the form $\langle G, \emptyset, \mathit{true}, \emptyset\rangle_1$ while a *final configuration* has either the form $\langle G, S, \mathit{false}, T\rangle_k$, when it is *failed*, or the form $\langle \emptyset, S, B, T\rangle_k$, when it is successfully terminated because there are no applicable rules.

Given a goal $G$, the operational semantics that we consider observes the non failed final stores of terminating computations. This notion of observable is the most used in the CHR literature and is captured by the following.

**Definition 4.3** (*Qualified answers* [43]) Let $P$ be a program and let $G$ be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query $G$ in the program $P$ is defined as:

$$\mathcal{QA}_P(G) = \{\exists_{-Fv(G)}(K \wedge d) \mid \mathcal{CT} \not\models d \leftrightarrow \mathit{false}, \langle G, \emptyset, \mathit{true}, \emptyset\rangle_1 \xrightarrow{\omega_t}_P{}^{*} \langle \emptyset, K, d, T\rangle_n \not\xrightarrow{\omega_t}_P\}$$

We also consider the following different notion of answer, obtained by computations terminating with a user-defined constraint which is empty. We call these observables *data sufficient answers* slightly deviating from the terminology of [43] (a goal which has a data sufficient answer is called a data-sufficient goal in [43]).

**Table 4.2** Transitions of $\omega_a$

| | |
|---|---|
| **Solve** | $\langle (c, G), S, C, T\rangle_n \xrightarrow{\omega_a}_P \langle G, S, c \wedge C, T\rangle_n$ where $c$ is a built-in constraint |
| **Introduce** | $\langle (k, G), S, C, T\rangle_n \xrightarrow{\omega_a}_P \langle G, \{c\#n\} \cup S, C, T\rangle_{n+1}$ where $k$ is a CHR constraint |
| **Apply** | $\langle G, H_1 \cup H_2 \cup S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle (B, G), H_1 \cup S, \theta \wedge D \wedge C, T \cup \{t\}\rangle_n$ where $P$ contains a (renamed apart) rule $r \ @ \ H_1' \backslash H_2' \iff D \mid B$ and there exists a matching substitution $\theta$ s.t. $\mathtt{chr}(H_1) = \theta H_1'$, $\mathtt{chr}(H_2) = \theta H_2'$, $\mathcal{CT} \models C \rightarrow \exists_{-Fv(C)}(\theta \wedge D)$ |

**Definition 4.4** (*Data sufficient answers*) Let $P$ be a program and let $G$ be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query $G$ in the program $P$ is defined as:

$$\mathcal{SA}_P(G) = \{\exists_{-Fv(G)}(d) \mid \ \mathcal{CT} \not\models d \leftrightarrow \mathit{false}, \\ \langle G, \emptyset, \mathit{true}, \emptyset\rangle_1 \xrightarrow{\omega_t}_P{}^* \langle \emptyset, \emptyset, d, T\rangle_n\}$$

Both previous notions of observables characterize an input/output behavior, since the input constraint is implicitly considered in the goal. Clearly in general $\mathcal{SA}_P(G) \subseteq \mathcal{QA}_P(G)$ holds, since data sufficient answers can be obtained by setting $K = \emptyset$ in Definition 4.3.

## 4.5 Abstract Operational Semantics

The first CHR operational semantics defined in [43] differs from the traditional semantics $\omega_t$. Indeed this original, so-called, abstract semantics denoted by $\omega_a$, allows the firing of a propagation rules an infinite number of times. For this reason $\omega_a$ can be seen as the abstraction of the traditional semantics where the propagation history is not considered. In Table 4.2 we have reported the transaction of the $\omega_a$ semantics following the structure of the theoretical semantics using configurations without a propagation history set.

Given a program $P$, the transition relation $\xrightarrow{\omega_a}_P \subseteq \text{Conf} \times \text{Conf}$ is the least relation satisfying the rules in Table 4.2.

Initial and final configurations can be defined analogously to those of $\omega_t$ semantics. In the same way, we can define the observables: qualified and data sufficient answers.

## 4.6 CHR with Priorities

De Koninck et al. [75] extended CHR with user-defined priorities. This new language, denoted by CHR$^{\omega_p}$, provides a high-level alternative for controlling program execution, that is more appropriate to needs of CHR programmers than other low-level approaches.

**Table 4.3** Transitions of $\omega_p$

| | |
|---|---|
| **Solve** | $\langle (c, G), S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle G, S, c \wedge C, T\rangle_n$ where $c$ is a built-in constraint |
| **Introduce** | $\langle (k, G), S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle G, \{c\#n\} \cup S, C, T\rangle_{n+1}$ where $k$ is a CHR constraint |
| **Apply** | $\langle \emptyset, H_1 \cup H_2 \cup S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle B, H_1 \cup S, \theta \wedge D \wedge C, T \cup \{t\}\rangle_n$ where $P$ contains a (renamed apart) rule $p :: r \ @ H_1' \backslash H_2' \iff D \mid B$ and there exists a matching substitution $\theta$ s.t. $\mathtt{chr}(H_1) = \theta H_1'$, $\mathtt{chr}(H_2) = \theta H_2'$, $\mathcal{CT} \models C \rightarrow \exists_{-Fv(C)}(\theta \wedge D)$, $\theta p$ is a ground arithmetic expression and $t = \mathtt{id}(H_1) +\!\!+ \mathtt{id}(H_2) +\!\!+ [r] \notin T$. Furthermore, no rule of priority $p'$ and substitution $\theta'$ exists with $\theta' p' < \theta p$ for which the above conditions hold |

$$1 :: \mathtt{source}(V) \Longrightarrow \mathtt{dist}(V, 0)$$
$$1 :: \mathtt{dist}(V, D_1) \backslash \mathtt{dist}(V, D_2) \iff D_1 \leq D_2 \mid true$$
$$D + 2 :: \mathtt{dist}(V, D), \mathtt{edge}(V, C, U) \Longrightarrow \mathtt{dist}(U, D + C)$$

**Fig. 4.2** A program for computing the shortest path in CHR$^{\omega_p}$

The syntax of CHR with priorities is compatible with the syntax of CHR. A simpagation rule has now the form

$$p :: r \ @ H^k \backslash H^h \iff D \mid B$$

where $r$, $H^k$, $H^h$, $D$, $B$ are defined as in the CHR simpagation rule in Sect. 4.3, while $p$ is an arithmetic expression, with $Fv(p) \subseteq (Fv(H^k) \cup Fv(H^h))$, which expresses the priority of rule $r$. If $Fv(p) = \emptyset$ then $p$ is a static priority, otherwise it is called dynamic.

The formal semantics of CHR$^{\omega_p}$, defined by [75], is an adaptation of the traditional semantics to deal with rule priorities. Formally this semantics, denoted by $\omega_p$, is a state transition system $T = (\text{Conf}, \overset{\omega_p}{\rightarrow}_P)$ where $P$ is a CHR$^{\omega_p}$ program while configurations in Conf, as well as the initial and final configurations, are the same as those introduced for the traditional semantics in Sect. 4.4. The transition relation $\overset{\omega_p}{\rightarrow}_P \subseteq \text{Conf} \times \text{Conf}$ is the least relation satisfying the rules in Table 4.3. The **Solve** and **Introduce** transitions are equal to those defined for the traditional semantics. The **Apply** transition instead is modified in order to take into account priorities. In fact, a further condition is added imposing that a rule can be fired only if no other rule that can be applied has a smaller value for the priority annotation (as usual in many systems, smaller values correspond to higher priority; For simplicity in the following we will use the terminology "higher" or "lower" priority rather than considering the values).

An example of a CHR$^{\omega_p}$ program (from [75]) is shown in Fig. 4.2. This program can be used to compute the length of the shortest path between a source node and all the other nodes in the graph. We assume that the source node $n$ is defined by using the constraint source($n$) and that the graph is represented by using the constraints edge($V$, $C$, $U$) for every edge of length $C$ between two nodes $V$, $U$. When the program terminates, we obtain a constraint dist($U$, $C$) if the length of the shortest path between the source node and $U$ is $C$.

The qualified and data sufficient answers for CHR$^{\omega_p}$ can be defined analogously to those of the standard language:

**Definition 4.5** (*Qualified answers*) Let $P$ be a CHR$^{\omega_p}$ program and let $G$ be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query $G$ in the program $P$ is defined as:

$$\mathcal{QA}_P(G) = \{\exists_{-Fv(G)}(K \wedge d) \mid \mathcal{CT} \not\models d \leftrightarrow \mathit{false}, \langle G, \emptyset, \mathit{true}, \emptyset\rangle_1 \overset{\omega_p}{\rightarrow}{}_P^{\,*} \langle \emptyset, K, d, T\rangle_n \not\overset{\omega_p}{\rightarrow}_P\}$$

**Definition 4.6** (*Data sufficient answers*) Let $P$ be a CHR$^{\omega_p}$ program and let $G$ be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query $G$ in the program $P$ is defined as:

$$\mathcal{SA}_P(G) = \{\exists_{-Fv(G)}(d) \mid \mathcal{CT} \not\models d \leftrightarrow \mathit{false}, \langle G, \emptyset, \mathit{true}, \emptyset\rangle_1 \overset{\omega_p}{\rightarrow}{}_P^{\,*} \langle \emptyset, \emptyset, d, T\rangle_n\}$$

# Chapter 5
# Non Turing Powerful Fragments of CHR

Given the spread of small computing device, it can be very useful to implement a language like CHR that runs concurrently in a given environment. In the 2009 CHR working week held in Ulm, we discussed the possibility of studying and implementing a simple version of CHR that can run on a huge amount of small devices like smart phones or in parallel using a multiparallel graphics accelerator like nVidia CUDA.[1] Using the full CHR language has been considered too complex to run on such simple devices and therefore the study of the expressive power of CHR fragments can help us to decide which is the right sublanguage to implement.

In the last few years, several papers have been devoted to investigate the expressivity of CHR, however very few decidability results for fragments of CHR have been obtained. Three main aspects affect the computational power of CHR: the number of atoms allowed in the heads, the nature of the underlying signature on which programs are defined, and the constraint theory, defining the meaning of built-ins. Some results in [32] indicate that when restricting to single-headed rules the computational power of CHR decreases. However, these results consider Turing complete fragments of CHR, hence they do not establish any decidability result. Indeed, single-headed CHR is Turing-complete [32], provided that the host language allows functors and supports unification. On the other hand, when allowing multiple heads, even restricting to a host language which allows only constants does not allow to obtain any decidability property, since also with this limitation CHR is still Turing complete [32, 116]. The only (implicit) decidability results concern propositional CHR, where all constraints have arity 0, and CHR without functors and without unification, since these languages can be translated to (colored) Petri nets [13].

Given this situation, when looking for decidable properties it is natural to consider further restrictions of the above-mentioned CHR language which allows the only built-in $=$ (interpreted in the usual way as equality on the Herbrand universe) and which, similarly to Datalog, is defined over a signature which contains no function symbols of arity $>0$.

[1] More information regarding these two ongoing projects can be found at http://www.uni-ulm.de/en/in/pm/teaching/tasks/cp-and-chr.html.

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_5, © Atlantis Press and the authors 2014

In this chapter, we will study the decidability of termination for two CHR dialects which, similarly to the Datalog-like languages, are defined by using a signature which does not allow function symbols (of arity $>0$). Both languages allow the use of the $=$ built-in in the body of rules, thus are built on a host language that supports unification. However, each imposes one further restriction. The first CHR dialect allows only *range-restricted* rules, that is, it does not allow the use of variables in the body or in the guard of a rule if they do not appear in the head. We show, using the theory of well-structured transition systems (WSTS) [6, 38], that the existence of an infinite computation is decidable in this dialect. The second dialect instead limits the number of atoms in the head of rules to one. We prove that in this case, the existence of a terminating computation is decidable. In this case, we provide a direct proof, since no reduction to Petri nets can be used (the language introduces an infinite states system) and well-structured transition system cannot be used (they do not allow to prove this kind of decidability properties). These results show that both dialects are strictly less expressive[2] than Turing machines. It is worth noting that the language (without function symbols) without these restrictions is as expressive as Turing machines.

## 5.1 Notation

As mentioned before, the computational power of CHR depends on several aspects, including the number of atoms allowed in the heads, the underlying signature $\Sigma$ on which programs are defined, and the constraint theory $\mathcal{CT}$, defining the built-ins.

In particular, the language under consideration in this chapter is the CHR defined over a signature which contains no function symbol of arity $>0$ and interpreted using the $\omega_a$ semantics. We will indicate this language as $\text{CHR}^{\omega_a}(C)$.

We will also use the notation $\text{CHR}^{\omega_a}(P)$ to denote the language where all constraints have arity zero (i.e., $\Sigma = \emptyset$). Finally, $\text{CHR}^{\omega_a}(F)$ indicates the CHR language which allows functor symbols and the $=$ built-in. Note that this last language is the signature used in most of the current CHR implementation. Indeed the host language of the majority of CHR implementations is Prolog and therefore the usual signature supports arbitrary Herbrand terms and unification.

The number of atoms in the heads also affects the expressive power of the language. We use the notation $\text{CHR}_1$, possibly combined with the notation above, to denote *single-headed* CHR, where heads of rules contain one atom.

[2] As we clarify later, "less expressive" here means that there exists no termination preserving encoding of Turing machines in the considered language.

## 5.2 Range-Restricted CHR$^{\omega_a}(C)$

In this section, we consider the (multiheaded) range-restricted CHR$^{\omega_a}(C)$ language described in the introduction. We call a CHR rule range-restricted if all the variables which appear in the body and in the guard appear also in the head of a rule. More formally, if Var $(X)$ denotes the variables used in $X$, the rule $r \ @ \ H^k \backslash H^h \iff D \mid B$ is range-restricted if Var $(B) \cup$ Var $(D) \subseteq$ Var $(H^k, H^h)$ holds. A CHR language is called range-restricted if it allows range-restricted rules only.

We prove that in range-restricted CHR$^{\omega_a}(C)$ the existence of an infinite computation is a decidable property. This shows that this language is less expressive than Turing machines and than CHR$^{\omega_a}(C)$. Our result is based on the theory of WSTS and we refer to [6, 38] for this theory. Here we only provide the basic definitions on WSTS, taken from [38].

Recall that a *quasi-order* (or, equivalently, preorder) is a reflexive and transitive relation. A *well-quasi-order* (wqo) is defined as a quasi-order $\leq$ over a set $X$ such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in $X$, there exist indexes $i < j$ such that $x_i \leq x_j$.

A *transition system* is defined as usual, namely it is a structure TS$= (S, \rightarrow)$, where $S$ is a set of *states* and $\rightarrow \subseteq S \times S$ is a set of *transitions*. We define $Succ(s)$ as the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of $s$. We say that TS is *finitely branching* if, for each $s \in S$, $Succ(s)$ is finite. Hence, we have the key definition.

**Definition 5.1 (Well-structured transition system with strong compatibility)** A *well-structured transition system with strong compatibility* is a transition system TS$= (S, \rightarrow)$, equipped with a quasi-order $\leq$ on $S$, such that the two following conditions hold:

1. $\leq$ is a well-quasi-order;
2. $\leq$ is strongly (upward) compatible with $\rightarrow$, that is, for all $s_1 \leq t_1$ and all transitions $s_1 \rightarrow s_2$, there exists a state $t_2$ such that $t_1 \rightarrow t_2$ and $s_2 \leq t_2$ holds.

The next theorem is a special case of a result in [38] and will be used to obtain our decidability result.

**Theorem 5.1** *Let $TS = (S, \rightarrow, \leq)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leq$ and computable $Succ(s)$ for $s \in S$. Then the existence of an infinite computation starting from a state $s \in S$ is decidable.*

**Decidability of divergence.** Consider a given goal $G$ and a (CHR) program $P$ and consider the transition system $T = (\text{Conf}, \overset{\omega_a}{\rightarrow}_P)$ defined in Sect. 4.5. Obviously, the number of constants and variables appearing in $G$ or in $P$ is finite. Moreover, observe that since we consider range-restricted programs, the application of the transitions $\overset{\omega_a}{\rightarrow}_P$ does not introduce new variables in the computations. In fact, even though rules are renamed (in order to avoid clash of variables), the definition of the Apply rule (in particular, the definition of $\theta$) implies that in a transition $s_1 \overset{\omega_a}{\rightarrow}_P s_2$ we have that

Var $(s_2) \subseteq$ Var $(s_1)$ holds. Hence, an obvious inductive argument implies that no new variables arise in computations. For this reason, given a goal $G$ and a program $P$, we can assume that the set Conf of all the configurations uses only a finite number of constants and variables. In the following, we implicitly make this assumption. We define a quasi-order on configurations as follows:

**Definition 5.2** Given two configurations $s_1 = \langle G_1, S_1, B_1 \rangle_i$ and $s_2 = \langle G_2, S_2, B_2 \rangle_j$ we say that $s_1 \leq s_2$ if

- for every constraint $c \in G_1$ $|\{c \in G_1\}| \leq |\{c \in G_2\}|$
- for every constraint $c \in \{d \,.\, d\#i \in S_1\}$ $|\{i \,.\, c\#i \in S_1\}| \leq |\{i \,.\, c\#i \in S_2\}|$
- $B_1$ is logically equivalent to $B_2$.

The next Lemma, with proof in Appendix A, states the relevant property of $\leq$.

**Lemma 5.1** *$\leq$ is a well-quasi-order on* Conf

Next, in order to obtain our decidability results, we have to show that the strong compatibility property holds. This is the content of the following lemma whose proof is in Appendix A.

**Lemma 5.2** *Given a CHR$^{\omega_a}(C)$ program $P$, $(\text{Conf}, \overset{\omega_a}{\rightarrow}_P, \leq)$ is a well-structured transition system with strong compatibility.*

Finally we have the desired result.

**Theorem 5.2** *Given a range-restricted CHR$^{\omega_a}(C)$ program $P$ and a goal $G$, the existence of an infinite computation for $G$ in $P$ is decidable.*

*Proof* First observe that, due to our assumption on range-restricted programs, $T = (\text{Conf}, \overset{\omega_a}{\rightarrow}_P)$ is finitely branching. In fact, as previously mentioned, the use of rule Apply cannot introduce new variables (and hence new different states). The thesis follows immediately from Lemma 5.2 and Theorem 5.1.

The previous Theorem implies that range-restricted CHR$^{\omega_a}(C)$ is strictly less expressive than Turing machines, in the sense that there cannot exist a termination preserving encoding of Turing machines into range-restricted CHR$^{\omega_a}(C)$. To be more precise, we consider an encoding of a Turing machine into a CHR language as a function $f$ which, given a machine $Z$ and an initial instantaneous description $D$ for $Z$, produces a CHR program and a goal. This is denoted by $(P, G) = f(Z, D)$. Hence we have the following.

**Definition 5.3 (Termination preserving encoding)** An encoding $f$ of Turing Machines into a CHR language is termination preserving[3] if the following holds: the machine $Z$ starting with $D$ terminates if the goal $G$ in the CHR program $P$ has only terminating computations, where $(P, G) = f(Z, D)$. The encoding is weak

[3] For many authors the existence of a termination preserving encoding into a nondeterministic language $L$ is equivalent to the Turing completeness of $L$, however there is no general agreement on this, since for others a weak termination preserving encoding suffices.

termination preserving if: the machine $Z$ starting with $D$ terminates if the goal $G$ in the CHR program $P$ has at least one terminating computation.

Since termination is undecidable for Turing machines, we have the following immediate corollary of Theorem 5.2.

**Corollary 5.1** *There exists no termination preserving encoding of Turing machines into range-restricted CHR$^{\omega_a}(C)$.*

Note that the previous result does not exclude the existence of weak encodings. For example, in [28] it is showed that the existence an infinite computation is decidable in CCS!, a variant of CCS, yet it is possible to provide a weak termination preserving encoding of Turing machines in CCS! (essentially by adding spurious nonterminating computations). We conjecture that such an encoding is not possible for CHR$^{\omega_a}(C)$. Note also that previous results imply that range-restricted CHR$^{\omega_a}(C)$ is strictly less expressive than CHR$^{\omega_a}(C)$: in fact there exists a termination preserving encoding of Turing machines into CHR$^{\omega_a}(C)$ [32, 116].

## 5.3 Single-Headed CHR$^{\omega_a}(C)$

As mentioned in the Introduction, while CHR$^{\omega_a}(C)$ and CHR$_1^{\omega_a}(F)$ are Turing complete languages [32, 116], the question of the expressive power of CHR$_1^{\omega_a}(C)$ is open. Here we answer to this question by proving that the existence of a terminating computation is decidable for this language, thus showing that CHR$_1^{\omega_a}(C)$ is less expressive than Turing machines. Throughout this section, we assume that the abstract semantics $\omega_a$ is considered (however, see the discussion at the end for an extension to the case of $\omega_t$). The proof we provide is a direct one, since neither WSTS nor reduction to Petri nets can be used here (see the Introduction).

### *5.3.1 Some Preparatory Results*

We introduce here two more notions, namely the forest associated to a computation and the notion of reactive sequence, and some related results. We will need them for the main result of this section.

First, we observe that it is possible to associate to the computation for an atomic goal $G$ in a program $P$ a tree where, intuitively, nodes are labeled by constraints (recall that these are atomic formulae), the root is $G$ and every child node is obtained from the parent node by firing a rule in the program $P$. This notion is defined precisely in the following, where we generalize it to the case of a generic (nonatomic) goal, where for each CHR constraint in the goal we have a tree. Thus, we obtain a *forest* $F_\delta = (V, E)$ associated to a computation $\delta$, where $V$ contains a node for each

repetition of identified CHR constraints in $\delta$. Before defining the forest, we need the concept of *repetition* of an identified CHR atom in a computation.

**Definition 5.4 (Repetition)** Let $P$ be a CHR program and let $\delta$ be a computation in $P$. We say that an occurrence of an identified CHR constraint $h\#l$ in $\delta$ is the $i$-th repetition of $h\#l$, denoted by $h\#l^i$, if it is preceded in $\delta$ by $i$ **Apply** transitions of propagation rules whose heads match the atom $h\#l$. We also define

$$r(\delta, h\#l) = max\{i \mid \text{there exists a } i\text{-th repetition of } h\#l \text{ in } \delta\}$$

**Definition 5.5** (**Forest**) Let $\delta$ be a terminating computation for a goal in a $\text{CHR}_1^{\omega_a}(C)$ program. The forest associated to $\delta$, denoted by $F_\delta = (V, E)$ is defined as follows. $V$ contains nodes labeled either by repetitions of identified CHR constraints in $\delta$ or by $\Box$. $E$ is the set of edges. The labeling and the edges in $E$ are defined as follows:

(a) For each CHR constraint $k$ which occurs in the first configuration of $\delta$ there exists a tree in $F_\delta = (V, E)$, whose root is labeled by a repetition $k\#l^0$, where $k\#l$ is the identified CHR constraint associated to $k$ in $\delta$.
(b) If $n$ is a node in $F_\delta = (V, E)$ labeled by $k\#l^i$ and the rule $r\ @\ h \odot g \mid C, k_1, \ldots, k_m$ is used in $\delta$ to rewrite the repetition $h\#l^i$, where $\odot \in \{\Longleftrightarrow, \Longrightarrow\}$, the $k_i's$ are CHR constraints while $C$ contains built-ins, then we have two cases:

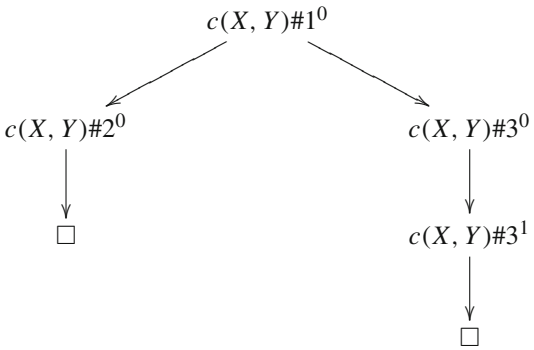
1. If $\odot$ is $\Longrightarrow$ then $n$ has $m + 1$ sons, labeled by ${k_j\#l_j}^0$, for $j \in [1, m]$, and by $h\#l^{i+1}$, where the ${k_j\#l_j}^0$ are the repetitions generated by the application of the rule $r$ to $h\#l^i$ in $\delta$.
2. If $\odot$ is $\Longleftrightarrow$ then:
   - if $m > 0$ then $n$ has $m$ sons, labeled by ${k_j\#l_j}^0$, for $j \in [1, m]$, where ${k_j\#l_j}^0$ are the repetitions generated by the application of the rule $r$ to $h\#l^i$ in $\delta$.
   - if $m = 0$ then $n$ has 1 son, labeled by $\Box$.

Note that, according to the previous definition, nodes which are not leaves are labeled by repetitions of identified constraints $k\#l^i$, where either $i < r(\delta, h\#l)$ or $h\#l$ does not occur in the last configuration of $\delta$. On the other hand, the leaves of the trees in $F_\delta$ are labeled either by $\Box$ or by the repetitions which do not satisfy the condition above. An example can help to understand this crucial definition.

*Example 5.1* Let us consider the following program $P$:

```
r1 @ c(X,Y) <=> c(X,Y),c(X,Y)
r2 @ c(X,Y) <=> X = 0
r3 @ c(0,Y) ==> Y = 0
r4 @ c(0,0) <=> true
```

There exists a terminating computation $\delta$ for the goal $c(X, Y)$ in the program $P$, which uses the clauses `r1`, `r2`, `r3`, `r4` in that order and whose associated forest $F_\delta$ is the following tree:




Note that the left branch corresponds to the termination obtained by using rule `r2`, hence the superscript is not incremented. On the other hand, in the right branch the superscript $^0$ at the second level becomes$^1$ at the third level. This indicates that a propagation rule (rule `r3`) has been applied.

Given a forest $F_\delta$, we write $T_\delta(n)$ to denote the subtree of $F_\delta$ rooted in the node $n$. Moreover, we identify a node with its label and we omit the specification of the repetition, when not needed. The following definition introduces some further terminology that we will need later.

**Definition 5.6** • Given a forest $F_\delta$, a path from a root of a tree in the forest to a leaf is called a *single constraint computation*, or *sc-computation* for short.

- Two repetitions $h\#l^i$ and $k\#m^j$ of identified CHR constraints are called r-equal, indicated by $h\#l^i == k\#m^j$, iff there exists a renaming $\rho$ such that $h = k\rho$.
- a sc-computation $\sigma$ is $p$-repetitive if $p = \max_{h\#l^i \in \sigma} |\{k\#m^j \in \sigma \mid h\#l^i == k\#m^j\}|$.
- The degree of a $p$-repetitive sc-computation $\sigma$, denoted by $dg(\sigma)$ is the cardinality of the set $P\_REP$ which is defined as the maximal set having the following properties:
  - contains a repetition $h\#l^i$ in $\sigma$ iff $p = |\{k\#m^j \in \sigma \mid h\#l^i == k\#m^j\}|$
  - if $h\#l^i$ is in $P\_REP$ then $P\_REP$ does not contain a repetition $k\#m^j$ s.t. $h\#l^i == k\#m^j$
- A forest $F_\delta$ is $l$-repetitive if one of its sc-computation $\sigma$ is $l$-repetitive and there is no $l'$-repetitive sc-computation $\sigma'$ in $F_\delta$ with $l' > l$.
- The degree $dg(F_\delta)$ of an $l$-repetitive forest $F_\delta$ is defined as

$$dg(F_\delta) = \sum_{\sigma} \{dg(\sigma) \mid \sigma \text{ is an } l\text{-repetitive sc-computation in } F_\delta\}.$$

After the forest, the second main notion that we need to introduce is that one of reactive sequence.[4]

Given a computation $\delta$, we associate to each (repetition of an) occurrence of an identified CHR atom $k\#l$ in $\delta$ a, so-called, reactive sequence of the form

$\langle c_1, d_1\rangle \ldots \langle c_n, d_n\rangle$, where, for any $i \in [1, n]$, $c_i$, $d_i$ are built-in constraints.

Intuitively each pair $\langle c_i, d_i\rangle$ of built-in constraints represents all the **Apply** transition steps, in the computation $\delta$, which are used to rewrite the considered occurrence of the identified CHR atom $k\#l$ and the identified atoms derived from it. The constraint $c_i$ represents the input for this sequence of **Apply** computation steps, while $d_i$ represents the output of such a sequence. Hence, one can also read such a pair as follows: the identified CHR constraint $k\#l$, in $\delta$, can transform the built-in store from $c_i$ to $d_i$. Different pairs $\langle c_i, d_i\rangle$ and $\langle c_j, d_j\rangle$ in the reactive sequence correspond to different sequences of **Apply** transition steps. This intuitive notion is further clarified later (Definition 5.9), when we will consider a reactive sequence associated to a repetition of an identified CHR atom.

Since in CHR computations the built-in store evolves monotonically, i.e., once a constraint is added it cannot be retracted, it is natural to assume that reactive sequences are monotonically increasing. So, in the follo wing we will assume that, for each reactive sequence $\langle c_1, d_1\rangle \ldots \langle c_n, d_n\rangle$, the following condition holds: $\mathrm{CT} \models d_j \rightarrow c_j$ and $\mathrm{CT} \models c_{i+1} \rightarrow d_i$ for $j \in [1, n]$, $i \in [1, n-1]$. Moreover, we denote the empty sequence by $\varepsilon$. Next, we define the strictly increasing reactive sequences w.r.t. a set of variables $X$.

**Definition 5.7 (Strictly increasing sequence)** Given a reactive sequence $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$, with $n \geq 0$ and a set of variables $X$, we say that $s$ is strictly increasing with respect to $X$ if the following holds for any $j \in [1, n]$, $i \in [1, n-1]$

- $Fv(c_j, d_j) \subseteq X$,
- $\mathrm{CT} \models d_i \not\rightarrow c_{i+1}$ and $\mathrm{CT} \models c_i \not\rightarrow d_i$.

Given a generic reactive sequence $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$ and a set of variables $X$, we can construct a new, strictly increasing sequence $\eta(s, X)$ with respect to a set of variables $X$ as follows. First the operator $\eta$ restricts all the constraints in $s$ to the variables in $X$ (by considering the existential closure with the exception of the variables in $X$). Then $\eta$ removes from the sequence all the stuttering steps (namely the pairs of constraints $\langle c, d\rangle$, such that $\mathrm{CT} \models c \leftrightarrow d$) except the last. Finally, in the sequence produced by the two previous steps, if there exists a pair of consecutive elements $\langle c_l, d_l\rangle\langle c_{l+1}, d_{l+1}\rangle$ which are "connected," in the sense that $c_{l+1}$ does not provide more information than $d_l$, then such a pair is "fused" in (i.e., replaced by) the unique element $\langle c_l, d_{l+1}\rangle$ (and this is repeated inductively for the new pairs). This is made precise by the following definition.

[4] This notion is similar to that one used in the (trace) semantics of concurrent languages, see, for example, [18, 20] for the case of concurrent constraint programming. The name comes from this field.

**Definition 5.8 (Operator $\eta$)** Let $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$ be a sequence of pairs of built-in stores and let $X$ be a set of variables. The sequence $\eta(s, X)$ is the obtained as follows:

1 First we define $s' = \langle c'_1, d'_1\rangle \cdots \langle c'_n, d'_n\rangle$, where for $j \in [1, n]$ $c'_j = \exists_{-X} c_j$ and $d'_j = \exists_{-X} d_j$.
2 Then we define $s''$ as the sequence obtained from $s'$ by removing each pair of the form $\langle c, d\rangle$ such that CT $\models c \leftrightarrow d$, if such a pair is not the last one of the sequence.
3 Finally we define $\eta(s, X) = s'''$, where $s'''$ is the closure of $s''$ w.r.t. the following operation: if $\langle c_l, d_l\rangle\langle c_{l+1}, d_{l+1}\rangle$ is a pair of consecutive elements in the sequence and CT $\models d_l \rightarrow c_{l+1}$ holds then such a pair is substituted by $\langle c_l, d_{l+1}\rangle$.

The following Lemma states a first useful property. The proof is in Appendix A.

**Lemma 5.3** *Let $X$ be a finite set of variables and let $s = \langle c_1, c_2\rangle \cdots \langle c_{n-1}, c_n\rangle$ be a strictly increasing sequence with respect to $X$. Then $n \leq |X| + 2$.*

Next we note that, given a set of variables $X$ the possible strictly increasing sequences w.r.t. $X$ are finite (up to l ogical equivalence on constraints), if the set of the constants is finite. This is the content of the following lemma, whose proof is in Appendix A. Here and in the following, with a slight abuse of notation, given two reactive sequences $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$ and $s' = \langle c'_1, d'_1\rangle \cdots \langle c'_n, d'_n\rangle$, we say that $s$ and $s'$ are equal (up to logical equivalence) and we write $s = s'$, if for each $i \in [1, n]$ CT $\models c_i \leftrightarrow c'_i$ and CT $\models d_i \leftrightarrow d'_i$ holds.

**Lemma 5.4** *Let $Const$ be a finite set of constants and let $S$ be a finite set of variables such that $u = |Const|$ and $w = |S|$. The set of sequences $s$ which are strictly increasing with respect to $S$ (up to logical equivalence) is finite and has cardinality at the most*

$$\frac{2^{w(u+w)(w+3)} - 1}{2^{w(u+w)} - 1}.$$

Finally, we show how reactive sequences can be obtained from a forest associated to a computation. First we need to define the reactive sequence associated to a repetition of an identified CHR atom in a computation. In this definition, we use the operator $\eta$ introduced in Definition 5.8.

**Definition 5.9** Let $\delta$ be a computation for a CHR$_1^{\omega_a}(C)$ program, $h\#l^j$ be a repetition of an identified CHR atom in $\delta$ and $r_1, \ldots, r_n$ the sequence of the **Apply** transition in $\delta$ that rewrite $h\#l^j$ and all the repetitions derived from it. If $s \xrightarrow{r_i}_P s'$ let $pair(r_i)$ be the pair $(\bigwedge B_1, \bigwedge B_2)$ where $B_1$ and $B_2$ are all the built-ins in $s$ and $s'$. We will denote with $seq(h\#l^j, \delta)$ the sequence $\eta(pair(r_1) \ldots pair(r_n), Fv(h))$.

Finally, we define the function $S_{F_\delta}$ which, given a node $n$ in a forest associated to a computation $\delta$ (see Definition 5.5), returns a reactive sequence. Such a sequence intuitively represents the sequence of the **Apply** transition steps, which have been used in $\delta$ to rewrite the repetition labeling $n$ and the repetitions derived from it.

**Definition 5.10 (Sequence associated to a node in a forest)** Let $\delta$ be a terminating computation and let $F_\delta = (V, E)$ be the forest associated to it. Given a node $n$ in $F_\delta$ we define:

- if the label of $n$ is $h\#l^i$, then $S_{F_\delta}(n) = seq(h\#l^i, \delta)$;
- if the label of $n$ is $\Box$ then $S_{F_\delta}(n) = \varepsilon$.

*Example 5.2* Let us consider for instance the forest shown in Example 5.1. The sequences associated to the nodes of this forest are:

- $S_{F(\delta)}(c(X, Y)\#1^0) = \langle true, X = 0 \wedge Y = 0\rangle$
- $S_{F(\delta)}(c(X, Y)\#2^0) = \langle true, X = 0\rangle$
- $S_{F(\delta)}(c(X, Y)\#3^0) = \langle X = 0, X = 0 \wedge Y = 0\rangle$
- $S_{F(\delta)}(c(X, Y)\#3^1) = \langle X = 0 \wedge Y = 0, X = 0 \wedge Y = 0\rangle$.

### 5.3.2 Decidability of Termination

We are now ready to prove the main result of this chapter. First, we need the following Lemma which has some similarities to the pumping lemma of regular and context free grammars. Indeed, if the derivation is seen as a forest, this lemma allows us to compress a tree if in a path of the tree there are two r-equal constraints with an equal (up to renaming) sequence. The lemma is proved in Appendix A.

Here and in the following given a node $n$ in a forest $F$ we denote by $A_F(n)$ the label associated to $n$.

**Lemma 5.5** *Let $\delta$ be a terminating computation for the goal $G$ in the $CHR_1^{\omega_a}(C)$ program $P$. Assume that $F_\delta$ is $l$-repetitive with $p = dg(F_\delta)$ and assume that there exists an $l$-repetitive sc-computation $\sigma$ of $F_\delta$ and a repetition $k\#l^i \in \sigma$ such that $l = |\{h\#n^j \in \sigma \mid h\#n^j == k\#l^i\}|$.*
*Moreover assume that there exist two distinct nodes $n$ and $n'$ in $\sigma$ such that $n'$ is a node in $T_\delta(n)$, $A_{F_\delta}(n) = k\#l^i$, $A_{F_\delta}(n') = k'\#l'^{i'}$ and $\rho$ is a renaming such that $S_{F_\delta}(n) = S_{F_\delta}(n')\rho$ and $k = k'\rho$.*

*Then there exists a terminating computation $\delta'$ for the goal $G$ in the program $P$, such that either $F_{\delta'}$ is $l'$-repetitive with $l' < l$, or $F_{\delta'}$ is $l$-repetitive and $dg(F'_\delta) < p$.*

Finally we obtain the following result.

**Theorem 5.3 (Decidability of termination).** *Let $P$ be a $CHR_1^{\omega_a}(C)$ program an let $G$ be a goal. Let $u$ be the number of distinct constants used in $P$ and in $G$ and let $w$ be the maximal arity of the CHR constraints which occur in $P$ and in $G$.*

*$G$ has a terminating computation in $P$ if and only if there exists a terminating computation $\delta$ for $G$ in $P$ s.t. $F_\delta$ is $m$-repetitive and $m \leq \frac{2^{w(u+w)(w+3)}-1}{2^{w(u+w)}-1} = L$.*

*Proof* We prove only that if $G$ has a terminating computation in $P$ then there exists a terminating computation $\delta$ for $G$ in $P$ s.t. $F_\delta$ is $m$-repetitive and $m \leq L$. The proof of the converse is straightforward and hence it is omitted.

The proof is by contradiction. Assume $G$ has a terminating computation $\delta$ in $P$ s.t. $F_\delta$ is $m$-repetitive, $m > L$ and there is no terminating computation $\delta'$ for $G$ in $P$ such that $F_{\delta'}$ is $m'$-repetitive and $m' < m$. Moreover, without loss of generality, we can assume that the degree of $F_\delta$ is minimal, namely there is no terminating computation $\delta'$ for $G$ in $P$ such that $F_{\delta'}$ is $m$-repetitive and $dg(F_{\delta'}) < dg(F_\delta)$.

Let $\sigma$ be a $m$-repetitive sc-computation in $F_\delta$. By definition, there exist $m$ repetitions of identified CHR constraints $k_1\#{l_1}^{i_1}$, ..., $k_r\#{l_m}^{i_m}$ in $\sigma$, which are $r$-equal. Therefore, there exist renamings $\rho_{s,t}$ such that $k_s = k_t\rho_{s,t}$ for each $s, t \in [1, m]$.

By Lemma 5.4 for each CHR constraint $k$ which occurs in $P$ or in $G$, the set of sequences $s$ which are strictly increasing with respect to $Fv(k)$ (up to logical equivalence) is finite and has cardinality at the most $L$. Then there are two distinct nodes $n$ and $n'$ in $\sigma$ and there exist $s, t \in [1, m]$ such that $A(n) = k_s\#{l_s}^{i_s}$ and $A(n') = k_t\#{l_t}^{i_t}$ and $S_{F_\delta}(n) = S_{F_\delta}(n')\rho_{s,t}$. Then we have a contradiction, since by Lemma 5.5 this implies that there exists a terminating computation $\delta'$ for $G$ in $P$ s.t. either $F_{\delta'}$ is $m'$-repetitive with $m' < m$ or $F_{\delta'}$ is $m$-repetitive and $dg(F_{\delta'}) < dg(F_\delta)$ and then the thesis.

As an immediate corollary of the previous theorem, we have that the existence of a terminating computation for a goal $G$ in a CHR$_1^{\omega_a}(C)$ program $P$ is decidable. Then we have also the following result, which is stronger than Corollary 5.1 since here weak encodings are considered.

**Corollary 5.2** *There is no weak termination preserving encoding of Turing machines into CHR$_1^{\omega_a}(C)$.*

As mentioned at the beginning of this section, the previous result is obtained when considering the abstract semantics $\omega_o$. However it holds also when considering the theoretical semantics $\omega_t$. In fact Lemma 5.5 holds if we require that two r-equal constraints have the same sequence and have fired the same propagation rules. Since the propagation rules are finite Theorem 5.3 is still valid if $m \leq 2^r \cdot \frac{2^{w(u+w)(w+3)}-1}{2^{w(u+w)}-1}$ where $r$ is the number of propagation rules.

## 5.4 Summary and Related Works

We have shown two decidability results for two fragments of CHR$^{\omega_a}(C)$, the CHR language defined over a signature which does not allow function symbols. The first result, in Sect. 5.2, assumes the abstract operational semantics, while the second one, in Sect 5.3, holds for both semantics (abstract and theoretical). These results are not immediate. Indeed, CHR$^{\omega_a}(C)$, without further restrictions and with any of the two semantics, is a Turing complete language [116, 32]. It remains quite expressive also with our restrictions: for example, CHR$_1^{\omega_a}(C)$, the second fragment that we have considered, allows an infinite number of different states, hence, for example, it cannot be translated to Petri nets.

**Table 5.1** Summary of termination preserving encoding of turing machines

| Signature | Operational semantics | $k = 1$ | $k > 1$ |
|---|---|---|---|
| P (propositional) | $\omega_a$ | No | No |
| range-restricted C (constants) (cf. Sect. 5.2) | $\omega_a$ | No | **No** |
| C (constants), without $=$ | $\omega_a$ and $\omega_t$ | No | Yes |
| C (constants) (cf. Sect. 5.3) | $\omega_a$ and $\omega_t$ | **No** | Yes |
| F (functors) | $\omega_a$ and $\omega_t$ | Yes | Yes |

These results imply that range-restricted $\mathrm{CHR}^{\omega_a}(C)$ and $\mathrm{CHR}^{\omega_a}_1(C)$, the two considered fragments, are strictly less expressive than Turing machines (and therefore than $\mathrm{CHR}^{\omega_a}(C)$). Also, it seems that range-restricted $\mathrm{CHR}^{\omega_a}(C)$ is more expressive that $\mathrm{CHR}^{\omega_a}_1(C)$, since the decidability result for the second language is stronger. However, a direct result in this sense is left for future work.

Several papers have considered the expressive power of CHR in the last few years. In particular, [116] showed that a further restriction of $\mathrm{CHR}^{\omega_a}_1(C)$, which does not allow built-ins in the body of rules (and which therefore does not allow unification of terms) is not Turing complete. This result is obtained by translating $\mathrm{CHR}^{\omega_a}_1(C)$ programs (without unification) into propositional CHR and using the encoding of propositional CHR intro Petri nets provided in [13]. The translation to propositional CHR is not possible for the language (with unification) $\mathrm{CHR}^{\omega_a}_1(C)$ that we consider. [13] also provides a translation of range-restricted $\mathrm{CHR}^{\omega_a}(C)$ to Petri nets. However in this translation, differently from our case, it is also assumed that no unification built-in can be used in the rules, and only ground goals are considered. Related to this work is also [32], where it is shown that $\mathrm{CHR}^{\omega_a}(F)$ is Turing complete and that restricting to single-headed rules decreases the computational power of CHR. However, these results are based on the theory of language embedding, developed in the field of concurrency theory to compare Turing complete languages, hence they do not establish any decidability result. Another related study is [117], where the authors show that it is possible to implement any algorithm in CHR in an efficient way, i.e., with the best-known time and space complexity. Earlier works by Frühwirth [45, 46] studied the time complexity of simplification rules for naive implementations of CHR. In this approach, an upper bound on the derivation length, combined with a worst-case estimate of (the number and cost of) rule application attempts, allows to obtain an upper bound of the time complexity. The aim of all these works is clearly different from ours.

A summary of the existing results concerning the computational power of several dialects of CHR is shown in Table 5.1. In this table, "no" and "yes" refer to the existence of a termination preserving encoding of Turing machines into the considered language, while "any" means theoretical or abstract. The new results shown in this chapter are indicated in a bold font.

# Chapter 6
# Expressive Power of Priorities in CHR

The original theoretical operational semantics for CHR, denoted by $\omega_t$, is nondeterministic, as usual for many other rule based and concurrent languages. Such a nondeterminism has to be resolved in the implementations by choosing a suitable execution strategy. Most implementations indeed use the, so-called, refined operational semantics, called $\omega_r$, which has been formalized in [33] and fixes most of the execution strategy. This semantics, differently from the theoretical one, offers a good control over execution, however it is quite low level and lacks flexibility.

For this reason [75] proposed an extension of CHR, called CHR$^{\omega_p}$, for supporting an high level, explicit form of execution control which is more flexible and declarative than the one offered by the $\omega_r$ semantics. This is obtained by introducing explicitly in the syntax of the language rule annotations, which allow one to specify the priority of each rule. The operational semantics, in the following denoted by $\omega_p$, is changed accordingly: Rules with higher priority are chosen first. Priorities can be either static, when the annotations are completely defined at compile time, or dynamic, when the annotations contain variables which are instantiated at runtime.

Even though in [117] it is shown that any algorithm can be implemented in CHR preserving time and space complexity, yet in [75] it is claimed that "priorities do improve the expressivity of CHR."

In this chapter, we provide a formal ground for this informal claim by using a notion of expressivity coming from the field of concurrency theory to show several expressivity results relating CHR, CHR$^{\omega_p}$ and $static$ CHR$^{\omega_p}$. In fact, in this field the issue of the expressive power of a language has received a considerable attention in the past few years and several techniques and formalisms have been proposed for separating the expressive power of different languages which are Turing powerful (and therefore cannot be properly compared by using the standard tools of computability theory). Such a separation is meaningful both from a theoretical and a pragmatic point of view, since different (Turing complete) languages can provide quite different tools for implementing our algorithms. Indeed, some existing techniques for comparing the expressive power of two languages take into account the translation process, trying to formalize how difficult such a process is.

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_6, © Atlantis Press and the authors 2014

One of these techniques, that we use in this chapter, is based on the notion of language encoding, first formalized in [21, 113, 123][1] and can be described as follows. Intuitively, a language $\mathcal{L}$ is more expressive than a language $\mathcal{L}'$ or, equivalently, $\mathcal{L}'$ can be encoded in $\mathcal{L}$, if each program written in $\mathcal{L}'$ can be translated into an $\mathcal{L}$ program in such a way that: (1) the intended observable behavior of the original program is preserved, under some suitable decoding; (2) the translation process satisfies some additional restrictions which indicate how easy this process is and how reasonable the decoding of the observables is. For example, typically one requires that the translation is compositional w.r.t. (some of) the syntactic operators of the language (see for example [21]).

In this chapter, we use the notion of acceptable encoding, defined in the Sect. 6.1, which imposes the following requirements on the translation. First, similarly to the previous cases, we require that the translation of the goal (in the original program) and the decoding of the results (in the translated program) are homomorphic w.r.t. the conjunction of atoms. This assumption essentially means that our encoding and decoding functions respect the structure of the original goal and of the results (recall that for CHR programs these are constraints, that is, conjunction of atoms). Next we assume that the results to be preserved are the, so-called, qualified answers. Also this is a rather natural assumption, since these are the typical CHR observables for many CHR reference semantics.

To simplify the treatment, we assume that both the source and the target language use the same built-in constraints, semantically described by a theory $\mathcal{CT}$, which is not changed in the translation process. It is, on the other hand, worth noticing that we do not impose any restriction on the program translation.

Our first result presented in Sect. 6.2.1 shows that, in the presence of static priorities, allowing two or more atoms in the head of rules does not change the expressive power of the language. This result is obtained by providing, an acceptable encoding of *static* $\text{CHR}^{\omega_p}$ into *static* $\text{CHR}_2^{\omega_p}$, where the latter notation indicates the *static* $\text{CHR}^{\omega_p}$ language where at most two atoms are allowed in the heads of rules.

We also show that when considering a slightly different notion of answers, namely data sufficient answers, there exists an acceptable encoding from *static* $\text{CHR}^{\omega_p}$ to *static* $\text{CHR}_2^{\omega_p}$ even if we add also the requirement that the goal encoding and output decoding functions are the identity. It is worth noting that such a result does not hold for CHR without priorities, as shown in [32].

In Sect. 6.2.2 we prove that dynamic priorities do no augment the expressive power of the language w.r.t. static priorities. This result is obtained by providing an acceptable encoding of $\text{CHR}^{\omega_p}$ (with dynamic priorities) into *static* $\text{CHR}^{\omega_p}$.

Finally in Sect. 6.3 we prove a separation result showing that (static) priorities augment the expressive power of CHR, that is $\text{CHR}^{\omega_p}$ is strictly more expressive than CHR, in the sense that there exists no acceptable encoding of $\text{CHR}^{\omega_p}$ into CHR (with the $\omega_t$ semantics).

[1] The original terminology of these chapters was "language embedding."

## 6.1 Language Encoding

In this work, we consider the following languages and semantics:

- $\text{CHR}^{\omega_t}$: this is standard CHR, where the theoretical semantics is used,
- $\text{CHR}^{\omega_p}$: this is CHR with priorities, where both dynamic and static priorities can be used, the semantics is that one defined in the previous section ($\omega_p$);
- $static\ \text{CHR}^{\omega_p}$: this is CHR with static priorities only, with the $\omega_p$ semantics;
- $static\ \text{CHR}_2^{\omega_p}$: this is CHR with static priorities only, with the $\omega_p$ semantics, where we allow at most two constraints in the head of a rule.

Since all these languages are Turing powerful [117] in principle one can always encode a language into another one. The question is how difficult and how natural such an encoding is. As mentioned in the introduction, depending on the answer to this question one can discriminate different languages. Indeed, several approaches which compare the expressive power of concurrent languages impose the condition that the translation is compositional w.r.t. some operator of the language, because compositionality is considered a natural sign for the translation. Moreover, usually one wants that some observable properties of the computations are preserved by the translation, which is also a natural requirement.

In the following, we will then make similar assumptions on our encoding functions for CHR languages. We formally define a *program encoding* as any function $\mathcal{PROG} : \mathcal{P}_{\mathcal{L}} \to \mathcal{P}_{\mathcal{L}'}$ which translates a $\mathcal{L}$ program into a (finite) $\mathcal{L}'$ program ($\mathcal{P}_{\mathcal{L}}$ and $\mathcal{P}_{\mathcal{L}'}$ denote the set of $\mathcal{L}$ and $\mathcal{L}'$ programs, respectively). To simplify the treatment, we assume that both the source and the target language use the same built-in constraints semantically described by a theory $\mathcal{CT}$. Next we have to define how the initial goal and the observables should be translated by the encoding and the decoding functions, respectively. We require that these translations are compositional w.r.t. the conjunction of atoms. This assumption essentially means that the encoding and the decoding respect the structure of the original goal and of the observables. Moreover, since the source and the translated programs use the same constraint theory, it is natural to assume also that these two functions do not modify or add built-in constraints (in other words, we do not allow to simulate the behavior and the effects of the constraint theory).

We do not impose any restriction on the program translation, hence we have the following definition.

**Definition 6.1** (**Acceptable encoding**). Suppose that $\mathcal{C}$ is the class of all the possible multisets of constraints. An *acceptable encoding* (of $\mathcal{L}$ into $\mathcal{L}'$) is a tern of mappings $(\mathcal{PROG}, \mathcal{INP}, \mathcal{OUT})$ where $\mathcal{PROG} : \mathcal{P}_{\mathcal{L}} \to \mathcal{P}_{\mathcal{L}'}$ is the program encoding, $\mathcal{INP} : \mathcal{C} \to \mathcal{C}$ is the goal encoding, and $\mathcal{OUT} : \mathcal{C} \to \mathcal{C}$ is the output decoding which satisfy the following conditions:

1. the goal encoding function is compositional, that is, for any goal $(A, B) \in \mathcal{C}$, $\mathcal{INP}(A, B) = \mathcal{INP}(A), \mathcal{INP}(B)$ holds. We also assume that the built-ins present in the goal are left unchanged and no new built-ins can be added;

2. the output decoding function is compositional, that is, for any qualified answer $(A, B) \in \mathcal{C}$, $\mathcal{OUT}(A, B) = \mathcal{OUT}(A), \mathcal{OUT}(B)$ holds. We also assume that the built-ins present in the answer are left unchanged and no new built-ins can be added;
3. Qualified answers are preserved for the class $\mathcal{C}$, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in \mathcal{C}$, $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{PROG}(P)}(\mathcal{INP}(G)))$ holds.

Moreover we define an *acceptable encoding for data sufficient answers* of $\mathcal{L}$ into $\mathcal{L}'$ exactly as an acceptable encoding, with the exception that the third condition above is replaced by the following:

3′. Data sufficient answers are preserved for the class $\mathcal{C}$, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in \mathcal{C}$, $\mathcal{SA}_P(G)$ is equal to the data sufficient answers in $\mathcal{OUT}(\mathcal{QA}_{\mathcal{PROG}(P)}(\mathcal{INP}(G)))$.[2]

Further weakening these conditions and requiring for instance that the translation of $A, B$ is some form of composition of the translation of $A$ and $B$ does not seem reasonable, as conjunction is the only form for goal composition available in CHR.

Note that, according to the previous definition, if there exists an acceptable encoding then there exists also an acceptable encoding for data sufficient answers. This is an immediate consequence of the fact that data sufficient answers are a subset of data qualified answers.

In the following, given a program $P$, we denote by $\mathrm{Pred}(P)$ and $\mathrm{Head}(P)$ the set of all the predicate symbols $p$ s.t. $p$ occurs in $P$ and in the head of a rule in $P$, respectively.

## 6.2 Positive Results

In this section, we will present some (acceptable) encodings for the four languages described at the beginning of Sect. 6.1. We first present some immediate results which derive directly from the language definitions. Then we will describe two of the main results of this paper, namely that there exists an acceptable encoding from $static\ \mathrm{CHR}^{\omega_p}$ to $static\ \mathrm{CHR}_2^{\omega_p}$ and from $\mathrm{CHR}^{\omega_p}$ to $static\ \mathrm{CHR}^{\omega_p}$. The combination of these results shows that $static\ \mathrm{CHR}_2^{\omega_p}$ is as powerful as the full $\mathrm{CHR}^{\omega_p}$, that is, a program with dynamic priorities can be (acceptably) encoded into one with static priorities and this, in its turn, can be encoded into a program which does not use more than two constraints in the head of rules.

We first observe that $\mathrm{CHR}^{\omega_t}$ is a sublanguage of $static\ \mathrm{CHR}^{\omega_p}$, since a $\mathrm{CHR}^{\omega_t}$ program can be seen as a $static\ \mathrm{CHR}^{\omega_p}$ program where all the rules have equal priority. Clearly $static\ \mathrm{CHR}_2^{\omega_p}$ is a sublanguage of $static\ \mathrm{CHR}^{\omega_p}$ that, in its turn, is a sublanguage of $\mathrm{CHR}^{\omega_p}$. Moreover, when a language $\mathcal{L}$ is a sublanguage of $\mathcal{L}'$

[2] Note that in 3. and in 3′. the function $\mathcal{OUT}()$ is extended in the obvious way to sets of qualified answers.

then a tern of identity functions provides an acceptable encoding between the two languages. Therefore, we have the following.

**Fact 1** *There exists an acceptable encoding from CHR$^{\omega_t}$ to static CHR$^{\omega_p}$, from static CHR$_2^{\omega_p}$ to static CHR$^{\omega_p}$, and from static CHR$^{\omega_p}$ to CHR$^{\omega_p}$.*

As previously mentioned, the existence of an acceptable encoding implies the existence of an acceptable encoding for data sufficient answers. Hence, we have the following immediate corollary.

**Corollary 6.1.** *There exists an acceptable encoding for data sufficient answers from CHR$^{\omega_t}$ to static CHR$^{\omega_p}$, from static CHR$_2^{\omega_p}$ to static CHR$^{\omega_p}$, and from static CHR$^{\omega_p}$ to CHR$^{\omega_p}$.*

### 6.2.1 Encoding Static CHR$^{\omega_p}$ into Static CHR$_2^{\omega_p}$

In this section, we will provide an acceptable encoding from *static* CHR$^{\omega_p}$ to *static* CHR$_2^{\omega_p}$. We assume that $P$ is a *static* CHR$^{\omega_p}$ program composed by $m$ rules and that the $i$-th rule (with $i \in \{1, \ldots, m\}$) has the form:

$$p_i :: rule_i \ @ \ h_{(i,1)}(\bar{t}_1), \ldots, h_{(i,l_i)}(\bar{t}_{l_i}) \backslash h_{(i,l_i+1)}(\bar{t}_{l_i+1}), \ldots, h_{(i,r_i)}(\bar{t}_{r_i}) \Leftrightarrow G_i | C_i.$$

Moreover, we denote by $p_{max}$ the lowest priority (i.e., the biggest $p_i$).

First, we require that the goal encoding (the second component of our acceptable encoding) is a nonsurjective function. The reason for this requirement is that the program encoding (first component of the triple) needs to use, in the translated program, some fresh constraints which do not appear in the initial (translated) goal. A simple goal encoding that satisfies this requirement is the one that does not change built-in constraints and adds a letter, say "a," at the beginning of the other constraints, as shown below:

$$\mathcal{INP}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

In the following of this section, by a slight abuse of notation, we use the notation $\mathcal{INP}()$ also to indicate a function from predicate symbols to predicate symbols.

In order to define the program encoding, we need the following constraints:

- $id(t)$; This will be used to simulate an Introduce transition step; $t$ is a term that will be used as a constraint identifier.
- $end$; It will be used to delete the constraint added in the process of simulating the firing of the rules.
- $rC[N]_i(\bar{t})$ with $N \in \{1, \ldots, r_i\}$ where $r_i$ is the number of constraint in the head of the $i$-th rule; this will be used to check if the rule $rule_i$ can fire.

- $rA_i(\bar{t})$ is a constraint which is added to the store when the $i$-th rule is fired; The $\bar{t}$ are the identifiers of the constraints which are consumed by the application of the $i$-th rule and therefore should be removed from the store.
- $new_k(V, \bar{u})$ where $V$ is a term, $\bar{u}$ is a sequence of terms and $k$ is a predicate symbol in $\mathcal{INP}(Pred(P))$; This new constraint will be used to add to a constraint $k(\bar{u})$ a new identifier $V$.

Note that since no constraint in this list starts with an "a," previous assumption on the goal encoding function $\mathcal{INP}(\,)$ implies that these constraints cannot be in any goal produced by $\mathcal{INP}(\,)$.

In the following, to simplify the notation when we are not interested in the arguments of a predicate we will simply use an underscore to indicate them (thus writing, for example, $p(\_), q(\_)$).

We can now define the program encoding function, denoted by $\alpha(\,)$. This function, given a $static$ CHR$^{\omega_p}$ program $P$, returns the program constructed as follows:

$$\begin{gathered}\text{for every predicate name } k \in \mathcal{INP}(Head(P))\\ 1 :: rule_{(1,k)} \ @\ id(V),\ p(\bar{X}) \Leftrightarrow id(V+1), new_k(V, \bar{X})\\ 2 :: rule_{(2,k)} \ @\ k(\bar{X}) \Leftrightarrow id(2), new_k(1, \bar{X})\end{gathered}$$

$$\begin{gathered}\text{for every } i \in \{1, \ldots, m\},\ N \in [1, r_i - 1]\\ 3 :: rule_{(3,i,N)} \ @\ end \backslash rC[N]_i(\_) \Leftrightarrow true\end{gathered}$$

$$\begin{gathered}\text{for every predicate name } k \in \mathcal{INP}(Head(P)),\ i \in \{1, \ldots, m\}\\ 3 :: rule_{(4,i,k)} \ @\ rA_i(\bar{V}) \backslash new_k(V', \bar{X}) \Leftrightarrow V' \in \bar{V} | true\end{gathered}$$

$$\begin{gathered}\text{for every } i, j \in \{1, \ldots, m\},\ N \in [1, r_i - 1]\\ 3 :: rule_{(5,j,i,N)} \ @\ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow \bar{V} \cap \bar{V}' \neq \emptyset | true\end{gathered}$$

$$\begin{gathered}\text{for every } i \in \{1, \ldots, m\}\\ 4 :: rule_{(6,i)} \ @\ rA_i(\_) \Leftrightarrow true\end{gathered}$$

$$\text{for every } i \in \{1, \ldots, m\}\, CHECK\_RULE(i)$$

$$\begin{gathered}\text{for every } i \in \{1, \ldots, m\}\\ 6 + p_i :: rule_{(7,i)} \ @\ rC[r_i]_i(V_1, \ldots, V_{r_i}, \bar{t}_1, \ldots, \bar{t}_{r_i}), id(V) \Leftrightarrow\\ G_i | Update(\mathcal{INP}(C_i), V), rA_i(V_{l_i+1}, \ldots, V_{r_i})\end{gathered}$$

$$\begin{gathered}7 + p_{max} :: rule_8 \ @\ id(\_) \Leftrightarrow end\\ 7 + p_{max} :: rule_9 \ @\ end \Leftrightarrow true\end{gathered}$$

where $CHECK\_RULE(i)$ are the following rules

$$\begin{gathered}\text{for every } N \in [2, r_i]\\ 5 :: rule'_{(i,N)} \ @\ rC[N-1]_i(\bar{V}_1, \bar{X}_1), new_{\mathcal{INP}(h_{(i,N)})}(V_2, \bar{X}_2) \Rightarrow\\ V_2 \notin \bar{V}_1 | rC[N]_i(\bar{V}_1, V_2, \bar{X}_1, \bar{X}_2)\end{gathered}$$

where by convention, $rC[1]_i(V, \bar{X}) = new_{\mathcal{INP}(h_{(i,1)})}(V, \bar{X})$ and $Update(C, V)$ is defined as follows

$$Update(k(\bar{t}), V) = new_k(V, \bar{t})$$
if $k(\bar{t})$ is a CHR constraint

$$Update(c(\bar{t}), V) = c(\bar{t})$$
if $c(\bar{t})$ is a built-in constraint

$$Update([\,], V) = id(V)$$

$$Update([d(\bar{X}) \mid Ds], V) = Update(d(\bar{X}), V), Update(Ds, V + 1).$$

*Example 6.1.* As an example for the application of the program encoding $\alpha(\,)$ let us consider the simple program $P$ composed by the following rule:

$$1 :: h_1(X), h_2(Y) \backslash h'(Z) \Leftrightarrow X = Y | h$$

$\alpha(P)$ is the following program:

$$1 :: id(V), ah_1(X) \Leftrightarrow id(V + 1), new_{ah_1}(V, X)$$
$$1 :: id(V), ah_2(X) \Leftrightarrow id(V + 1), new_{ah_2}(V, X)$$
$$1 :: id(V), ah'(X) \Leftrightarrow id(V + 1), new_{ah'}(V, X)$$

$$2 :: ah_1(X) \Leftrightarrow id(2), new_{ah_1}(1, X)$$
$$2 :: ah_2(X) \Leftrightarrow id(2), new_{ah_2}(1, X)$$
$$2 :: ah'(X) \Leftrightarrow id(2), new_{ah'}(1, X)$$

$$3 :: end \backslash rC2_1(V_1, V_2, X_1, X_2) \Leftrightarrow true$$
$$3 :: end \backslash rC3_1(V_1, V_2, V_3, X_1, X_2, X_3) \Leftrightarrow true$$

$$3 :: rA_1(V) \backslash new_{ah_1}(V', X) \Leftrightarrow V' = V | true$$
$$3 :: rA_1(V) \backslash new_{ah_2}(V', X) \Leftrightarrow V' = V | true$$
$$3 :: rA_1(V) \backslash new_{ah'}(V', X) \Leftrightarrow V' = V | true$$

$$3 :: rA_1(V) \backslash rC2_1(V_1, V_2, X_1, X_2) \Leftrightarrow V \notin \{V_1, V_2\} | true$$
$$3 :: rA_1(V) \backslash rC3_1(V_1, V_2, V_3, X_1, X_2, X_3) \Leftrightarrow V \notin \{V_1, V_2, V_3\} | true$$

$$4 :: rA_1(V) \Leftrightarrow true$$

$$5 :: new_{ah_1}(V_1, X_1), new_{ah_2}(V_2, X_2) \Rightarrow V_2 \neq V_1 | rC2_1(V_1, V_2, X_1, X_2)$$
$$5 :: rC2_1(V_1, V_2, X_1, X_2), new_{ah'}(V_3, X_3) \Rightarrow V_3 \notin \{V_1, V_2\} | rC3_1(V_1, V_2, V_3, X_1, X_2, X_3)$$

$$7 :: rC3_1(V_1, V_2, V_3, X_1, X_2, X_3), id(V) \Leftrightarrow X_1 = X_2 | new_{ah}(V), id(V + 1), rA_1(V_3)$$

$$8 :: id(V) \Leftrightarrow end$$
$$8 :: end \Leftrightarrow true$$

We now provide a brief explanation of the program encoding $\alpha(\,)$. Intuitively the encoding simulates the execution of the original program. The constraint identifier introduced by Introduce transition in the original program is simulated by adding a unique term as an argument to a $new_k(\_)$ constraint.

The simulation process can be divided into the following three phases:

1. **Initialization**. In the initialization phase, for each $k \in \mathcal{INP}(Head(P))$ we introduce two (sets of) rules replacing a constraint $k(\bar{t})$ with $new_k(n, \bar{t})$. Moreover we use an $id$ predicate symbol to memorize the highest identifier used. The first rule to be fired is a rule $rule_{(2,k)}$ that triggers the firing of rules $rule_{(1,k)}$. Note that rules $rule_{(1,k)}$ have maximal priority and therefore if a constraint of the form $id(t)$ occurs in the CHR store they are always tried before rules $rule_{(2,k)}$.
2. **Main**. The main phase is divided into three subphases. The first subphase is the *evaluation* that starts when the init phase terminates (at this point all the constraints $k(\bar{t})$, with $k \in \mathcal{INP}(Head(P))$ have been converted into $new_k(l, \bar{t})$). Rules $rule'_{(i,N)}$ determine what rules belonging to the original program can fire. The second subphase is the *activation*. During this subphase if $rule_i$ can be fired in the original program $P$ then $rule_{(7,i)}$ can be fired in the program $\alpha(P)$. If the original program has not reached the final state then one of the rules $rule_{(7,i)}$ fires starting the *deletion* subphase. In this last subphase rules $rule_{(4,i,k)}$, $rule_{(5,j,i,N)}$ and $rule_{(6,i)}$ delete all the constraints that are used to simulate the constraints deleted by the application of the $i$-th rule in the original program $P$.
3. **Termination**. The termination phase is triggered by rule $rule_8$ that is used to detect when no rule $rule_{(7,i)}$ can fire (this happens iff the original program has reached a final state). Rules $rule_{(3,i,N)}$ and $rule_9$ delete all the constraints produced during the computation for the simulation purpose, that is $id$, $rC[N]_i$ and $end$.

It is worth noting that in the program encoding presented we implicitly assumed that the constraint theory $\mathcal{CT}$ has equalities and inequalities constraints (i.e. we can evaluate whether $n = n'$ and $n \neq n'$ where $n, n' \in \mathbb{N}$). All the operators $\in$, $\notin$ and $\cap$ written in the guards can be replaced by equalities and inequalities. In rules $rule'_{(i,N)}$, for instance, the guards $V' \notin \bar{V}$, where $\bar{V} = V_1, \ldots, V_n$ can be replaced by $V' \neq V_1, \ldots, V' \neq V_n$. Rules

$$3 :: rule_{(4,i,k)} \ @ \ rA_i(\bar{V}) \backslash new_k(V', \bar{X}) \Leftrightarrow V' \in \bar{V} | true$$

where $\bar{V} = V_{l_{i+1}}, \ldots, V_{r_i}$ can be rewritten by the set of rules

$$\{3 :: rule_{(4,i,k,o)} \ @ \ rA_i(\bar{V}) \backslash new_k(V', \bar{X}) \Leftrightarrow V' = V_o | true| \ \ | \ o \in [l_{i+1}, r_i]\}$$

and finally rules

$$3 :: rule_{(5,j,i,N)} \ @ \ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow \bar{V} \cap \bar{V}' \neq \emptyset | true$$

where $\bar{V} = V_{l_{i+1}}, \ldots, V_{r_i}$ and $\bar{V}' = V'_1, \ldots, V'_N$ can be rewritten by the set of rules

$$\{3 :: rule_{(5,j,i,N,o,p)} \ @ \ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow V_o = V'_p | true \ \mid \\ o \in [l_{i+1}, r_i] \text{ and } p \in [1, N] \ \}.$$

However, strictly speaking, the assumption of having equalities and inequalities is not needed (we used them for the sake of simplicity). In fact, only rules $rule'_{(i,N)}$ can be translated into rules having inequalities in their guards. These rules have a structure similar to the following rule:

$$p :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V \neq V_1, \ldots, V \neq V_n | c_3(V, V_1, \ldots, V_n, \bar{X}, \bar{Y})$$

Since we know that $V, V_1, \ldots, V_n$ will always be matched with ground terms we can replace the previous rule with the following rules:

$$\begin{gathered} p_1 :: eq(\bar{Z}) \backslash c_3(\bar{Z}) \Leftrightarrow true \\ p_2 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V = V_1 | eq(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \\ \ldots \\ p_2 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V = V_n | eq(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \\ p_3 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow c_3(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \end{gathered}$$

where $p_1$, $p_2$, $p_3$ are priorities s.t. $p_1 < p_2 < p_3$ and $eq$ is a new constraint. Thus, we have removed inequalities. Equalities instead can be removed by simply changing the name of terms in the head of the rules. For instance, the equality $X = Y$ in a rule like

$$k_1(X), k_2(Y) \Leftrightarrow X = Y | C$$

can be removed replacing the previous rule with the following one

$$k_1(X), k_2(X) \Leftrightarrow C.$$

To conclude the definition of the acceptable encoding, we need the last ingredient: the output decoding function. If we run the goal $\mathcal{INP}(G)$ in the program $\alpha(P)$ we obtain the same qualified answers obtained by running $G$ in the program $P$, with the only difference that if in the qualified answer of $P$ there is a CHR constraint $k(\bar{t})$ then in the corresponding qualified answer of the encoded program $\alpha(P)$ there will be either a constraint $new_{ak}(V, \bar{t})$ (if $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition step) or a constraint $ak(\bar{t})$ (if $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal $G$).

Therefore, the decoding function that we need is:

$$\mathcal{OUT}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = new_{ak}(V, \bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = ak(\bar{t}). \end{cases}$$

The following Theorem, whose proof is in the Appendix, shows that the triple $(\alpha(), \mathcal{INP}(), \mathcal{OUT}())$ that we have defined indeed provides the desired encoding.

**Theorem 6.1.** *The triple* $(\alpha(), \mathcal{INP}(), \mathcal{OUT}())$ *provides an acceptable encoding from* $static\ CHR^{\omega_p}$ *into* $static\ CHR_2^{\omega_p}$.

Note that, as mentioned after Definition 6.1, if there exists an acceptable encoding then there exists also an acceptable encoding for data sufficient answers. Hence, previous result implies that there is an acceptable encoding for data sufficient answers from $static$ CHR$^{\omega_p}$ into $static$ CHR$_2^{\omega_p}$.

Moreover, when considering data sufficient answers it is possible to strengthen previous Theorem by requiring that the goal encoding and the output decoding functions are the identity functions. Intuitively this does not hold if we consider the program encoding $\alpha(\ )$ presented in the previous session because, when the goal encoding function is the identity function, constraint such as $id$, $end$, $rC[N]_i$ could be in the initial goal of the encoded program. However, when we are focusing on data sufficient answers we can overcome this problem and use the same program encoding as a base for a new program encoding for data sufficient answers. We can indeed exploit the fact that when a fresh constraint for a program $P$ is in a goal then the program has no data sufficient answers for that goal.

Below we exploit this idea and we will first define a program translation $\beta(P, q)$ that, given a $static$ CHR$^{\omega_p}$ program $P$ and a predicate symbol $q$ produces a modified program $P'$ which has the same data sufficient answers of $P$ for every goal that does not contain the predicate symbol $q$, produces a failure otherwise.[3]

Let us then consider a $static$ CHR$^{\omega_p}$ program $P$ composed by $m$ rules

$$p_i :: rule_i \ @\ H_i \backslash H'_i \Leftrightarrow G_i | B_i$$

where $1 \leq p_i \leq p_{max}$. Without loss of generality, we can assume that $start$ and $init$ are not contained in $Head(P)$. Moreover, let $f$ be a function that maps predicate symbols into predicate symbols which are not in $Pred(P) \cup \{start, init, q\})$. $f$ can be extended to multiset of constraints in the obvious way.

The transformation $\beta(P, q)$ produces the following program

$$1 :: rule_{m+1} \ @\ start, q(\_) \Leftrightarrow false$$

for every predicate name $k \in Head(P)$

$$1 :: rule_{(m+2,k)} \ @\ start, f(k(\_)) \Leftrightarrow false$$

$$1 :: rule_{m+3} \ @\ start, init \Leftrightarrow false$$

[3] Note that we are not requiring that the presence of a constraint of the form $q(\bar{t})$ always brings to a failure. We allow for instance the use of $q(\bar{t})$ during the execution. The program fails only if a constraint of the form $q(\bar{t})$ is in the original goal.

$$2 :: rule_{m+4} \ @ \ start \Leftrightarrow init$$

$$\text{for every predicate name } k, k' \in Head(P)$$
$$3 :: rule_{(m+5,k)} \ @ \ k(\_) \Rightarrow start$$
$$3 :: rule_{(m+6,k,k')} \ @ \ k(\_) \backslash k'(\bar{Y}) \Leftrightarrow f(k'(\bar{Y}))$$

$$\text{for every predicate name } k \in Head(P)$$
$$4 :: rule_{(m+7,k)} \ @ \ k(X) \Rightarrow f(k(\bar{X}))$$

$$\text{for every } i \in \{1, \ldots, m\}$$
$$4 + p_i :: rule'_i \ @ \ f(H_i) \backslash f(H'_i), \Leftrightarrow G_i | f(B_i), init$$

$$5 + p_{max} :: rule_{(m+8)} \ @ \ init, init \Leftrightarrow init$$

$$\text{for every predicate name } k \in Head(P)$$
$$6 + p_{max} :: rule_{(m+9,k)} \ @ \ k(\_), init \Leftrightarrow true$$

The following lemma, whose proof is in the appendix, shows that indeed the transformed program has the behavior that we have described before.

**Lemma 6.1.** *Let $P$ be a static CHR$^{\omega_p}$ program and let $q$ be a predicate symbol. For every goal $G$, if $G$ does not contain the predicate symbol $q$ then $\mathcal{SA}_P(G) = \mathcal{SA}_{\beta(P,q)}(G)$, $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$ otherwise.*

Now let us denote with $\beta'()$ the extensions of $\beta()$ to a list of predicate symbols ($\beta'(P, []) = P$ and $\beta'(P, [X|XS]) = \beta(\beta'(P, XS), X)$).

Suppose that $New\_Symbols(P)$ is the list of the new predicate symbols introduced by $\alpha(P)$ (namely $id, end, rC[N]_i, rA_i, new_{ak}$) and w.l.o.g suppose that these predicate symbols are fresh in $P$.

Using the Lemma we can prove the following result previously described.

**Theorem 6.2.** *The triple $(\beta'(\alpha(P), New\_Symbols(P)), id, id)$, where $id$ is the identity function and $\alpha()$ is defined as before, provides an acceptable encoding for data sufficient answers from static CHR$^{\omega_p}$ into static CHR$_2^{\omega_p}$.*

*Proof.* The proof derives by Lemma 6.1 using the program encoding of Theorem 6.1. Indeed given a program $P$ w.l.o.g. we can assume that $id$, $end$, $rC[N]_i$, $rA_i$ and $new_{ak}$ (with $k \in Head(P)$) are not contained in $Head(P)$. Therefore for every goal $G$ containing at least one of them, we have that

$$\mathcal{SA}_P(G) = \emptyset.$$

By using the same arguments of Theorem 6.1, for each goal $G$ s.t. no predicate symbol in $New\_Symbols(P)$ is in $G$ we have that $\mathcal{SA}_P(G) = \mathcal{SA}_{\alpha(P)}(G)$. Moreover, by construction, $\alpha(P) \in static$ CHR$_2^{\omega_p}$. By Lemma 6.1 for every goal $G$, if $G$ does not contain the predicate symbols in $New\_Symbols(P)$ then $\mathcal{SA}_P(G) = \mathcal{SA}_{\beta'(P,New\_Symbols(P))}(G)$, $\mathcal{SA}_{\beta'(P,New\_Symbols(P))}(G) = \emptyset$ otherwise. Therefore we have that for each goal $G$, $\mathcal{SA}_{\beta'(\alpha(P),New\_Symbols(P))}(G) =$

$\mathcal{SA}_P(G)$. Moreover, since $\alpha(P) \in static\ \mathrm{CHR}_2^{\omega_p}$, by definition of $\beta'()$ we have that $\beta'(\alpha(P), New\_Symbols(P)) \in static\ \mathrm{CHR}_2^{\omega_p}$ and then the thesis.

It is worth noting that Theorem 6.2 does not hold when the traditional semantics is considered, as shown in [31].

### 6.2.2 Encoding CHR$^{\omega_p}$ into Static CHR$^{\omega_p}$

In this section, we prove that the CHR$^{\omega_p}$ language, which allows dynamic priorities, is not more expressive than $static$ CHR$^{\omega_p}$, which allows static priorities only. This result is obtained by providing an (acceptable) encoding of CHR$^{\omega_p}$ into $static$ CHR$^{\omega_p}$.

As usual, we assume that $P$ is a CHR$^{\omega_p}$ program composed by $m$ rules and we also assume that the $i$-th rule (with $i \in \{1, \ldots, m\}$) has the form:

$$p_i :: rule_i \ @\ H_i \backslash H_i' \Leftrightarrow G_i | B_i$$

Moreover, given a multiset of CHR constraints $\bar{H} = h_1(\bar{t}_1), \ldots, h_n(\bar{t}_n)$ and a sequence of (distinct) variables $\bar{V} = V_1, \ldots, V_n$, we denote by $new'(\bar{H}, \bar{V})$ the multiset of atoms $new_{h_1}(V_1, \bar{t}_1), \ldots, new_{h_n}(V_n, \bar{t}_n)$.

As for the goal encoding and the output decoding functions we use here the same functions $\mathcal{INP}()$ and $\mathcal{OUT}()$ defined in Sect. 6.2.1.[4] The program encoding $\mathcal{T}(P)$ from CHR$^{\omega_p}$ into $static$ CHR$^{\omega_p}$ is instead defined as the function that, given a program $P$, produces the following program:

$$\begin{gathered}
\text{for every predicate name } ak \in \mathcal{INP}(Head(P)) \\
1 :: rule_{(1,k)} \ @\ start \backslash id(V), ak(\bar{X}) \Leftrightarrow id(V+1), new_{ak}(V, \bar{X}) \\
2 :: rule_{(2,k)} \ @\ ak(\bar{X}) \Rightarrow start, id(0)
\end{gathered}$$

$$2 :: rule_3 \ @\ start \Leftrightarrow highest\_priority(inf)$$

$$\begin{gathered}
\text{for every } i \in \{1, \ldots, m\} \\
3 :: rule_{(4,i)} \ @\ end \backslash instance_i(_) \Leftrightarrow true
\end{gathered}$$

$$4 :: rule_5 \ @\ end \Leftrightarrow true$$

$$\text{for every } i \in \{1, \ldots, m\} \quad \texttt{EVALUATE\_PRIORITIES}(i)$$

$$7 :: rule_9 \ @\ highest\_priority(inf), id(V) \Leftrightarrow end$$

$$\text{for every } i \in \{1, \ldots, m\} \quad \texttt{ACTIVATE\_RULE}(i)$$

[4] In the following of this section, by an abuse of notation, we use the function $\mathcal{INP}()$ also as a function from predicate symbols to predicate symbols.

If $rule_i$ is not a propagation rule then EVALUATE_PRIORITIES($i$) are the following rules

$$6 :: rule_{(7,i)} \ @ \ new'(\mathcal{INP}(H_i), \bar{Z}), new'(\mathcal{INP}(H'_i), \bar{U}) \backslash highest\_priority(inf) \Leftrightarrow G_i | highest\_priority(p_i)$$

$$6 :: rule_{(8,i)} \ @ \ new'(\mathcal{INP}(H_i), \bar{Z}), new'(\mathcal{INP}(H'_i), \bar{U}) \backslash highest\_priority(P) \Leftrightarrow G_i, p_i < P | highest\_priority(p_i)$$

if $rule_i$ is a propagation rule then EVALUATE_PRIORITIES($i$) are the following rules

$$5 :: rule_{(6,i)} \ @ new'(\mathcal{INP}(H_i), \bar{Z}) \Rightarrow G_i | instance_i(\bar{Z})$$

$$6 :: rule_{(7,i)} \ @ \ instance_i(\bar{Z}), new'(\mathcal{INP}(H_i), \bar{Z}) \backslash highest\_priority(inf) \Leftrightarrow G_i | highest\_priority(p_i)$$
$$6 :: rule_{(8,i)} \ @ \ instance_i(\bar{Z}), new'(\mathcal{INP}(H_i), \bar{Z}) \backslash highest\_priority(P) \Leftrightarrow G_i, p_i < P | highest\_priority(p_i)$$

if $rule_i$ is a propagation rule then ACTIVATE_RULE($i$) is the following rule

$$8 :: rule_{(10,i)} \ @ \ new'(\mathcal{INP}(H_i), \bar{Z}) \backslash instance_i(\bar{Z}), highest\_priority(P), id(V) \Leftrightarrow G_i, p_i = P | Update(\mathcal{INP}(B_i), V), highest\_priority(inf)$$

if $rule_i$ is not a propagation rule then ACTIVATE_RULE($i$) is the following rule

$$8 :: rule_{(10,i)} \ @ \ new'(\mathcal{INP}(H_i), \bar{Z}), new'(\mathcal{INP}(H'_i), \bar{U}), highest\_priority(P), id(V) \Leftrightarrow G_i, p_i = P | Update(\mathcal{INP}(B_i), V), highest\_priority(inf)$$

In the above encoding we assume that the constraint theory $\mathcal{CT}$ allows to use equalities and inequalities (so we can evaluate whether $p_i = h$ and $p_i > h$ where $h \in \mathbb{Z}$ and $p_i$ is an arithmetic expression). We also assume $inf$ is a conventional constant which is bigger than all $p_i$ (i.e., it represents the lowest priority). The $Update$ function is exactly the one defined in Sect. 6.2.1.

*Example 6.2.* Let us consider as $P$ the shortest path program depicted in Fig. 4.2. The correspondent $\mathcal{T}(P)$ is the following program:

$$1 :: start \backslash id(V), asource(\bar{X}) \Leftrightarrow id(V+1), new_{asource}(V, \bar{X})$$
$$1 :: start \backslash id(V), adist(\bar{X}) \Leftrightarrow id(V+1), new_{adist}(V, \bar{X})$$
$$1 :: start \backslash id(V), aedge(\bar{X}) \Leftrightarrow id(V+1), new_{aedge}(V, \bar{X})$$

$$2 :: asource(\bar{X}) \Rightarrow start, id(0)$$
$$2 :: adist(\bar{X}) \Rightarrow start, id(0)$$

$$2 :: aedge(\bar{X}) \Rightarrow start, id(0)$$

$$2 :: start \Leftrightarrow highest\_priority(inf)$$

$$3 :: end \backslash instance_1(\bar{Z}) \Leftrightarrow true$$
$$3 :: end \backslash instance_2(\bar{Z}) \Leftrightarrow true$$
$$3 :: end \backslash instance_3(\bar{Z}) \Leftrightarrow true$$

$$4 :: end \Leftrightarrow true$$

$$5 :: new_{asource}(V, X) \Rightarrow instance_1(V)$$
$$6 :: new_{asource}(V, X) \backslash highest\_priority(inf) \Leftrightarrow highest\_priority(1)$$
$$6 :: new_{asource}(V, X) \backslash highest\_priority(P) \Leftrightarrow 1 < P | highest\_priority(1)$$

$$6 :: new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(inf) \Leftrightarrow$$
$$X_2 \leq Y_2 | highest\_priority(1)$$
$$6 :: new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(P) \Leftrightarrow$$
$$X_2 \leq Y_2, 1 < P | highest\_priority(1)$$

$$5 :: new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \Rightarrow$$
$$instance_3(V_1, V_2)$$
$$6 :: new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \backslash highest\_priority(inf) \Leftrightarrow$$
$$highest\_priority(X_2 + 2)$$
$$6 :: new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \backslash highest\_priority(P) \Leftrightarrow$$
$$X_2 + 2 < P | highest\_priority(X_2 + 2)$$

$$7 :: highest\_priority(inf), id(V) \Leftrightarrow end$$

$$8 :: new_{asource}(V, X) \backslash instance_1(\bar{V}), highest\_priority(P), id(V') \Leftrightarrow$$
$$1 = P | new_{adist}(V', X, 0), id(V' + 1), highest\_priority(inf)$$
$$8 :: new_{adist}(V_1, X, X_1) \backslash new_{adist}(V_2, X, X_2), highest\_priority(P), id(V') \Leftrightarrow$$
$$X_1 \leq X_2, 1 = P | id(V'), highest\_priority(inf)$$
$$8 :: new_{adist}(V_1, X, X_1), new_{aedge}(V_2, X, X_2, X_3) \backslash instance_3(V_1, V_2), highest\_priority(P),$$
$$id(V') \Leftrightarrow X_1 + 2 = P | new_{adist}(X_3, X_1 + X_2), id(V' + 1), highest\_priority(inf)$$

We now provide some explanations for the above encoding. Intuitively the result of the encoding can be divided into three phases:

1. **Init.** In the init phase, for each (user defined) predicate symbol $ak \in \mathcal{INP}(Head(P))$ we introduce a rule $rule_{(1,k)}$, which replaces $ak(\bar{t})$ by $new_{ak}(V, \bar{t})$ where $V$ is a variable which will be used to simulate the identifier used in identified constraints. Moreover, we use the $id$ predicate symbol to memorize the highest identifier used. Rules $rule_{(2,k)}$ (one for each predicate symbol $ak \in \mathcal{INP}(Head(P))$, as before) are used to fire rules $rule_{(1,k)}$ and also to start the following phase (via $rule_3$). Note that rules $rule_{(1,k)}$ have maximal priority and therefore are tried before rules $rule_{(2,k)}$.
2. **Main.** The main phase is divided into two phases: the *evaluation* phase starts when the init phase adds the constraint $highest\_priority(inf)$. Rules $rule_{(6,i)}$, $\ldots$, $rule_{(8,i)}$ store in $highest\_priority$ the highest priority on all the rule instances that can be fired. After the end of the evaluation phase, the *activation*

starts. During this phase if a rule can be fired one of the rules $rule_{(10,i)}$ is fired. After the rule has been fired, the constraint $highest\_priority(inf)$ is produced which starts a new evaluation phase.

3. **Termination.** The termination phase is triggered by rule $rule_9$. This rule fires when no instance from the original program can fire. During the termination phase all the constraints produced during the computation (namely $id$, $instance_i$, $highest\_priority$, $end$) are deleted.

In the following we now provide some more details on the two crucial points in this translation: the evaluation and the activation phases.

- **Evaluation.** The rules in the set denoted by

$$\texttt{EVALUATE\_PRIORITIES}(i)$$

are triggered by the insertion of $highest\_priority(inf)$ in the constraint store. In the case of a propagation rule $rule_i \in P$, the rules in

$$\texttt{EVALUATE\_PRIORITIES}(i)$$

should consider the possibility that there is an instance of $rule_i$ that cannot be fired because it has been previously fired. When an instance of a propagation rule can fire, rule $rule_{(6,i)}$ adds a constraint $instance_i(\bar{v})$, where $\bar{v}$ are the identifiers of the CHR atoms which can be used to fire $rule_i$. The absence of the constraint $instance_i(\bar{v})$ in the constraint store means that either $rule_i$ cannot be fired by using the CHR atoms identified by $\bar{v}$ or has already fired for the CHR atoms identified by $\bar{v}$.

The evaluation of the priority for a simpagation or a simplification rule is instead more simple because the propagation history does not affect the execution of these two types of rules.

Rules $rule_{(7,i)}$ and $rule_{(8,i)}$ replace the constraint $highest\_priority(p)$ with the constraint $highest\_priority(p')$ if a rule of priority $p'$ can be fired and $p > p'$.

- **Activation.** When the evaluation phase ends if a rule can fire then one of the rules $rule_{(10,i)}$ is fired since $highest\_priority(inf)$ has been removed from the constraint store.

The only difference between a propagation rule and a simpagation/simplification rule is that when a propagation rule is fired the corresponding constraint $instance_i(\bar{v})$ is deleted to avoid the execution of the same propagation rule in the future.

It is worth noting that the nondeterminism in the choice of the rule to be fired provided by the $\omega_p$ semantics is preserved, since all the priorities of $\texttt{ACTIVATE\_RULE}(i)$ are equal.

The following result shows that the qualified answers are preserved by our encoding. Its proof, in Appendix, follows the lines of the reasoning informally explained above. The functions $\mathcal{INP}()$ and $\mathcal{OUT}()$ are those defined in Sect. 6.2.1, while $\mathcal{T}()$ is defined before.

**Theorem 6.3.** *The triple ($\mathcal{T}()$, $\mathcal{INP}()$, $\mathcal{OUT}()$) provides an acceptable encoding between CHR$^{\omega_p}$ and $static$ CHR$^{\omega_p}$.*

Analogously to the case of previous section, previous result implies that there exists an acceptable encoding for data sufficient answers from CHR$^{\omega_p}$ into $static$ CHR$^{\omega_p}$.

## 6.3 Separation Results

In this section, we prove that priorities do augment the expressive power of CHR. To do so, we prove that there exists no acceptable encoding from $static$ CHR$^{\omega_p}$ into CHR$^{\omega_t}$.

In order to prove this separation result, we need the following lemma which states a key property of CHR computations under the $\omega_t$ semantics. Essentially it says that, given a program $P$ and goal $G$, if there exists a derivation for $G$ in $P$ which produces a qualified answer $(d, K)$ where $d$ is a built-in constraint, then when considering the goal $(d, G)$ we can perform a derivation in $P$, which is essentially the same of the previous one, with the only exception of a solve transition step (in order to evaluate the constraint $d$). Hence, it is easy to observe that such a new computation for $(d, K)$ in $P$ will terminate producing the same qualified answer $(d, K)$.

The proof of the following Lemma is then immediate.

**Lemma 6.2.** *Let $P$ be a CHR$^{\omega_t}$ program and let $G$ be a goal. Assume that $G$ in $P$ has the qualified answer $(d, K)$. Then the goal $(d, G)$ has the same qualified answer $(d, K)$ in $P$.*

Lemma 6.2 is not true anymore if we consider CHR$^{\omega_p}$ programs. Indeed if we consider the program $P$ consisting of the rules

$$1 :: h(X) \Leftrightarrow X = yes | false$$

$$2 :: h(X) \Leftrightarrow X = yes$$

then the goal $h(X)$ has the qualified answer $X = yes$ in $P$, while the goal $X = yes, h(X)$ has no qualified answer in $P$. With the help of the previous lemma, we can now prove our main separation result.

**Theorem 6.4.** *There exists no acceptable encoding for data sufficient answers from CHR$^{\omega_p}$ into CHR$^{\omega_t}$.*

*Proof.* The proof is by contradiction. Consider the following program $P$ in CHR$^{\omega_p}$

$$1 :: h(X) \Leftrightarrow X = yes | false$$

$$2 :: h(X) \Leftrightarrow X = yes$$

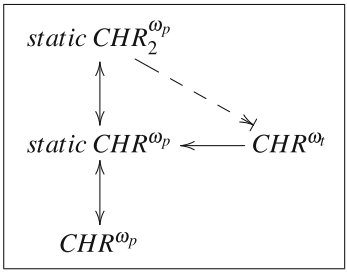



**Fig. 6.1** Graphical summary: $\dashrightarrow$: absence of an acceptable encoding $\rightarrow$: presence of an acceptable encoding

and assume that $(\gamma(), \mathcal{INP}(), \mathcal{OUT}())$ is an acceptable encoding for data sufficient answers from CHR$^{\omega_p}$ into CHR$^{\omega_t}$.

Let $G$ be the goal $h(X)$. Then $\mathcal{SA}_P(G) = \{X = yes\}$. Since the goal $h(X)$ has the data sufficient answer $X = yes$ in the program $P$ and since the encoding preserves data sufficient answers, $\mathcal{QA}_{\gamma(P)}(\mathcal{INP}(a(X)))$ contains a qualified answer $S$ such that $\mathcal{OUT}(S) = (X = yes)$. Moreover, since the output decoding function is such that the built-ins appearing in the answer are left unchanged, we have that $S$ is of the form $(X = yes, K)$, where $K$ is a (possibly empty) multiset of CHR constraints.

Then since the goal-encoding function is such that the built-ins present in the goal are left unchanged $\mathcal{INP}(X = yes, h(X)) = (X = yes, \mathcal{INP}(h(X)))$ and therefore from previous Lemma 6.2, it follows that the program $\gamma(P)$ with the goal $\mathcal{INP}(X = yes, h(X))$ has the qualified answer $S$.

However, $(X = yes, h(X))$ has no data sufficient answer in the original program $P$. This contradicts the fact that $(\gamma(), \mathcal{INP}(), \mathcal{OUT}())$ is an acceptable encoding for data sufficient answers from CHR$^{\omega_p}$ into CHR$^{\omega_t}$, thus concluding the proof.

Since the existence of an acceptable encoding implies the existence of an acceptable encoding for data sufficient answers we have the following immediate corollary:

**Corollary 6.2.** *There exists no acceptable encoding from CHR$^{\omega_p}$ into CHR$^{\omega_t}$.*

## 6.4 Summary and Related Works

We have studied the expressive power of CHR with priorities and we have shown that, differently from the case of standard CHR, allowing more than two atoms in the head of rules does not augment the expressive power of the language. We have also proved that dynamic priorities do not increase the expressive power w.r.t. static ones. These results are proved by providing translations from *static* CHR$^{\omega_p}$ into *static* CHR$_2^{\omega_p}$ and from CHR$^{\omega_p}$ into *static* CHR$^{\omega_p}$ which preserve the standard observables of CHR computations (qualified answers). A graphical representation of the results is shown in Fig. 6.1.

On the other hand, we have proved that, when considering the theoretical semantics, there exists no acceptable encoding of CHR with (static) priorities into standard CHR. This means that, even though both languages are Turing powerful, priorities augment the expressive power of the language in a quite reasonable sense, as discussed in the introduction.

Among the other few chapters which consider the expressive power of CHR a quite relevant one is [117], where the authors show that it is possible to implement any algorithm in CHR in an efficient way, i.e., with the best known time and space complexity. This result is obtained by introducing a new model of computation, called the CHR machine, and comparing it with the well-known Turing machine and RAM machine models. Earlier works by Frühwirth [44, 45] studied the time complexity of simplification rules for naive implementations of CHR. In this approach an upper bound on the derivation length, combined with a worst-case estimate of (the number and cost of) rule application attempts, allows to obtain an upper bound of the time complexity. The aim of all these works is different from ours, even though they can be used to state that, in terms of classical computation theory, $\text{CHR}^{\omega_p}$ is equivalent to CHR.

Another chapter which studies the expressive power of CHR is [116], where the author shows that several subclasses of CHR are still Turing-complete, while single-headed CHR without host language and propositional abstract CHR are not Turing-complete. Recently, these results have been further extended in [31].

Our notion of acceptable encoding has been recently used in [14] to justify a source-to-source transformation.

When moving to the more general field of concurrent languages one can find several works related to the present one. In particular, concerning priorities, [125] shows that the presence of priorities in process algebras does augment the expressive power. More precisely the authors show, among other things, that a finite fragment of asynchronous CCS with (global) priority can not be encoded into $\pi$-calculus nor in the broadcast based b-$\pi$ calculus. This result is related to our separation result for $\text{CHR}^{\omega_p}$ and CHR, even though the formal setting is completely different.

More generally, often in process calculi and in distributed systems separation results are obtained by showing that a problem can be solved in a language and not in another one (under some additional hypothesis, similar to those used here). For example, in [102] the author proves that there exists no *reasonable* encoding from the $\pi$-calculus to the asynchronous $\pi$-calculus by showing that the symmetric leader election problem has no solution in the asynchronous version of the $\pi$-calculus. A survey on separation results based on this problem can be found [126].

# Part III
# Solving Constraints Exploiting Concurrent Systems

# Chapter 7
# Constraints in Clouds

Cloud computing, introduced in Sect. 3.1.3, can be a powerful infrastructure for solving Constraint Satisfaction Problems (CSPs) in parallel since sometimes dedicated machines are too expensive to afford. In a scenario where a lot of CSPs need to be solved concurrently, the resources provided by a cloud can be used with four main advantages with respect to the use of a dedicated machine: flexibility, scalability, cost effectiveness, and reliability. The cloud allows flexibility and scalability since borrowing or releasing resources can be done very easily at runtime following the variation of the computational requirements needed to solve the CSPs. Moreover, since cloud services providers like Amazon give discounts (e.g., Amazon spot instances are cheaper than normal nodes), the cloud-based constraint solver can dynamically decide to exploit the possible abundance of resources to get solutions faster by using more than one CSP solver in parallel. It is indeed proven that running different solvers in a sequential order or in parallel is very effective for reducing the solving time of CSPs. As far as reliability is concerned, using cloud computing can improve the hardware failure tolerance: when a node fails the only thing to do to cope with the failure is to borrow a new resource.

In this chapter, we propose a framework called Constraint in Clouds (CiC) that exploits a distributed system to solve CSPs. The framework allows a user to send a CSP instance to the system that deals with the solving process autonomously and returns the solution of the problem. The framework could have more than one user at the same time, hence it needs to deal with more than one CSP instance in parallel. The goal of CiC is to minimize the time users wait for answers and the cost of the resources used to get the solutions to the CSPs.

Technically, considering the definition of Cloud computing in Sect. 3.1.3, we can state that our goal is to provide a Cloud Software as a Service for solving CSPs and deploy it in a Community cloud. Alternatively, one can think of this framework as a volunteer computing project for solving CSPs. Indeed, the key idea of volunteer computing is to solve problems using distributed computing resources donated by others. It was first used in 1996 by the project called Great Internet Mersenne Prime Search (http://www.mersenne.org/), and was followed by other

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_7, © Atlantis Press and the authors 2014

academic or commercial projects like SETI@home (http://setiathome.ssl.berkeley.edu/) and Folding@home (http://folding.stanford.edu/) projects launched in 1997. These projects received considerable media coverage, and each one attracted several hundred thousand volunteers.

We would like to design our architecture using the SOC paradigm described in Sect. 3.5 and implementing it using Jolie [94] because in this way we could obtain

- scalability. Massive number of communications with different computers can easily be handled;
- modularity. New services can easily be integrated and organized in a hierarchy. This is particularly important in an architecture like ours which should be divided into modules or sub services;
- deployment. Jolie allows us to deploy the framework in a number of different ways. It provides interaction between heterogeneous services, like in the case of web services (e.g., integrating a Google map application in a hotel-search application). We can therefore easily interact with other services (even graphical ones) in the future and make our architecture be part of a more complex system.

In this chapter, we will first present a brief overview of what has been done in the literature to solve CSPs in parallel. In Sect. 7.2, we present in more detail the CiC framework; in Sect. 7.3, we describe a first CiC prototype and some tests we performed with it; in Sect. 7.4, we present a brief summary.

In the next chapters, we show other attempts to improve the CiC framework or its implementation. In particular, smart ways of assigning resources are studied in Chap. 8 while Chaps. 9 and 10 describe how Jolie, but in principle also other SOC language, can be extended with features that improve the efficiency and the conciseness of the implementation of the CiC framework.

Note that a presentation of Jolie, the language used to develop the prototype, is beyond the scope of this thesis. In Chap. 10, we will just present the theoretical model behind Jolie. For more details about this language please refer to [94].

## 7.1 Parallel Constraint Solving

In the literature, a lot of attempts to parallelize constraint solving have been tried. Following [22], the main approaches to parallel constraint solving can roughly be divided into the following main categories:

- Search-space splitting approach. It explores the parallelism provided by the search space, when a branching is done the different branches can be explored in parallel (OR-parallelism). One challenge with this approach is load balancing: the branches of a search tree are typically extremely imbalanced and require a non-negligible overhead of communication for balancing the load. Recent works based on this approach are, e.g., [68, 88,134];

- Portfolio approach. It explores the parallelism provided by different viewpoints on the same problem, for instance different algorithms or parameter tunings. This idea has also been exploited in a nonparallel context, e.g., [35, 56, 132]. One challenge here is to find a scalable source of diverse viewpoints that provide orthogonal performance and is therefore of complementary interest;
- Problem splitting approach. The instance itself is split into pieces to be solved by each processor. One challenge here is that because no processor has a complete view on the problem, it typically becomes much more difficult to solve an instance. Instance splitting typically relates to distributed CSPs [133] which assumes that the instance is naturally split between agents, for instance for privacy reasons;
- Other approaches. Typically based on the parallelization of one key algorithm of the solver, for instance constraint propagation. Usually, parallelizing propagation is challenging [70] and the scalability of this approach is limited by Amdahl's law: if propagation consumes 80 % of the runtime, then by parallelizing it, even with a massive number of processors, the speedup that can be obtained will be under five. Some other approaches focus on particular topologies or make assumptions on the problem.

Search-space splitting and portfolio approaches are the most likely to offer scalable speedups. Note that even for these approaches scalability issues are yet to be investigated: most related works use a number of processors between 4 and 16; the only exception we are aware of is [68] where 61 CPUs are used in the context of search-space splitting and [22] where they use up to 128 CPUs in the context of portfolio and search-space splitting. To our knowledge, no one has ever tried to solve CSPs in a massively parallel system with more than 128 processors. Our goal is to go beyond this limit and use thousands of CPUs for solving CSPs if it is necessary.

In this context, we would also like to mention cpHydra [35], which is a sequential CSP solver that uses a portfolio approach and machine learning algorithm to speed up the solving process. cpHydra, which is the winner of the 2008 CSP Solver Competition [1], combines many CP solvers in a portfolio and determines via case-base reasoning the subset of the solvers to use in an interleaved fashion and the time to allocate for each solver, given a CSP instance. In this work, we propose to use a similar case base reasoner algorithm, or alternatively another machine learning algorithm, to forecast the execution times of our solvers and using these data to speed up the solving of the instances submitted to the system.

## 7.2 The CiC Framework

The CiC framework is composed of four main kinds of entities, viz., distributor, learner, worker, and preprocessor. The most important component of the framework is the distributor that has the task of coordinating the workers for solving the CSPs. The distributor assigns jobs to the workers. A job is a request to solve a given CSP instance using a certain solver. The distributor then collects the results of the workers'

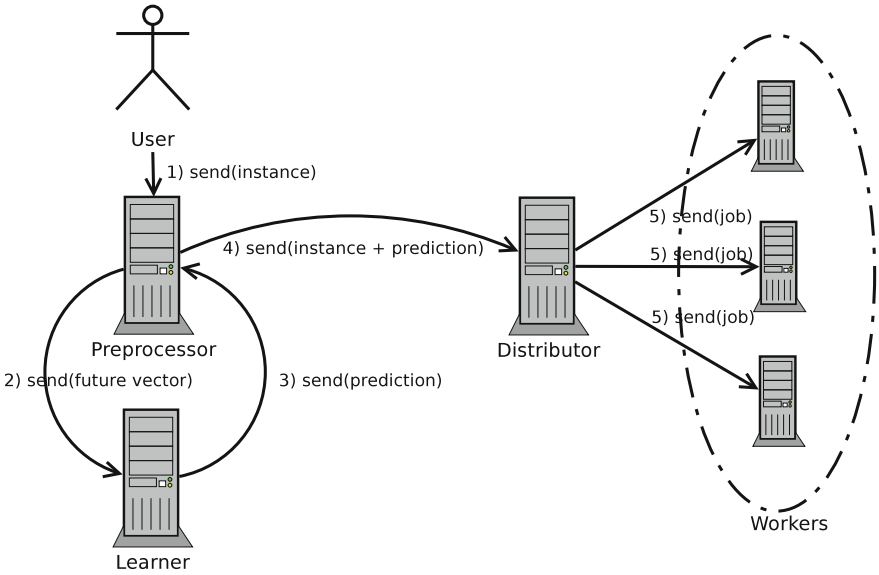



**Fig. 7.1** Flow exchange between entities of the framework

computation. Its goal is to dispatch the jobs minimizing the time or the cost needed to solve the CSPs exploiting informations provided by the learner that can be viewed as an oracle that tries to predict how difficult a CSP is and what is the best solver to employ. To make these predictions, the learner uses machine learning algorithms that are trained over a well-known dataset of CSP instances.

The workers are the components that are running the solvers and are searching the solution of the CSPs. They are only required to run some solvers taken from a fixed portfolio, receive a job from the distributor, and send back the solution of the problem. Even though all the components of the framework are intended to be run on a cloud, it is vital to the system to deploy at least these workers on the cloud because the distributor should be able to dynamically borrow more workers.

The last component of the framework is the preprocessor that has to collect the CSP instances submitted by the users, test if the instances are well-defined, and extract from them some statistics that are later used by the learner and distributor. For instance, the preprocessor is responsible for the extraction of the feature vector needed by the machine learning algorithm of the learner.

Figure 7.1 depicts how the system behaves when a user submits the request of solving a CSP instance. The user sends to the preprocessor a problem instance $i$. Once $i$ is sent, the preprocessing computes a feature vector, i.e., a list of statistics of the CSP instance submitted like the number of constraints. The feature vector is then sent to the learner and used to predict how difficult the problem is and what solver among a portfolio of solvers is the fastest one for the given instance. The learner returns these predictions to the preprocessor that forwards them and the CSP

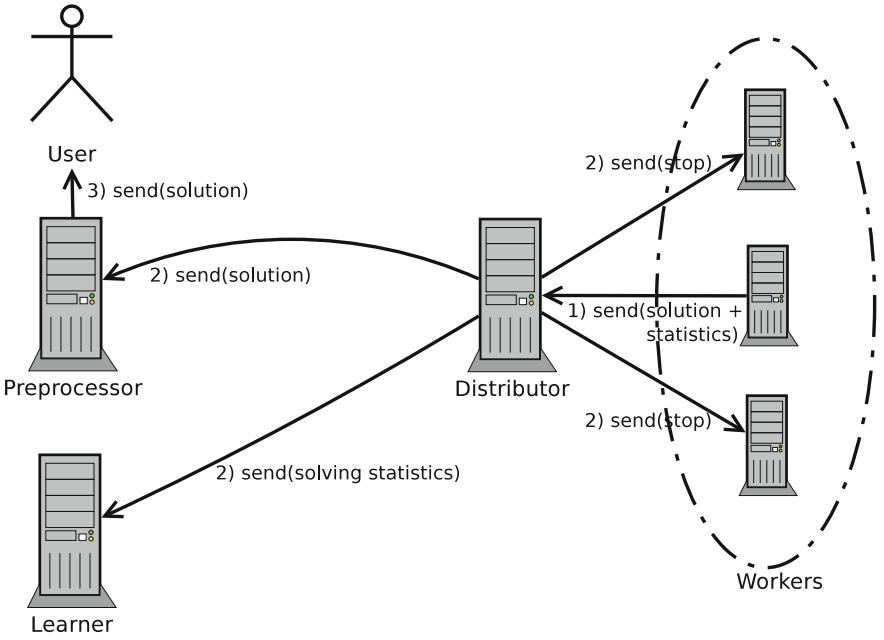



**Fig. 7.2** Flow exchange between entities of the framework

instance to the distributor. The distributor uses all the given information to coordinate all the workers that are needed to solve the instance in the most cost-effective way. In particular, it decides which worker should try to solve the problem; it selects what solver the worker should use and assigns the job to the worker.

Figure 7.2 depicts the messages exchanged when a worker has solved the problem. In this case, the worker sends to the distributor the solution along with some solving statistics like the runtime of the solver and the memory that was used. The distributor can decide to stop the other workers that are working on the same problem, it forwards the solution to the preprocessor and the solving statistics to the learner. The preprocessor forwards the solution to the user while the learner collects the new solving statics in order to improve the prediction models for increasing the future accuracy of the predictions.

To use the framework, a common language to specify the CSP instance should be used. Unfortunately, the CP community lacks a standardized representation of problem instances and this still limits the acceptance of CP by the business world. One attempt to overcome this problem was taken by the Association for Constraint Programming with the Java Specification Request JSR-331 “Constraint Programming API” [2, 4]. The goal of this specification is the creation of a powerful API for specifying CP problems. In the last 5 years, other approaches focusing on more low-level languages emerged. The aim of these approaches is to define a minimal domain-dependent language that supports all the major constraint features and requires,

at the same time, a minimal implementation effort to be supported by constraint solvers. Two languages following this goal are worth mentioning: FlatZinc [11] and XCSP [101]. The first language was originally created to be the target into which a higher level constraint model (e.g., MiniZinc [99]) is translated. Today, FlatZinc is also used as a low level "lingua franca" for solver evaluation and testing. For instance, since 2008, FlatZinc has been used in the MiniZinc Challenge [3, 119], a competition where different solvers are compared by using a benchmark of MiniZinc instances that are compiled into FlatZinc. XCSP is a language structurally very similar to FlatZinc. It was defined with the purposes of being a unique constraint model that could be used by all the CP solvers. It was first proposed in 2005 for the solvers competing in the CSP International Solver Competition [1] and has then been used in other contexts and extended. For specifying the instance for the cloud base constraint solver framework we have decided to use XCSP, specifically its 2.1 version.

The Extensible Markup Language (XML) is a simple and flexible text format playing an increasingly important role in the exchange of a wide variety of data on the Web. The objective of the XSCP is to ease the effort required to test and compare different algorithms by providing a common testbed of constraint satisfaction instances. The proposed representation is low level: for each instance the domains, variables, relations (if any), predicates (if any), and constraints are exhaustively defined. No control flow constructs like "for" cycles or "if then else" statements can be used.

Roughly speaking, there exist two variants of this format: a fully tagged representation and an abridged representation. The first one is a full XML, completely structured representation which is suitable for using generic XML tools but is quite verbose and tedious to use for a human being. The second representation is just a shorthand notation of the first one and it is easier to read and to write for a human being, but less suitable for generic XML tools.

XCSP is the perfect choice to be used to specify CSPs in our framework because:

- it has been used in the last constraint solver competitions and thus many solvers support it;
- such a low-level representation is useful to extract the feature vectors.
- XML is one of the most used document formats to transmit data in clouds and service-oriented systems.

As already mentioned, modularity, scalability, and reliability are some of the major concerns that guided the design of the framework. We think that the separation of the task of solving a CSP into four loosely coupled subtask makes the system modular. For instance, a new feature like the support of the FlatZinc format can be easily added simply allowing the preprocessor the receive FlatZinc programs and convert them into XCSP. Moreover, since replication is the main technique used in distributed system to cope with hardware failures and spikes of computational demands, to obtain a scalable and reliable framework we allow every entity of the system to be replicated. In the CiC framework, for instance, the preprocessor entity could be deployed in

more than one physical machines, every one of them performing the same task in parallel, and potentially allowing the preprocessing of a huge number of instance submissions in parallel.

## 7.3 First Experiments

We have implemented a prototype as a proof of concept to check if the CiC framework was feasible and scalable. We were not aiming at providing from the beginning an implementation with all the features we previously described. For this reason, for example, the development of the learner was postponed.

The components of the system (viz. preprocessor, worker, and distributor) were developed as Jolie services. We chose to implement one of the simplest dispatching strategies: we let the system try to solve every CSP with all the solvers available in the portfolio using all the available workers. In particular, the dispatcher assigns to a free worker the job of solving the oldest instance, i.e., the instance that was submitted first, using a solver that was not already tried for the given instance. When the solution of the problem was received, the workers solving the same instance were not interrupted. Moreover, we forbid the system to assign a job for solving an instance if the instance was already solved, this even in the case a solver of the portfolio was not used to solve the given instance.

For the testing of this naive system, we deployed the services on Dell Optiplex computers running Linux with Intel core 2 duo and Pentium 4 processors. Up to 100 of them were employed for the workers and only one for both the distributor and the preprocessing services.

To evaluate the system, as a benchmark, we consider the instances of the 2009 CSP Solver Competition [1]. The solver portfolio was composed of six solvers: Abscon 112v4 AC, Abscon 112v4 ESAC, Choco2.1.1 2009-06-10, Choco2.1.1b 2009-07-16, Mistral 1.545, SAT4J CSP 2.1.1 that were used in the 2009 CSP Solver Competition and one, bpsolver 2008-06-27, used in the 2008 competition. These solvers were provided as black box, hence their tunings were not possible.

The experiments we performed focused on the following instances: (i) Easy SAT: 1607 satisfiable instances solved in less than 1 min; (ii) Easy UNSAT: 1048 unsatisfiable instances solved in less than 1 min; (iii) Hard SAT: 207 satisfiable instances solved in between 1 and 30 min; (iv) Hard UNSAT: 106 unsatisfiable instances solved in between 1 and 30 min. Such times refer to the solving times of the competition.

In Table 7.1, we present the number of instances solved in 30 min for the easy instances and in 1 h for the hard instances. Every experiment was performed and reported for three runs. The results we obtained are promising. Even without the learner and using a small portfolio of solvers, the number of the instances solved in a fixed amount of time increases as the number of computers increases. Moreover, note that only one computer was used to run the preprocessing and the distributor services, and yet the system can handle 100 computers without any problems.

**Table 7.1** Experimental results

| n° | Easy SAT (30 min) | | | Easy UNSAT (30 min) | | | Hard SAT (1 h) | | | Hard UNSAT (1 h) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 14 | 15 | 17 | 18 | 18 | 3 | 3 | 6 | 7 | 7 | 9 |
| 40 | 132 | 128 | 135 | 150 | 150 | 150 | 8 | 8 | 7 | 16 | 17 | 13 |
| 60 | 141 | 140 | 140 | 320 | 318 | 322 | 19 | 15 | 14 | 23 | 23 | 22 |
| 80 | 144 | 145 | 151 | 335 | 323 | 328 | 25 | 21 | 25 | 29 | 30 | 30 |
| 100 | 179 | 179 | 192 | 336 | 345 | 334 | 25 | 25 | 25 | 44 | 33 | 36 |

We also used the testing data to evaluate the limits of the choices we made in the development of the prototype. For instance, we chose not to interrupt workers because the data collected running every solver for every instance could be used to obtain a better learner. However, looking at the runs, we noticed that sometimes a solver is able to solve an instance very quickly while all the other solvers require a very long time. Hence, letting all the solvers run for every instance could be a huge waste of computing resources. In Chap. 8, we focus on this problem studying strategies to reach good compromises between the number of instances solved and the cost of resources that need to be used.

## 7.4 Summary

In this chapter, we have proposed a new framework for solving CSPs exploiting the resources of a cloud. For this reason, in the design process phase, we focused our attention toward scalability and modularity concerns. This leads us to the creation of a system composed of four separate components, each of them necessary for the efficient execution of the system. Moreover, the components and their interaction were designed to be replicated allowing the system to be more reliable and able to adapt to huge loads of CSP solving requests.

We presented a proof-of-concept implementation using the SOC language Jolie. Although some important features are not implemented in this first prototype, we showed that the system is indeed scalable and is able to exploit a cloud environment. The use of a SOC language was proven successful since it allows us to implement the system following the framework abstraction in a straightforward way without having a significant loss of performances. However, using the Jolie language, we have also noticed some of the limitations of the SOC languages. In particular, SOC languages do not support broadcasting primitives that can be used to describe the communication between workers and dispatcher in a more concise and efficient way. Another limit of Jolie is related to the mechanism that it uses for handling time-outs. In Chaps. 9 and 10, we present in detail these limitations and we propose an extension to Jolie in order to improve the implementation of the CiC framework.

# Chapter 8
# A Classification-Based Approach to Manage a Solver Portfolio

The past decade has witnessed a significant increase in the number of constraint solving systems deployed for solving constraint satisfaction problems (CSP). It is well recognized within the field of constraint programming that different solvers are better at solving different problem instances, even within the same problem class [56]. It has been shown in other areas, such as satisfiability testing [131], and integer linear programming [82], that the best on-average solver can be outperformed by a portfolio of possibly slower on-average solvers. This selection process is usually performed using a machine learning technique based on feature data extracted from CSPs.

Three specific approaches that use contrasting approaches to portfolio management in CSP, sat and qbf are CPHydra, SATzilla, and Acme, respectively. CPHydra is a portfolio of constraint solvers exploiting a case-base of problem-solving experience [35]. CPHydra combines case-base reasoning of machine learning with the idea of partitioning cpu-time between components of the portfolio in order to maximize the expected number of solved problem instances within a fixed time limit. SATzilla [131] builds runtime prediction models using linear regression techniques based on structural features computed from instances of Boolean satisfiability problem. Given an unseen instance of the satisfiability problem, SATzilla selects the solver from its portfolio that it predicts to have the fastest running time on the instance. The Acme system is a portfolio approach to solve quantified Boolean formulae, i.e., SAT instances with some universally quantified variables [106].

In this chapter, we present a very different approach to managing a portfolio for constraint solving when the objective is to solve a set of problem instances so that the average completion time, i.e., the time at which we have either found a solution or proven that none exist, of each instance is minimized. This scenario arises in a context in which problem instances are submitted to the CiC framework presented in Chap. 7. In addition, there is a significant scheduling literature that focuses on minimizing average completion time, much of which is based around the use of dispatching heuristics [128].

The approach we propose is strongly inspired by dispatching rules for scheduling. Our approach is conceptually simple, but powerful. Specifically, we propose the use

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_8, © Atlantis Press and the authors 2014

of classifier techniques as a basis for making high-level and qualitative statements about the solvability of CSP instances with respect to a given solver portfolio. We also use classifier techniques as a basis for a dispatching-like approach to solve a set of problem instances in a single processor scenario. We show that when runtimes are properly clustered, simple classification techniques can be used to predict the class of runtime as, for example, short, medium, long, time-out, etc. We show that, even considering a processing system with only one computational node, this approach significantly outperforms a well-known general-purpose CSP solver and performs well against an oracle implementation of a portfolio. We then show what happens if we apply the approach for systems having more than one computational unit.

Clearly, these studies were conducted to evaluate what are the best machine learning algorithms and the best distribution strategy to use for the learner and the distributor entities of the framework presented in Chap. 7.

The remainder of this chapter is organized as follows. In Sect. 8.1, we summarize the requisite background on constraint satisfaction and machine learning required for this chapter. Section 8.2 presents the large collection of CSP instances on which we base our study. We discuss the various classification tasks upon which our approach is based in Sect. 8.3, and evaluate the suitability of different representations and classification for these tasks in Sect. 8.4. We demonstrate the utility of our classification-based approach for managing a solver portfolio exploiting system with one or more processors in Sects. 8.5 and 8.6 respectively. We discuss related work in Sect. 8.7 and summarize in Sect. 8.8.

## 8.1 Preliminaries

Machine learning *"is concerned with the question of how to construct computer programs that automatically improve with experience."* It is a broad field that uses concepts from computer science, mathematics, statistics, information theory, complexity theory, biology, and cognitive science [93]. Machine learning can be applied to well-defined problems, where there is both a source of training examples and one or more metrics for measuring performance. In this chapter, we are particularly interested in classification tasks. A classifier is a function that maps an *instance* with one or more discrete or continuous features to one of a finite number of classes [93]. A classifier is trained on a set of instances whose class is already known, with the intention that the classifier can transfer its training experiences to the task of classifying new instances.

## 8.2 The International CSP Competition Dataset

We focused on demonstrating our approach on as comprehensive and realistic a set of problem instances as possible. Therefore, we constructed a comprehensive dataset of CSPs based on the various instances gathered for the annual International CSP Solver

Competition [1] from 2006–2008. An advantage of using these instances is that they are publicly available in the XCSP format [101]. The first competition was held in 2005, and all benchmark problems were represented using extensional constraints only. In 2006, both intentional and extensional constraints were used. In 2008, global constraints were also added. Overall, there are *five categories of benchmark problems* in the competition: 2-ARY-EXT instances involving extensionally defined binary (and unary) constraints; N-ARY-EXT instances involving extensionally defined constraints, at least one of which is defined over more than two variables; 2-ARY-INT instances involving intensionally defined binary (and unary) constraints; N-ARY-INT instances involving intensionally defined constraints, at least one of which is defined over more than two variables; and, GLB instances involving any kind of constraints, including global constraints.

The competition required that any instance should be solved within 1,800 s. Any instance not solved by this cut-off time was considered unsolved. To facilitate our analysis, we remove from the dataset any instance that could not have been solved by any of the solvers of our portfolio by the cut-off. In total, our dataset contains around 4,000 instances across these various categories. Later, we will further restrict ourselves to a challenging subset of these.

## 8.3 From Runtime Clustering to Runtime Classification

We show how clusters of runtimes can be used to define classification problems for a dispatching-based approach to managing an algorithm portfolio. While our focus here is not to develop the CPHydra system, we will, for convenience, use its constituent solvers and feature descriptions of problem instances to build our classifiers. We demonstrate our approach on a comprehensive and realistic set of problem instances.

Based on the three solvers used in the 2008 CSP Solver Competition variant of CPHydra we present in Fig. 8.1a, b, and c the runtime distributions for each of its solvers, Mistral, Choco, and Abscon, respectively,[1] showing for every solver in the portfolio the number of instances of the dataset solved in given time windows. Having removed from the dataset any instance that could not have been solved by any of these solvers within a 1,800 s time-limit, each instance is ensured to be solved by at least one solver. However, it is not the case that each solver finds that the same instances are either easy or hard. There are many instances, as we will show below, for which one of the solvers decides the instance quickly, while another solver struggles to solve it. Therefore, we define a classification task that given a CSP instance returns the fastest solver for that instance.

[1] Visit the competition site for links to each of the solvers.

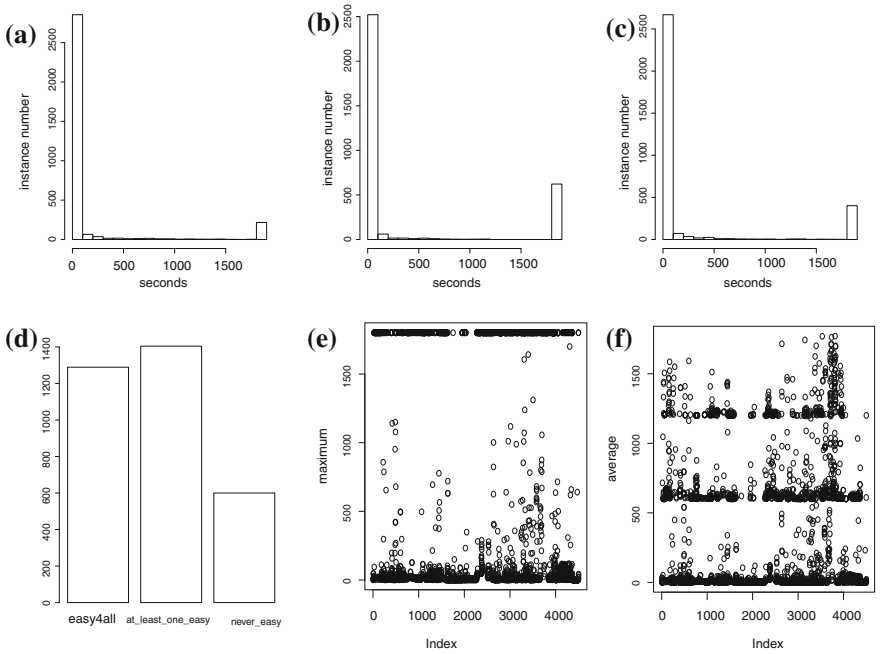



**Fig. 8.1** The performance of each solver in the portfolio on the dataset. **a** Runtimes for Mistral. **b** Runtimes for Choco. **c** Runtimes for Abscon. **d** Occurrence of each class for $3C_{10}$ in the dataset. **e** The maximum runtimes of the portfolio on each instance. **f** The average runtimes of the portfolio on each instance

**Classification 1 (Fastest Solver Classification** (FS)) *Given a CSP instance $i$ and a CSP solver portfolio $\Pi$, the* FS *classification task is to predict which solver in $\Pi$ gives the fastest runtime on $i$.*

From Fig. 8.1 it is clear that there are many instances that can be solved easily or not at all. To capture this property we introduce a classification task called 3C which is defined over a solver portfolio as follows.

**Classification 2** ($3C_k$) *Given a CSP instance $i$ and a CSP solver portfolio $\Pi$, the $3C_k$ classification task is to predict whether $i$: (a) can be solved by all solvers in $\Pi$ in at most $k$ seconds; (b) can be solved by at least one solver, but not all, in $\Pi$ in at most $k$ seconds; or (c) takes more than $k$ seconds to solve with each solver in $\Pi$.*

The number of instances in our CSP dataset in each class is presented in Fig. 8.1d. Note that while many instances were easy (i.e., solvable within 10 s) for all solvers, a larger number were easy for some, but not all (the middle stack in the histogram). We consider two additional classifiers related to the performance of the portfolio as a whole. We compute the maximum and the average time required by each solver in the portfolio to solve each instance. The maximum times are presented in Fig. 8.1e,

in which the x-axis lists the index of each instance and the y-axis represents the maximum runtime. Note that a time limit of 1,800 s was applied on the dataset which gives the upper bound of the maximum solving time and which is why there are a number of instances presenting across the top of the plot. For applying this classifier, we consider only two intervals of running time according to the data presented in Fig. 8.1e: at most 1,500 s and greater than 1,500 s.

**Classification 3** (MAXC$_{ks}$) *Given a CSP instance $i$ and a CSP solver portfolio $\Pi$, the* MAXC$_{ks}$ *classification task is to predict which interval of running-times in $ks$ that instance $i$ can be solved using the worst performing solvers from $\Pi$.*

Similarly, the average times are presented in Fig. 8.1f. In this plot, we note that there are three distinct classes of runtimes: instances that take on average between 0 and 600 s, between 601 and 1,200, and more than 1,200. Again, this division is influenced by the fact that an instance's maximum solving time is at most 1,800 s.

**Classification 4** (AVGC$_{ks}$) *Given a CSP instance $i$ and a CSP solver portfolio $\Pi$, the* AVGC$_{ks}$ *classification task is to predict which interval of running-times in $ks$ that instance $i$ can be solved taking the average solving times for each of the solvers in $\Pi$.*

To complement the AVGC classifier, we will also make use of a classifier that considers the variance, or spread, of runtimes across the constituent solvers of a portfolio over a given instance. We refer to this classifier as SPREAD.

**Classification 5** (SPREAD$_k$) *Given a CSP instance $i$ and a CSP solver portfolio $\Pi$, the* SPREAD$_k$ *classification task is to predict whether the difference across the runtimes of the constituent solvers is at most (or at least) $k$.*

For applying this classifier, we consider the difference of at most 100 s, based on the given runtimes.

The classifiers presented in this section define a very expressive qualitative language to describe the expected performance of a solver portfolio on a given CSP instance. For example, we can make statements like *"this instance is easy for all solvers in the portfolio,"* or *"this instance is easy for some, but not all solvers, but the average running time is low and has low variation."* This contrasts with all current approaches to managing a solver portfolio. As our empirical results will demonstrate, this approach is also very powerful in terms of efficient solving.

## 8.4 Experiments in Runtime Classification

In this section, we experiment with the various classification problems discussed in Sect. 8.3. To do so, we first establish "good" features to represent CSPs starting from the features used in CPHydra. The objective of these experiments is to show that accurate classifiers for solver runtimes can be generated, and that these can be successfully used to build effective dispatching heuristics for managing a solver

portfolio for CSPs. We only focus on the 3C, AvgC, and MaxC classifiers. The experimental dataset is based on the 2008 International CSP Solving Competition. We consider a portfolio comprising three solvers: Abscon, Choco, and Mistral. Running times for each of these solvers are available from the CSP competition's web site. A time-out on solving time is imposed at 1,800 s. We exclude from the dataset the CSP instances that cannot be solved in that amount of time and also some other instances due to the reason that we will explain later. In total, our final dataset, upon which we run our experiments comprises 3293 CSP instances.

*Knowledge Representation and Classifiers*. Since the selection of good features has a significant impact on the classifier performances, we investigate which ones are more suitable to capture problem hardness. In particular, we consider three feature-based representations of the CSP instances in our dataset: SATzilla features representing each CSP instance encoded into SAT, those features used by CPHydra(with some modifications), and the combination of the two.

SATzilla uses a subset of the features introduced by [100]: starting from 84 features they discard those computationally expensive and too instable to be of any value. At the end they consider only 48 features that can be computed in less than a minute (for more information see [130]). In this work, we are able to use directly these features simply translating each competition CSP instance into SAT using the Sugar solver[2] and then using SATzilla to extract its feature description. In some (but few) cases, the encoding of a CSP instance into SAT requires an excessive amount of time (i.e., more than a day). In order to make a fair comparison between the set of features, we simply dropped such instances from the dataset.

For the second feature representation, we started from the 36 features of CPHydra. While the majority of them are syntactical, the remaining is computed by collecting data from short runs of the Mistral solver. Among the syntactical features, worth mentioning are the number of variables, the number of constraints and global constraints, the number of constants, the sizes of the domains, and the arity of the predicates. The dynamic features instead take into account the number of nodes explored and the number of propagations done by Mistral with a time limit of 2 s. When we extracted the CPHydra features using our dataset we noticed that two of them (viz. the logarithm of the number of constants and the logarithm of the number of extra values) were strongly correlated with other features. Since strongly correlated features are not useful for discriminating between different problems, we discarded these two features. Inspired by Nudelman et al. [100], we considered additional features like the ratio of the number of constraints over the number of variables, and the ratio of the number of variables over the number of constraints. Moreover, we added features representing an instance's *variable graph* and *variable-constraint graph*. In the former, each variable is represented by a node with an edge between pairs of nodes if they occur together in at least one constraint. In the latter, we construct a bipartite graph in which each variable and constraint is represented by a node, with an edge between a variable node $v$ and a constraint node $c$ if $v$ is constrained by $c$. From these

[2] http://bach.istc.kobe-u.ac.jp/sugar/.

graphs, we extract the average and standard deviation of the node degrees and take their logarithm. With these, the total amount of features we consider are 43.

The third feature-based description of the CSP instances, which we refer to as Hylla, is simply the concatenation of the two feature descriptions discussed above. We consider a variety of classifiers, implemented in publicly available tools RapidMiner[3] and WEKA.[4] Our SVM classifier is hand-tuned to the specific tasks considered in this chapter according to the best parameters found using RapidMiner; however, it is only applied to the 3C and AvgC tasks because it appeared to be problematic to tune for the MaxC task. The other WEKA classifiers are used with their default settings.

*Results.* The results of the runtime classification tasks are presented in Tables 8.1, 8.2, and 8.3. Three alternative feature descriptions, as discussed earlier, are compared; these are denoted as CPHydra, Hylla, and SATzilla, in the tables. We compare the performance of various classifiers on each of the three classification tasks (3C, MaxC, and AvgC). A 10-fold cross-validation is performed. The performance of each classifier, on each representation, on each classification task are measured in terms of the classification accuracy and the $\kappa$-statistic. The latter measures the relative improvement in classification over a random predictor [29].

Differences in performance are tested for statistical significance using a Paired t-Test at a 95 % confidence level. The performance on the CPHydra feature set is used as a baseline. In each table, values that are marked with a $\circ$ represent performances that are statistically significantly better than CPHydra, while those marked with a $\bullet$ represent performances that are statistically significantly worse.

In summary, both classification accuracies and $\kappa$ values are high across all three tasks. Interestingly, combining both CPHydra and SATzilla features improves performance in only one $\kappa$ value, and without any significant improvement in classification accuracy. The CPHydra feature set thus gives rise to the best overall performance. Based on these promising results, we consider in Sect. 8.5 the utility of using these classifiers as a basis for managing how a solver portfolio can be used to solve a collection of CSP instances.

## 8.5 Scheduling a Solver Portfolio

We now consider a solver portfolio and demonstrate the utility of our classification-based approach for its management via some experimental results. In this section, we will focus on systems having only one computational node, in Sect. 8.6 instead, we will consider the more interesting case when more than one processor could be used in parallel.

*Portfolio Construction and Its Management.* The portfolio is composed of three solvers previously introduced: Mistral, Choco, and Abscon. It is designed so as to face the challenge of solving a collection of CSP instances as quickly as possible.

[3] http://rapid-i.com/content/view/181/196/.

[4] http://www.cs.waikato.ac.nz/ml/weka/.

**Table 8.1** Classification accuracy and $\kappa$-statistics for the 3C classifier

| Classifier | (CPHydra) | (Hylla) | (SATzilla) | (CPHydra) | (Hylla) | (SATzilla) |
|---|---|---|---|---|---|---|
| trees.J48 | 83.83 | 82.52 • | 79.70 • | 0.75 | 0.73 | 0.68 • |
| meta.MultiBoostAB | 84.75 | 84.91 | 82.19 • | 0.76 | 0.76 | 0.72 • |
| trees.RandomForest | 85.34 | 85.14 | 82.39 • | 0.77 | 0.77 | 0.72 • |
| functions.LibSVM | 85.63 | 84.68 | 82.62 • | 0.77 | 0.76 • | 0.73 • |
| lazy.IBk | 83.69 | 83.53 | 78.71 • | 0.74 | 0.74 | 0.67 • |
| bayes.NaiveBayes | 61.70 | 62.74 | 52.49 • | 0.37 | 0.44 ○ | 0.31 • |
| meta.RandomCommittee | 84.99 | 85.44 | 82.57 • | 0.76 | 0.77 | 0.73 • |
| rules.OneR | 72.89 | 72.89 | 69.16 • | 0.57 | 0.57 | 0.51 • |

○, • statistically significant improvement or degradation over CPHydra

**Table 8.2** Classification accuracy and $\kappa$-statistics for the AvgC classifier

| Classifier | (CPHydra) | (Hylla) | (SATzilla) | (CPHydra) | (Hylla) | (SATzilla) |
|---|---|---|---|---|---|---|
| trees.J48 | 84.18 | 83.35 | 82.42 • | 0.63 | 0.61 | 0.58 • |
| meta.MultiBoostAB | 85.14 | 85.10 | 83.95 • | 0.65 | 0.65 | 0.62 • |
| trees.RandomForest | 85.37 | 85.53 | 84.42 | 0.65 | 0.65 | 0.62 • |
| functions.LibSVM | 84.99 | 84.24 | 83.67 • | 0.64 | 0.63 | 0.60 • |
| lazy.IBk | 83.45 | 83.13 | 81.39 • | 0.62 | 0.61 | 0.57 • |
| bayes.NaiveBayes | 64.96 | 53.81 • | 41.69 • | 0.25 | 0.22 | 0.12 • |
| meta.RandomCommittee | 85.03 | 85.32 | 84.25 | 0.65 | 0.65 | 0.62 |
| rules.OneR | 78.97 | 78.97 | 75.75 • | 0.45 | 0.45 | 0.34 • |

○, • statistically significant improvement or degradation CPHydra

**Table 8.3** Classification accuracy and $\kappa$-statistics for the MaxC classifier

| Classifier | (CPHydra) | (Hylla) | (SATzilla) | (CPHydra) | (Hylla) | (SATzilla) |
|---|---|---|---|---|---|---|
| trees.J48 | 89.61 | 89.08 | 87.99 • | 0.73 | 0.72 | 0.69 • |
| meta.MultiBoostAB | 90.19 | 90.33 | 89.47 | 0.75 | 0.75 | 0.73 |
| trees.RandomForest | 90.35 | 90.70 | 89.90 | 0.75 | 0.76 | 0.74 |
| lazy.IBk | 89.18 | 89.00 | 87.87 • | 0.73 | 0.72 | 0.70 • |
| bayes.NaiveBayes | 71.37 | 67.30 • | 54.26 • | 0.31 | 0.34 | 0.18 • |
| meta.RandomCommittee | 90.28 | 90.64 | 90.10 | 0.75 | 0.76 | 0.74 |
| rules.OneR | 82.98 | 82.98 | 77.87 • | 0.54 | 0.54 | 0.38 • |

○, • statistically significant improvement or degradation over CPHydra

Specifically, it tries to minimize the average *completion time* of each instance; the completion time of an instance is the time by which it is solved. One would wish to minimize average completion time, if for instance a CSP solver was deployed as a web service. We assume all CSP instances are known at the outset. This setting is similar to setting used in the international SAT competitions and also relevant in the context of the International CSP Competition.

Minimizing average completion time can be achieved by solving each problem in increasing order of difficulty, i.e., by using the well-known *shortest processing time* heuristic. In a solver portfolio context, this corresponds to solve an instance with the

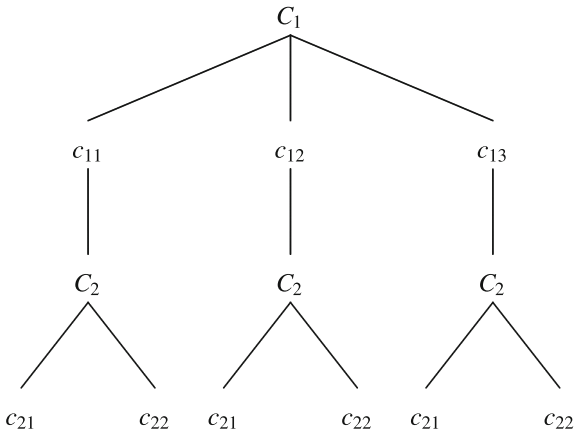



**Fig. 8.2** Instance ordering using classifiers $C_1$ (dividing in 3 classes) and $C_2$ (dividing in 2 classes)

fastest solver for it, and orders each one by solving time. This gives us the most basic oracle (perfect) approach to minimize average completion time. As it is unlikely to have such perfect information, the portfolio can be *managed* via an *instance selector* and a *solver selector* whose purpose is to predict, respectively, the right order of the instances from easiest to hard and the fastest solver on a given instance.

The classifiers developed previously can help to manage the portfolio. Consider for instance Fig. 8.1d in which we see that the instances are grouped into three: (i) those that can be solved by all solvers in 10 s; (ii) those that can be solved by at least one solver in 10 s; (iii) those that are solved by all solvers in more than 10 s. The figure exhibits a rather balanced distribution, especially between the first two classes. This hints that 3C can provide a good way to distinguish the easy instances from hard ones and that the classifiers AVGC, SPREAD, and MAXC could be useful to break ties between the instances of the second and third classes. We exemplify this approach in Fig. 8.2. Given two classifiers $C_1$ and $C_2$ with three and two classes, respectively, an instance ordering $C_1 \prec C_2$ would mean that the instances are first divided according to the predictions of $C_1$ resulting in three classes, and then those in each class of $C_1$ would be further divided according to the predictions of $C_2$, resulting in six classes in total. The ordering of the instances then would be from the leftmost to the rightmost leaf in the tree.

Even though it is not clear in which order to use our classifiers, we restrict ourselves to starting with 3C as it gives the most balanced classification among all. As we will verify empirically later, this approach yields the best performance. Once the order of the instances is determined according to the predictions of the relative classifiers, we pick the solver of an instance using FS. Given an instance and a solver chosen for the instance via the FS classifier, the solver is run on the instance for 1,800 s. In case of time-out, we switch to another solver randomly. As noted earlier, our benchmarks are composed of instances that can be solved by at least one solver by the cut-off. Our portfolio therefore guarantees that each instance is solved, independently of the way the instances are ordered and the solvers are picked up.

*Oracles, Baselines, and Experimental Methodology.* In our experiments, we compare the quality of our classifiers for managing a portfolio of solvers with various oracle-based management strategies, as well as with a best on-average solver, and naive baseline strategies. We do not compare against CPHydra since it solves a challenge specific to the International CSP Solver competition, namely to maximize the *probability* that a problem instance is solved within 1,800 s. Instead, we aim at solving each instance as quickly as possible.

The most basic oracle-based (perfect) management strategy to minimize average completion time is solving each instance with the fastest solver on it, and order each one by solving time. Alternatively, we can consider ordering the instances using our classifiers, but choosing always the fastest solver on each instance. Such an approach can help us understand how close our instance ordering approach is to being perfect. Similarly, we can consider ordering the instances by solving time but picking the solver decided by FS which can help determine how good our solver selection classifier is.

There are six obvious baseline management strategies. The first (Random) is to select the next instance to solve and its solver at random. The second and third strategies (Random + FS) and (Best Instance Selector + random) use classifiers for one selector while picking the other randomly; the Best Instance Selector corresponds to the instance selector of our best classifier-based approach. We also mix oracle and random approaches on instance and solver selection and consider Random + oracle and Oracle + Random. The last (Mistral) is to use a best on-average solver to solve each instance and heuristically order the instances. Note that this is not a portfolio approach, therefore it may leave some instances unsolved by the cut-off. We build such a baseline using the Mistral constraint solver, as Mistral has been the overall winner of the CSP Solver Competition for a number of years and has a parser for the competition problem format that we use in our experiments. Consequently, we construct a classifier called MISTRALC that classifies the instances into those that: can be solved quickly (within 10 s); that can be solved without time-out (between 10 and 1,799 s); or will time-out (1,800 s or more). We based these specific classes from the data presented in Fig. 8.1a. The other two solvers of the portfolio (Abscon and Choco) solve fewer problems than Mistral and on average their solving times are higher. Hence, we do not consider any baseline using these.

Another baseline is the worst-case scenario where the instances are ordered in completely the wrong way (from most difficult to the easiest) and always the worst solver is picked up. We have experimented with such a method many times and found it to be always much worse than Random. Given the unlikely nature of the worst-scenario, we do not consider this baseline in our experiments. We believe that this exclusion is safe. As explained later, every management strategy is executed several times and eventually the average of the results are reported. The analysis of the Random results have revealed that they cover a wide range, from being close to perfect to being totally naive. Hence, Random represents a more realistic scenario than the worst-case.

In the experiments, we adopt a simple random split validation approach to evaluating the quality of our classifiers for managing a portfolio of solvers. We use the

instances described in Sect. 8.4 to have a unified benchmark for all tests that we perform. Given a management strategy, we execute it (via simulation) 1,000 times. A single run consists in random split of the dataset into a testing set (1/5 of the instances) and a training set (the remaining 4/5). We use the random forest algorithm (WEKA) for learning the models. For every run, we compute the overall average solving times and report in Table 8.4 the average of the average solving times across 1000 runs. Differences in performance are tested for statistical significance using a Paired t-test at a 95 % confidence level.

Note that the training and the simulation are done in two separate phases. For every run, we have previously trained all the classifiers with the entire training set, hence, the accuracy of the classifiers should be similar to the one presented in Sect. 8.4. The simulator is implemented in Python using the SimPy Simulation Package[5] and the SciPy package[6] for the statistical computations.

*Results.* In Table 8.4, we present three categories of portfolio solvers: three oracle-based, nine classifier-based, and six baselines. They are ordered by average completion time. $\text{Oracle}_1$ is ranked first, of course, since it has full information about which solver solves which instance. The Random baseline is ranked last. The Mistral baseline selects as the next instance the one which it believes it can solve quickly (in less than 10 s). In case of time-out (set to 1,800 s), while Fs or random solver selector switches to another solver randomly, Mistral records 1,800 s as the solving time. Hence, Mistral results indicate only a lower bound.

$\text{Scheduler}_{11}$ to $\text{Scheduler}_4$ use the various classifiers presented in Sect. 8.3. In particular, $\text{Scheduler}_{11}$ to $\text{Scheduler}_{16}$ order the instances by first the 3C classifier and then the MaxC, AvgC, and Spread classifiers in six different ways. As noted previously, we expect that ordering the instances first by 3C is a good heuristic but it is not clear in which order to use the remaining classes afterwards. Given that the best result comes from $\text{Scheduler}_{11}$, we test the importance of the use of multiple classes via $\text{Scheduler}_2$ to $\text{Scheduler}_4$ by dropping one class at a time from the instance ordering of $\text{Scheduler}_{11}$. Even if we present here the results of only nine classifier-based management strategies, we have indeed experimented with all (subsets of ) possible orderings of instances by using 3C, MaxC, AvgC, and Spread and observed that $\text{Scheduler}_{11}$ is indeed the real winner overall. This is the reason why we use exactly this instance ordering in $\text{Oracle}_2$.

Although the best management strategy $\text{Scheduler}_{11}$ utilizes the 3C$\prec$ AvgC$\prec$ Spread$\prec$ MaxC instance ordering, which we later use as our Best Instance Selector in our baseline strategies, permuting the orders of the classes after 3C does not worsen the results in a statistically significant way, except in $\text{Scheduler}_{16}$. Instead, the results become significantly worse, as we start using one class less with respect to $\text{Scheduler}_{11}$ in the instance ordering. We therefore observe that our classifiers all contribute to the best management strategy.

[5] http://simpy.sourceforge.net/.

[6] http://www.scipy.org/.

**Table 8.4** Performance of classification-based portfolios on a single processor

| Scheduler category | Scheduler configuration | Selectors | | Average | | Average solver calls | | |
|---|---|---|---|---|---|---|---|---|
| | | Instance selector | Solver selector | | | Mistral | Choco | Abscon |
| Oracles | $\text{Oracle}_1$ | Oracle | Oracle | 916.35 | ○ | 513.74 | 29.21 | 116.05 |
| | $\text{Oracle}_2$ | 3C≺ MaxC≺ AvgC≺ Spread | oracle | 2714.83 | ○ | 513.74 | 29.21 | 116.05 |
| | $\text{Oracle}_3$ | Oracle | Fs | 3699.13 | ○ | 538.75 | 27.94 | 103.75 |
| Classifiers | $\text{Scheduler}_{11}$ | 3C≺ MaxC≺ AvgC≺ Spread | Fs | 6777.26 | | 538.74 | 27.89 | 103.73 |
| | $\text{Scheduler}_{12}$ | 3C≺ AvgC≺ Spread≺ MaxC | Fs | 6780.31 | | 538.74 | 27.93 | 103.75 |
| | $\text{Scheduler}_{13}$ | 3C≺ Spread≺ AvgC≺ MaxC | Fs | 6793.17 | | 538.74 | 27.94 | 103.76 |
| | $\text{Scheduler}_{14}$ | 3C≺ MaxC≺ Spread≺ AvgC | Fs | 6800.61 | | 538.75 | 27.87 | 103.74 |
| | $\text{Scheduler}_{15}$ | 3C≺ Spread≺ MaxC≺ AvgC | Fs | 6810.69 | | 538.74 | 27.90 | 103.81 |
| | $\text{Scheduler}_{16}$ | 3C≺ AvgC≺ MaxC≺ Spread | Fs | 6836.53 | • | 538.74 | 27.99 | 103.82 |
| | $\text{Scheduler}_2$ | 3C≺ MaxC≺ AvgC | Fs | 6963.35 | • | 538.73 | 27.95 | 103.74 |
| | $\text{Scheduler}_3$ | 3C≺ MaxC | Fs | 7325.42 | • | 538.73 | 28.01 | 103.72 |
| | $\text{Scheduler}_4$ | 3C | Fs | 8510.90 | • | 538.75 | 27.92 | 103.77 |
| Baselines | Mistral | MistralC | Mistral | ≥ 11495.77 | • | 659 | 0 | 0 |
| | Random+oracle | Random | Oracle | 12574.45 | • | 513.74 | 29.21 | 116.05 |
| | Random+Fs | Random | Fs | 23801.71 | • | 538.73 | 27.93 | 103.78 |
| | Best Inst. + Random | 3C≺ MaxC≺ AvgC≺ Spread | Random | 37350.05 | • | 262.72 | 246.01 | 254.46 |
| | Oracle+random | Oracle | Random | 44131.72 | • | 262.63 | 245.67 | 254.50 |
| | Random | Random | Random | 113236.64 | • | 262.43 | 245.50 | 255.08 |

○, • indicate statistically significant improvement or degradation over $\text{Scheduler}_{11}$

Scheduler$_{11}$ outperforms Mistral with statistical significance by at least a factor of two with Mistral being unable to solve 6.4 % of its instances by the cut-off. Comparing Scheduler$_{11}$ to all the baseline and oracle methods, we observe that the performance of Scheduler$_{11}$ is in general much closer to the oracles than it is to the baselines, both in terms of instance ordering alone, solver selection alone, and both. In particular, the prediction accuracy of FS is remarkable, compared to Mistral and random solver selection strategies. Clearly, a simple classifier-based approach to controlling a solver portfolio approach for CSPs is very promising.

## 8.6 Parallel Solver Portfolio

We now consider the use of classification to solve CSPs using a multiprocessor system. In this setting, more than one solver could be run in parallel for solving the same instance. It is therefore possible to try new strategies like the one of running all solvers in parallel for instances that are difficult for all the solvers but one, and run only the best solver for all the other instances. This strategy looks promising since it does not waste resources running all the solvers for all the instances and, at the same time, it minimizes the risk of choosing the wrong solver for some instances.

Minimizing the use or computational resources when a multiprocessor system is used could be very important. Since the more the resources are used the more expensive the computation becomes; in this section, we consider a metric function that takes into account also the number of processors. Specifically, instead of considering only the solving time, in order to compare different strategies we consider the product of the average solving time and the number of processors used by the system. Minimizing this function is equivalent to maximizing the efficiency of a system having a cost that increases linearly over the number of processors.

The experiments setting is the same as the one presented in Sect. 8.5. The only things that differ is that now a dispatching strategy consists of three selectors, namely instance, solver, and parallel. The first two, as described in Sect. 8.5, are used to sort the instances and choose the solver to use. The parallel selector instead can be viewed as a function that, given an instance, decides if the instance could be solved running all the solvers of the portfolio in parallel.

As a starting point we consider four simple parallel selectors:

Sel0 all the solvers are run in parallel for all the instances;
Sel1 selects to use all the solvers just for instances that are predicted to be easy for one solver but not for all by the 3C classifier;
Sel2 selects to use all the solvers just for instances that are predicted to have a big spread by the SPREAD classifier;
Sel3 selects to use all the solvers just for instances that are predicted to be easy for one solver but not for all and are predicted to have a big spread by the 3C and SPREAD classifiers, respectively.

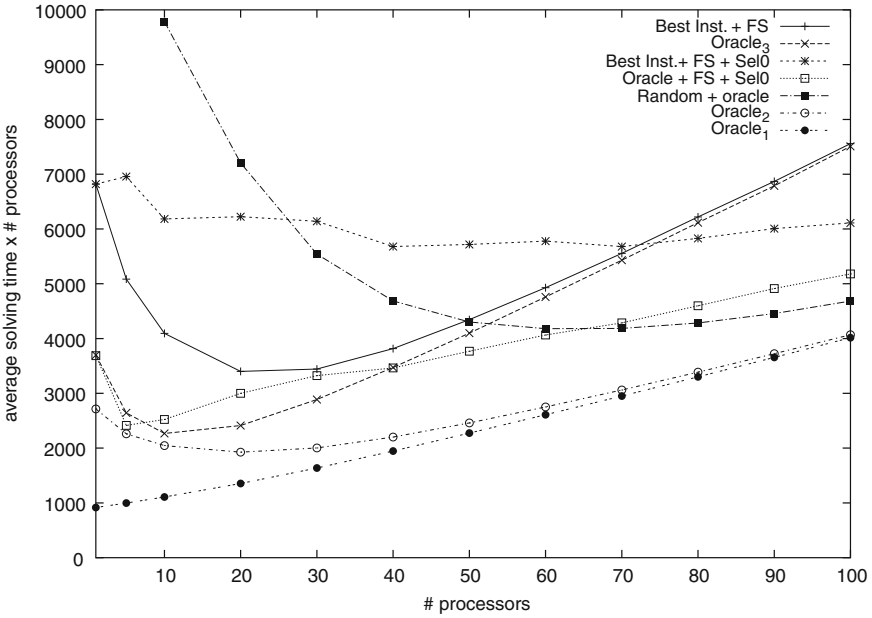



**Fig. 8.3** Performance of strategies exploiting multiprocessors systems

*Results.* Figure 8.3 depicts the cost of every strategy (i.e., the product of the average solving time and the number of processors used) as the number of processors increases. For presentation purposes, we report only a subset of the strategies presented in the Sect. 8.5 with two new additional strategies, namely (Best Inst.+ FS + Sel0) and (Oracle + FS + Sel0), that are obtained from the "Best Inst.+ FS" and "Oracle + FS," respectively, using the Sel0 as parallel selector (i.e., same strategies as before but now all the instances were solved using the solvers in parallel and not sequentially).

As can be seen from the plot, the more processors that are used the less important becomes the ordering of the instance selector. For instance, when more than 70 processors are used, the (Random + Oracle) strategy that sorts the instances randomly but chooses to use always the best solver is better than any strategy using the FS classifier. Intuitively, this can be explained by the fact that when more processors are used, more instances are solved in parallel and thus the mistake of scheduling a difficult instance first is less frequent.

Every strategy has a cost that steadily increases after a certain number of processors are used; this means that after a certain point the increase in the number of processors does not contribute significantly to the reduction in the average solving time. This result is predicted by the Amdahl's low [7] that states that the speedup of a program exploiting in parallel multiple computation nodes is limited by the time needed for the sequential fraction of the program. Notice however that here the test is conducted with a fixed number of instances and therefore, according to Gustafson's

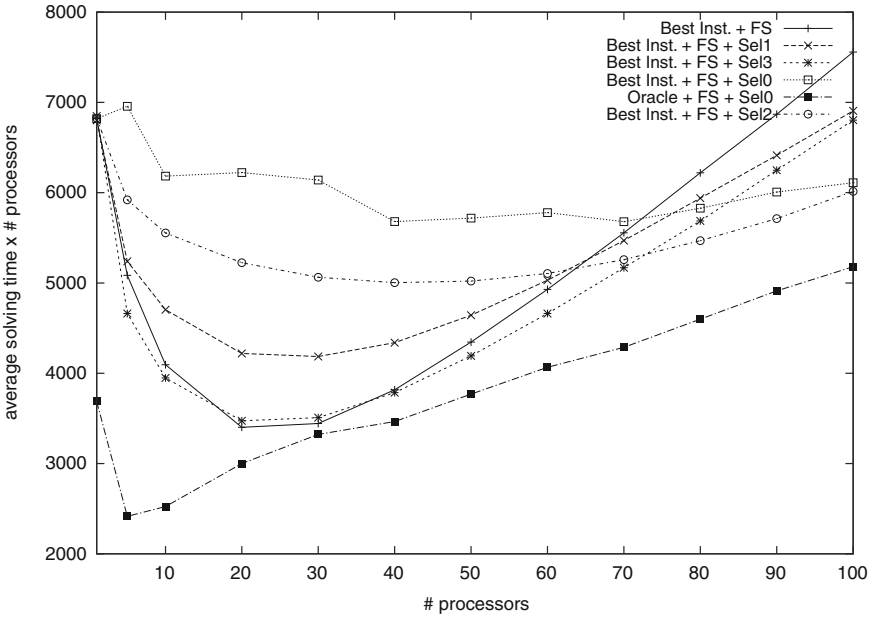



**Fig. 8.4** Performance of new strategies exploiting multiprocessors systems

law [62], in these settings it is still possible to obtain a linear speedup if the dimension of the problem varies. Hence, we expect that efficient systems to solve concurrently thousands of instances will need more than 100 processors.[7]

In Fig. 8.4, we report the strategies that enhance the (Best Inst.+ FS) strategy considering the four parallel selectors previously introduced. We report as baselines the (Best Inst.+ FS), (Best Inst. + FS + Sel0), and (Oracle + FS + Sel0) strategies already visualized in Fig. 8.3. From this plot, we notice that parallel selectors have a non-negligible effect on the cost of a strategy. Specifically, strategies having very selective parallel selectors (i.e., parallel selectors where only few instances are tried with all the solvers in parallel) have a smaller cost when systems have few processors, but their performance does not scale up. For instance, the strategy (Best Inst. + FS + Sel3), which has a more selective parallel selector than Sel2, performs initially better than the (Best Inst. + FS + Sel2) strategy until less than 70 processors are used. In particular, let us note that running all the solvers in parallel for all the instances is not a good choice for systems with a small number of processors but it can become a good strategy for systems having a lot of computation units. This could be very interesting since running all the solvers for a great number of instances allows the collection of more data that can be used to improve the classifiers accuracy.

[7] Unfortunately, the current datasets available for the CP community do not allow us to make simulations with thousands of instances.

These initial results suggest that a deeper and more exhaustive study of parallel selectors is needed in order to obtain a better and more efficient CSP portfolio-based parallel solver. Note that, here we assume that the cost of the system depends linearly on the number of processors. However, cloud providers usually make discounts if users are planning to use a lot of processors. In this kind of scenario, the metric function to evaluate the cost of the system is different and therefore it could be the case that the most efficient strategy is obtained with systems having more than 20 or 30 processors.

## 8.7 Related Work

The three closest approaches to solver portfolio management are CPHydra, SATzilla, and Acme. CPHydra, using a CBR system for configuring a set of solvers to maximize the chances of solving an instance in 1,800 s, was overall winner of the 2008 International CSP Solver Competition. Gebruers et al. [53] also use case-based reasoning to select solution strategies for constraint satisfaction. In contrast, SATzilla [131] relies on runtime prediction models to select the solver from its portfolio that (hopefully) has the fastest running time on a given problem instance. In the International SAT Competition 2009, SATzilla won all three major tracks of the competition. An extension of this work has focused on the design of solver portfolios [130]. The Acme system is a portfolio approach to solve quantified Boolean formulae, i.e., SAT instances with some universally quantified variables [106]. Streeter et al. [118] use optimization techniques to produce a schedule of solvers that should be executed in a specific order, for specific amounts of time, in order to maximize the probability of solving the given instance.

In [57], a classification-based algorithm selection for a specific CSP is studied. Given an instance of the bid evaluation problem (BEP), the purpose is to be able to decide a priori whether an Integer Programming (IP) solver, or a hybrid one between IP and CP (HCP) will be the best. Such a selection is done on the basis of the instance structure which is determined via (a subset of) 25 static features derived from the constraint graph [82]. These features are extracted on a set of training instances and the corresponding best approach is identified. The resulting data is then given to a classification algorithm that builds decision trees. Our purpose is not only to be able to predict the best solver for a given instance but also to choose the right instance in the right time so as to minimize the average finishing time of the set of instances. Consequently, we develop multiple classifiers and utilize them so as to predict their order of difficulty. Moreover, our features are general purpose and our approach works for any CSP in the XCSP format with any of its suitable solvers. Furthermore, we take into account as well dynamic features which provide complementary information.

Also related to our work is the instance-specific algorithm configuration tool ISAC [69]. Given a highly parameterized solver for a CSP instance, the purpose is to tune the parameters based on the characteristics of the instance. Again, such characteristics are determined via static features and extracted from the training instances. Then the

instances are clustered using the $g$-means algorithm, the best parameter tuning for each cluster is identified, and a distance threshold is computed which determines when a new instance will be considered as close enough to the cluster to be solved with its parameters. The fundamental difference with our approach is that instances that are likely to prefer the same solver are grouped with a clustering algorithm based on their features. We instead do not use any clustering algorithm. We create clusters ourselves according to the observed performance of the solvers on the instances and predict which cluster an instance belongs to based on its features using classification algorithms.

## 8.8 Summary

We have presented a novel approach to managing a portfolio for constraint solving. We proposed the use of classifier techniques as a basis for making high-level and qualitative statements about the solvability of CSP instances with respect to a given solver portfolio. We showed how these could then be used for solving a collection of problem instances. While this approach is conceptually very simple, we demonstrated that using classifiers to develop dispatching rules for a solver portfolio is very promising. The code for computing the CPHydra features and the simulator is available at www.cs.unibo.it/~jmauro/ictai_2011.html.

This is a first investigation toward the ambitious goal of developing an online service-based portfolio CSP solver as described in Chap. 7.

As part of the future work, we will investigate the benefit of using automatic algorithm tuning tools like GGA [8] and ParamILS [65] to train a larger portfolio of solvers. It has been observed in ISAC and Hydra that additional performance benefits can be achieved with solvers that have been expressly tuned for a particular subset of problem types.

Finally, we would like to exploit the solving statistics (e.g., solving times, memory consumption) obtained at runtime to improve on the fly the predictions of the models. This goal has been already considered for the QSAT domain [107]. We plan to follow similar ideas using online machine learning techniques [127].

# Chapter 9
# Broadcast Messages in Jolie

Service-Oriented Computing (SOC), presented in Sect. 3.5 is a paradigm for programming distributed applications by means of the composition of services. Services are autonomous, self-descriptive computational entities that can be dynamically discovered and composed in order to build more complex functionalities. The resulting systems, called Service-Oriented Architectures (SOA), have a wide diffusion. In an SOA services are loosely coupled, i.e., they stress a minimality on the dependencies that each service has with respect to the others, and can be stateful; this last point is the case of orchestrators which maintain a state for each created session. Usually, in a stateful service a session is created at the first client invocation. But, differently from the object-oriented approach, SOC does not guarantee references for identifying the new session. Thus a fundamental aspect which affects the efficiency and the performance of SOAs is the mechanism which allows to manage sessions. In fact, in a typical pattern of interaction, a service may manage many different sessions, corresponding to different clients. Since communications are usually supported with stateless protocols (e.g., SOAP on HTTP), when a service receives a message from a client $C$ the system must be able to identify which is the session corresponding to $C$ and that, therefore, must receive the message. In other words, sessions usually need to be accessed only by those invokes (messages) which hold some specific rights.

A relevant mechanism for solving this problem, first introduced by BPEL [5] and then used in JOLIE [95, 96], COWS [80], and in other languages, is that based on *correlation sets*. Intuitively, a correlation set is a set of variables whose values allow to distinguish sessions initiated by different clients. More precisely, both the sessions and the incoming messages contain some specific "correlation values" defining the variables in the correlation set. When a message $m$ arrives, it is routed to the session which has the same values as $m$ for the correlation variables.

As a simple example of correlation set, consider the case of a service $S$ used for buying goods. Suppose that $S$ handles all the communication of a specific customer using a unique session, while different customers have different sessions. Assuming that a customer is uniquely determined by her name and surname, we can use a

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_9, © Atlantis Press and the authors 2014

correlation set consisting of the two variables $name$ and $surname$ for determining the customer's session. Now, let us suppose that $S$ can receive the following three types of messages (with the obvious meaning):

- $buy(name, surname, product\_id)$;
- $delete\_order(name, surname, product\_id)$;
- $pay(name, surname, product\_id, credit\_card\_info)$.

When a customer, say John Smith, wants to buy product 1 he can send a message of the form $buy(John, Smith, 1)$. When this message is received the service checks whether there is a session that correlates with it, i.e., whether there exists a session whose variables $name$ and $surname$ are, respectively, instantiated to the values $John$ and $Smith$. If this is the case message $m$ is assigned to such session. On the other hand, if John Smith is a new customer and no session correlates with $m$ then the message is not delivered (note however, that in this case a new session could be created which correlates with the message, see for example [5, 9, 60]).

The BPEL and Jolie languages are currently allowing the use of messages whose target is only one session. However, there are a lot of scenarios where being able to send a broadcast message to more than one session could be useful. For instance, broadcast messages can be used to define the message exchange between the workers and the distributor in the CiC framework defined in Chap. 7. Another case where broadcast messages could be useful is a cloud environment where every user can start, control, and terminate a virtual machine on the cloud (a framework similar for instance to Amazon EC2). Let us suppose that we would like a unique entry point to this system and this entry point is a service that can receive and send messages to the users and the administrators of the cloud. We could consider to have a session for every virtual machine and control the virtual machine through this session. The key to identify a session can be the union of the following fields:

- the name, surname, and date of birth of the user (we assume that these values univocally determine the user);
- the kind of virtualized operating system (i.e, Ubuntu, Windows, …);
- the version of the operating system;
- the priority of the virtual machine (high, medium, low).

Having this key a user (say John Smith born on the 1st of Jan 1970) can start a Windows 7 machine with low priority sending for instance a message like *start(John, Smith, 19700101, windows, 7, low)*. Later, he can control and terminate the session (and therefore the virtual machine) simply sending messages like *execute* or *terminate* specifying every time all the fields of the key.

On the other hand, suppose now that an administrator wants to apply a patch to all the Windows virtual machines. Without a broadcast primitive he/she should retrieve all the keys of sessions controlling a Windows machine and later send them the message that triggers the application of the patch. From the programmer point of view, this usually involves the definition of a session or service that keeps the log of all the sessions. This session/service often slows down the performances due to the

creation or deletion of new sessions. On the other hand having a broadcast primitive an administrator could send:

- a message like *get_location()* that will be sent to every session for asking to the session which hardware machine is used to run the virtual machine;
- a message like *patch(operating_system, operating_system_version, …)* to patch all the virtual machines with a certain operating system and version;
- a messages like *terminate(name, surname, birthday_date)* that can terminate all the virtual machines belonging to a user;
- messages like *stop(priority)* or *stop(operating_system, priority)* can be used to stop every virtual machine having a specific priority or operating system + priority.

These are only few examples of the use of broadcast primitives. Another important application for these messages is for the implementation of a publish/subscribe pattern: This is a messaging pattern where senders (publishers) of messages do not send the messages directly to specific receivers (subscribers). The messages are instead divided into classes and the subscribers subscribe for the reception of messages of a given class. The system is responsible for sending every message belonging to a certain class to every subscriber that has subscribed for that class. Publisher may not know who are the subscribers and vice versa.

This pattern can be easily implemented using broadcast and a service having a correlation set that contains the class identifier. Whenever a subscriber subscribes for a class, a new session responsible for forwarding of the message is created. The publisher now can send a broadcast message specifying in the message its class. The correlation mechanism will check this value and route the message to every session that has subscribed for that class. The session can later forward the message to the real subscriber.

In this chapter, we present a data structure and an implementation of the correlation mechanism that supports the broadcast primitive without degrading the performances of the correlation of normal messages.

The operations that a correlation mechanism has to support can be seen as the select, insert, and delete operations of a relational database, where every tuple of the relation is a session. The correlation set is a key of a relation. When a normal message arrives, it always contains a key that determines the target session. In the database analogy, the correlation operation is then a "select" operation, and in the case of normal messages the (complete) key is used to retrieve the target session. On the contrary, a broadcast message specifies only part of the key, indeed its target is potentially a set of sessions. Continuing in the database analogy, the broadcast operation can be efficiently implemented by adding an index for every type of broadcast messages. However, since increasing the number of indexes decrease the performances of the insert and delete queries (i.e., creation and deletion of sessions), the less indexes we have the better it is. We will then define a solution that uses the minimal number of indexes needed to correlate the messages to the right sessions. The indexes will be implemented using radix trees.

We would like to underline that in this work we have taken as a starting point the correlation mechanism of Jolie . We made this choice because Jolie was the language

we chose to develop the framework presented in Chap. 7 and because we find that Jolie correlation mechanism is more flexible than the BPEL one. For instance, Jolie correlation variables are normal variables and not a late-bound constant like in BPEL. While in BPEL the values of a correlation set are defined only by a specially marked send or receive message and once defined they cannot change, in Jolie the programmer can decide to instantiate or change the values of a correlation set at runtime. In BPEL, all the fields (correlation proprieties or correlation tokens) of a message key should be always defined. Jolie instead allows partially defined keys. This flexibility comes with a price: the implementation of the search of a correlating session is linear with respect to the number of session while in BPEL it is constant (usually hash table is used).

The correlation mechanism can be seen as a special case of the well-known content-based publish/subscribe mechanism [98]. Indeed the correlation mechanism can be seen as a simpler content-based publish/subscribe mechanism where messages are notifications, sessions are subscriptions, and correlation variables are attributes. The correlation mechanism exploits, however, two constraints that usually a content-based publish/subscribe mechanism does not have. In correlation, few attributes need to be considered and only equality predicates are used to compare the attributes. Hence, this work could be considered as an improvement over publish/subscribe algorithms such as [30, 37] for scenarios where the previous two constraints hold.

After having provided some background in Sect. 9.1, we explain the idea of the algorithm in Sect. 9.2. In Sect. 9.3, we show how the data structure is created and used, while in Sect. 9.4, we prove the correctness of the algorithm and we perform some complexity analysis. Finally, Sect. 9.5 presents a summary.

## 9.1 Background

In this section, we formally define the main concepts that we will use in the rest of this chapter. A correlation set, c-set for short, can be seen as a key that can be used to retrieve a session. For our purposes a c-set can be seen as a set of variables names (in BPEL these correspond to c-set proprieties) that can assume values in a domain. To simplify the notation, we assume that the variables of a c-set can assume values in the domain $D$ defined as the set of strings on a given signature.

**Definition 9.1 (c-set).** Given a service $S$, a correlation set for $S$ is a finite set of variables names. When these variables are defined, their values uniquely identify a session of $S$.

Sessions may define the variables of a c-set. The definition of variables belonging to a c-set is captured with the following definition.

**Definition 9.2 (c-instance).** Given a c-set $c$, we say that a c-instance for $c$ is a total function that maps every variable of $c$ to a value in $D$.

We will say that a session $s$ has a c-instance $\varphi$ if for every variable $v$ in $c$ the variable $v$ has been assigned and its value is $\varphi(v)$.

Services, especially those having multi-party sessions, may need more than one c-set because the users may need to use different keys to identify a session. These services, also known as multi correlation services, do not require to have a c-instance for every c-set. However, since c-sets are used to identify a session we require that a session must have at least a c-instance. Moreover, we do not allow the starting of a session having the same c-instance of another existing session.

Every message that is exchanged will contain some arguments associated to a c-set. Usually, these arguments are called correlation tokens or correlation values and are used to find the recipient of the message. BPEL and other service engines allow the use of potentially one correlation token (c-token for short) for every c-set of the service. For example, a multi-party session can be initialized submitting a message having as correlation tokens the values for all the c-sets of the service. In this work, instead we will consider messages having only one c-token. This restriction is, however, insignificant since the behavior that is caused by the exchange of messages with more than one c-token can be easily simulated in our framework. This is due to the fact that differently from BPEL we do not need the exchange of a message to change the value of a correlation variable.

Formally, we can define a c-token in the following way.

**Definition 9.3 (c-token).** Given a message $m$ a c-token is a pair $(c, \varphi)$ where

- $c$ is a c-set containing the variables used to specify the message recipients
- if $m$ is a normal message then $\varphi$ is a total function that maps a variable of $c$ into a value in $D$
- if $m$ is a broadcast message then $\varphi$ is a partial function that maps a variable of $c$ into a value in $D$. Moreover $\varphi$ is not total.

For instance, the service for buying goods has only one c-set $c = \{name, surname\}$ and the c-instance of John's session is the function $\varphi$ s.t. $\varphi(name) = John$ and $\varphi(surname) = Smith$. The message $buy(John, Smith, 1)$ has instead as c-token the couple $(c, \varphi)$. If we want to send a message $m$ to every person named John for wishing him a happy name day, we can use a broadcast message whose c-token will be the couple $(c, \varphi')$ where $\varphi'(name) = John$ and $\varphi(surname)$ is not defined.

As it can be seen in the previous definition, the introduction of the broadcast primitive allows the user to not define all the variables of a c-set. Normal messages, like c-instances, need to define all the variables of a c-set because they need to identify their (unique) target session. On the other hand, broadcast messages can specify only a part of the key, indeed their target can be a set of sessions. Note that, in case of multicorrelation services, the c-token definition does not allow to consider part of two different keys to determine the targets of a broadcast message. We do not allow this possibility since we haven't find a significant example that justifies this increased power. However, we could easily extend our framework to treat also this case. Now we can formally define when a message correlates with a session.

Intuitively, a message correlates with a session when the values of the correlation token match the c-instance of a session. In the following $\varphi_m(v) \uparrow$ denotes that $\varphi_m$ is not defined in $v$.

**Definition 9.4** (**Correlation**). Given a service $S$, a session $s$ and a message $m$ with c-token $(c_m, \varphi_m)$ we will say that $s$ correlates with $m$ iff $s$ has a c-instance $\varphi$ for the c-set $c_m$ and $\forall v \in c_m.\ \varphi_m(v) = \varphi(v) \vee \varphi_m(v) \uparrow$.

## 9.2 The Idea

As we have discussed above, the current mechanisms for assigning a message to the correct session does not support the possibility of identifying a set of sessions. A naive implementation for the support of broadcast messages would use an associative array for every c-set variable. However, if this solution is used, for finding the targets of a broadcast message, we have to compute a set intersection whose complexity depends on the number of sessions. Another naive solution is using an associative arrays for every subsets of correlation variables that can be used in a broadcast message. If we consider a c-set with $n$ variables, this means that for the support of the broadcast primitive we could have $2^n - 1$ associative arrays, since with $n$ variables we can use up to $2^n - 1$ different kind of broadcast messages (one for every subset of the c-set variables). Inspired by [114], our key idea in order to improve the complexity of message assignment is to use radix trees to memorize the c-instances of all the sessions and, therefore, for routing messages to the correct session. In this section, we will explain intuitively the idea, while its formalization and complexity analysis are contained in the next sections.

A trie, or a prefix tree, is an ordered tree for storing strings, in which there is one node for every common prefix. Edges are labeled with characters, while the strings are stored in extra leaf nodes. Tries are extremely useful for constructing associative arrays with keys that can be expressed as strings, since the time complexity of retrieving the element with a given key is linear time in the length of the key. In fact, looking up for a key of length $k$ consists in following a path in the trie, from the root to a leaf, guided by the characters in the key. A radix tree (or Patricia tree, [97]) is essentially a compact representation of a trie in which any node that has no siblings is merged with its parent (so, each internal node has at least two children). Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters. This makes radix tree more efficient than tries for storing sets of strings (keys) that share long prefixes. The operations of lookup (to determine whether a string is in the set represented by a radix tree), insert (of a string in the tree), and delete (of a string from the tree) have all worst-case complexity of $O(l)$, where $l$ is the maximal length of the strings in the set.

Intuitively our idea is to use radix trees to map incoming messages to sessions, by using the values of the c-set variables as keys. In other words, the session pointers can be seen as elements stored in an associative array, while the values of the variables of

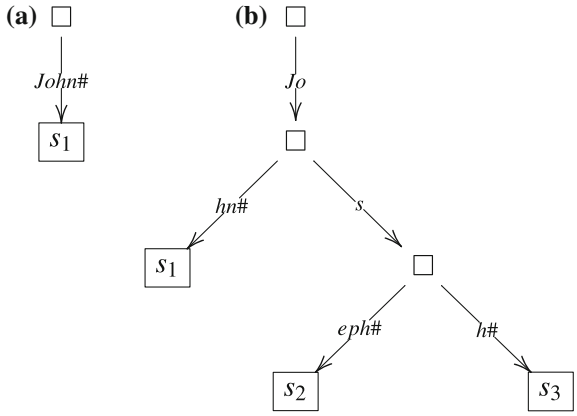



**Fig. 9.1** Example of radix trees

the c-sets, conveniently organized as strings, are the keys. Our radix trees implement such a structure by memorizing the values of the c-set variables which appear in the existing sessions. In particular, since every broadcast message can define only part of the c-set variables, to be able to process every message we could use a radix tree for every subset of the c-set variables. This however is not an optimal solution. For example, if a service has two c-set variables *name* and *surname*, we could receive the following kind of messages

1. broadcast messages s.t. their c-tokens do not define any variable
2. broadcast messages s.t. their c-tokens define only the field *name*
3. broadcast messages s.t. their c-tokens define only the field *surname*
4. normal messages s.t. their c-tokens define both *name* and *surname*

With the naive approach we need to use four associative arrays (one for every message type). Using radix trees is, however, possible to use a unique radix tree for 1st, 2nd, and 4th types since the c-tokens of the 1st and 2nd kind of messages can be considered as prefix of the c-tokens of the 4th type of messages. For the message of the 3rd type, instead, we have to use a different radix tree, since in this case the c-tokens are not a prefix of those for the 4th type of messages. So, it is sufficient to use two radix trees to cover all the possible cases.

To better explain the idea, let us consider some more examples. In the following, we use a special character, denoted by # and not used elsewhere, to denote in a string the termination of the values of c-set variables.

We first consider a unique c-set variable with only one field: *name*. When there exist no session for such a variable, we have a radix tree consisting of the only root (recall that in radix trees the root is associated with the empty string). We represent such a radix tree as a $\square$. If now a session $s_1$ is created which is identified by the value *John* for the c-set variable *name* then the radix tree became as the one depicted in Fig. 9.1a. The value *John* allows to reach $s_1$ by an (obvious) lookup in the tree.

Next, assume that two more sessions are created: a session $s_2$, which is identified by the value *Joseph* for the variable *name* and a session $s_3$ which is identified by

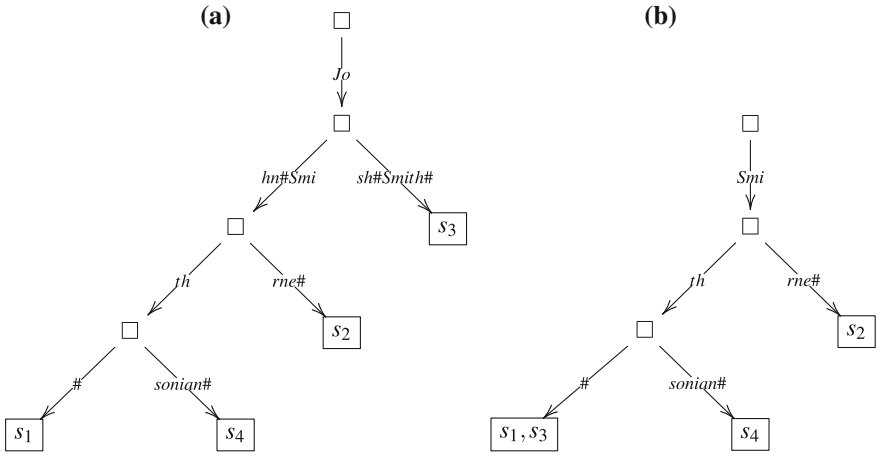



**Fig. 9.2** Example or radix trees for c-set with two variables

$Josh$. The radix tree we obtain is the one depicted in Fig. 9.1b. Notice that the longest common prefixes of the three key values are associated to edges of the tree. When a message arrives, the value that it carries for the *name* variable allows one to select a root-leaf path in the tree, so reaching the correct session.

Assume now that our correlation set is composed by the two variables *name* and *surname* and consider four sessions $s_1 - -s_4$ identified as follows by the values of the c-set variables:

$$s_1 : name = John, surname = Smith;\ s_2 : name = John, surname = Smirne$$
$$s_3 : name = Josh, surname = Smith;\ s_4 : name = John, surname = Smithson$$

Correspondingly, we have the radix tree depicted in Fig. 9.2a. In this case, as mentioned before, we need more that one radix tree to store the values of c-sets variables of the sessions. This is because in a broadcast message the value of some c-set variables could be not specified. For example, in the case above, let us consider a broadcast message which contains the token $Smith$ for $surname$ and no token for $name$. If we have only a radix tree like the one depicted in Fig. 9.2a, we cannot find with a lookup which session correlates with it. This is due to the fact that the first part of the key of the radix tree is the value of the variable $name$. Hence, we need an additional radix tree like the one depicted in Fig. 9.2b that can be used to retrieve sessions for messages that do not define the variable $name$.

It is easy to see that these two radix trees allow to cover all the possible cases. First, consider what happens if we receive a message $m$ where $name = John$ and $surname = Smith$, hence we consider the string $John\#Smith\#$. In this case, by using the Fig. 9.2a radix tree, we see that the message $m$ will be assigned to $s_1$, since this is the session which correlates with $m$. However, note that this first tree covers also the case in which no value for surname is provided by the message, hence we

do not need a further radix tree to keep only the sessions that define only the variable $name$. For example, if we receive a message $m$ with $name = John$, that is we consider the string $John\#$, then the Fig. 9.2a radix tree shows that $m$ correlates to the sessions $s_1, s_2, s_4$.

On the other hand, if we receive a broadcast message $m'$ where $name$ is not defined and $surname = Smith$, we will use the Fig. 9.2b radix tree (with the string $Smith\#$) to find that the session correlating with $m'$ are $s_1, s_3$.

## 9.3 Building the Radix Trees

As previously discussed, with our approach every c-set of the service has a group of radix trees that can be used for checking the correlation of a message to a session. We have also shown that, if we assume that the c-set has $n$ variables, one does not need to consider $2^n$ different radix trees, because a radix tree for a sequence of variables covers also all the cases given by the prefixes of such a sequence.

In this section, we provide an algorithm that, given a c-set with $n$ variables, in the worst case constructs a set containing $\binom{n}{\lceil n/2 \rceil} (= \frac{n!}{\lceil n/2 \rceil! \, \lfloor n/2 \rfloor!})$ radix trees. In the next section, we will prove that such set allows us to route all the possible messages to a service. We also prove that this set is minimal, in the sense that any other set of radix trees which allow to route correctly all the messages has at least the same cardinality. So, our algorithm cannot be improved with respect to the number of radix trees generated.

In the following, we assume that the c-set $c$ has $n$ variables and the set $V$ contains all and only these variables. We denote by $seq_i$ a sequence $x_1, \ldots, x_{h_i}$ of variables of $c$. Given a list of sequences of variables $seq_1, \ldots, seq_m$ such that $seq_i$ is a prefix of $seq_{i+1}$, for $i \in [1, m-1]$, we use the notation $RT(seq_1, \ldots, seq_m)$ to indicate any radix tree whose keys are strings of the form $d_1\# \ldots \#d_{h_i}\#$ where $d_j = \varphi(x_j)$, for $j \in [1, h_i]$, and for some c-set instance $\varphi$. In other words, $RT(seq_1, \ldots, seq_m)$ is a kind of schema which can be instantiated by considering the values of the variables for one specific sequence $seq_i$, with $i \in [1, m]$ (and using # as separator of values), to obtain a specific concrete radix tree. As previously discussed, a radix tree (described by) $RT(seq_1, \ldots, seq_m)$ allows us to check the existence of a session defining all the variables in one of the sequences $seq_i$. For example, the radix tree in Fig. 9.2a can be denoted by $RT(\langle\rangle, \langle name\rangle, \langle name, surname\rangle)$ while the radix tree in Fig. 9.2b is denoted by $RT(\langle surname\rangle)$.[1] By using this notation our problem can be stated as follows: we need to find the minimum number $h$ of radix trees schemas $RT_1(seq_{1,1}, \ldots seq_{1,l_1}), \ldots, RT_h(seq_{h,1}, \ldots seq_{h,l_h})$ such that, for each set $X \subseteq V$, there exists a sequence $seq_{k,o}$ that contains all and only the variables in $X$.

We find convenient to formulate this problem in terms of a graph representation. Indeed, given a set of variables $V$, we can create a labeled direct graph $G(V)$ where:

[1] Note that the order of the cset variables is important and therefore for instance $RT(\langle name, surname\rangle) \neq RT(\langle surname, name\rangle)$.

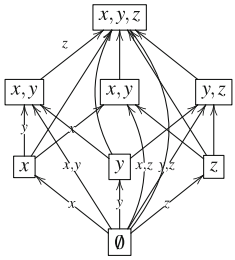



**Fig. 9.3** Example of the graph obtained with three variables: $x$, $y$, $z$ (note that for convenience only few arc labels are reported)

- the nodes are (labeled by) elements in $\mathcal{P}(V)$. Intuitively, we will consider all the set of variables that can be defined by a c-token;
- there is an arc from $u$ to $v$ if $u \subset v$;
- the arc $(u, v)$ is labeled with the variables $v \setminus u$ (where $\setminus$ denotes set difference).

For example, in Fig. 9.3, we see the graph constructed by considering the three variables $x$, $y$, and $z$ where we can receive all the possible seven broadcast messages. A path on this graph corresponds to a radix tree schema (see Definition 9.5). Hence, with this graph representation our problem can be stated as follows: we have to find the minimum number of paths that cover all the nodes of the graph where, as usual, we say that a path

$$u_1 \xrightarrow{x_1} u_2 \xrightarrow{x_2} \ldots \xrightarrow{x_n} u_{n+1}$$

covers the nodes $u_1, \ldots, u_{n+1}$.

The algorithm that produces this minimum number of paths is Algorithm 1 and its intuition is the following. Consider the graph $G(V)$ associated to a c-set $V$, as explained above. We first partition all the nodes of $G(V)$ into levels according to the number of variables of the nodes, so level $i$ contains all the nodes that have exactly $i$ variables. Then starting from the lowest levels (i.e., level 0 and 1) we consider two next levels at a time, say level $i$ and $i + 1$. These two levels are seen as a bipartite graph where the nodes of each level form an independent set. We then use a maximum bipartite matching algorithm for selecting a set of arcs between the nodes of these two levels. Next, we repeat the same procedure with levels $i + 1$ and $i + 2$, and we continue until we reach the level $n$. At this point we take the graph $G'(V)$ obtained by considering all the nodes in the original graph $G(V)$ and only the edges which have been selected by the matching algorithm. As we prove in the next section, the maximal paths[2] on the graph $G'(V)$ form a minimum set of paths covering all the nodes of $P$.

Before providing the algorithm we need to introduce some notation. We assume that each node is (labeled by) an element of $\mathcal{P}(V)$ ($n = |V|$), as mentioned above and we denote by $level_V(i)$ the set of nodes in the $i$-th level, i.e., the set of elements in

[2] A maximal path is a path that cannot be a proper part of another path.

$\mathcal{P}(V)$ which have cardinality $i$. Moreover, $graph(A, B)$ denotes the bipartite direct graph $(A \cup B, E)$ where $(u, v) \in E$ iff $u \subset v$. Finally, $maximal\_matching(G)$ is one of the maximal matchings of the bipartite graph $G$ chosen in a nondeterministically way. Algorithm 1 takes as input the set $P \subseteq \mathcal{P}(V)$ and returns the graph containing a minimum set of paths covering all the nodes of $P$. Once we have obtained a graph by using the Algorithm 1, it is possible to compute the radix trees by simply finding all the maximal paths, as shown below.

**Algorithm 1** $radix\_trees(P)$

1: $i = 0$
2: $V = level_P(i)$
3: $M = \emptyset$
4: **while** $(i < n)$ **do**
5: $\quad i = i + 1$
6: $\quad V' = level_P(i)$
7: $\quad G = graph(V, V')$
8: $\quad M' = maximal\_matching(G)$
9: $\quad V = V - \{v \mid (v, x)$ is an edge in $M'$, for some x$\}$
10: $\quad V = V \cup V'$
11: $\quad M = M \cup M'$
12: **end while**
13: **return** $(P, M)$

**Definition 9.5.** Given $P \subseteq \mathcal{P}(V)$ we say that a radix tree schema $RT(u'_1, u'_2 \ldots, u'_m)$ is produced by the algorithm $radix\_tree(P)$ if

$$u_1 \xrightarrow{x_1} u_2 \xrightarrow{x_2} \ldots \xrightarrow{x_m} u_{m+1}$$

is a maximal path in the graph $G = radix\_tree(P)$ and

- $u'_i$ is a sequence of all the variables in the set $u_i$, for each $i \in [1, m]$;
- $u'_i$ is a prefix of $u'_{i+1}$, for each $i \in [1, m-1]$.

We now consider an example of application of the previous algorithm to the graph in Fig. 9.3. In Fig. 9.4, we have reported the three steps denoting by $\Rightarrow$ the arcs selected by the maximal matching algorithm (i.e., arcs in $M$) while $\rightarrow$ indicates the arcs considered by the maximal matching algorithm (i.e., arcs in G, line 7). The nodes in frame are the nodes that are used for computing the maximal matching (i.e., the nodes in $V$ and in $level_P(i)$), while nodes in dotted frame are the nodes already processed (not considered by the matching algorithm and deleted from $V$, line 9).

From the final graph (Fig. 9.4d), we can compute the radix trees schemas by taking the maximal paths:

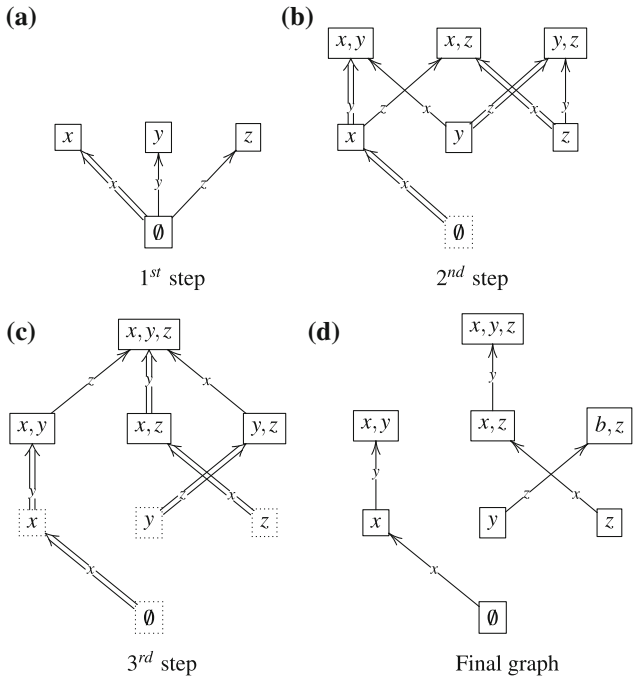
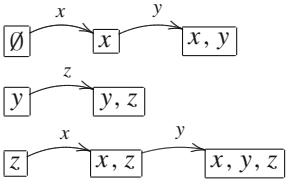



**Fig. 9.4** Example of execution of Algorithm 1 with three variables

The first path corresponds to the radix tree schema $RT(\langle\rangle, \langle x\rangle, \langle x, y\rangle)$ while the other two correspond to $RT(\langle y\rangle, \langle y, z\rangle)$ and $RT(\langle z\rangle, \langle z, x\rangle, \langle z, x, y\rangle)$, respectively.



## *9.3.1 Using Radix Trees*

Once we have created the radix tree schemas by using our algorithm, we need some operations for inserting and removing values from them, thus creating the concrete radix trees to be used for correlating messages and sessions. Moreover, we need to define a lookup operation, that, given a message, allows us to use the (concrete) radix tree to find all the correlating sessions. To this aim, we first introduce the three operations described below. Here and in the following, unless differently specified, with "radix tree" we mean a concrete radix tree, containing values for keys and whose

leafs contain (pointers to) sessions. Moreover, we assume w.l.o.g. that the service has a unique c-set and therefore only one group of radix trees. If the service has more than one c-set the following considerations should be applied to every c-set.

- $RT.add(s)$ is the operation for adding to the radix tree $RT$ the session $s$;
- $RT.del(s)$ is the dual operation that deletes the session $s$ in $RT$;
- $RT.find(m)$ returns all the sessions which correlate with $m$. If no sessions in $RT$ correlates with $m$ then the *null* pointer is returned.

Assuming that RT belongs to the radix tree schema $RT(seq_1, \ldots, seq_k)$, when $RT.add(s)$ is invoked $s$ is added to the radix tree $RT$ using as key the string $\varphi_s(x_1)\# \ldots \#\varphi_s(x_l)\#$ where $\langle x_1, \ldots, x_l \rangle = seq_k$ and $\varphi_s$ is the c-instance for $s$. In a similar way $RT.del(s)$ deletes from $RT$ the session pointer to $s$.

If $\langle x_1, \ldots, x_l \rangle$ is the sequence of all the variable defined by the c-token $\varphi$ of a message $m$, the operation $RT.find(m)$ can be applied iff there exists a sequence $seq_i = \langle x_1, \ldots, x_l \rangle$. In this case, this operation returns all the sessions whose keys have as prefix the string $\varphi(x_1)\# \ldots \#\varphi(x_l)\#$.

Using these basic operations, we can now define the operations which manage the set of radix trees produced by our algorithm. More precisely, we assume that the set of radix tree schemas produced by the algorithm has been instantiated to a set of (concrete) radix trees. Then this set is managed by the following three operations: $find\_session(m)$ (for finding a session that correlates with a message $m$); $add\_session(s)$ (for adding the session $s$); $del\_session(s)$ (for deleting a session $s$). The definition of the $add\_session(s)$ and $del\_session(s)$ is obvious since the only thing to do is to execute $RT.add(s)$ and $RT.del(s)$ for every radix tree $RT$. The $find\_session(m)$ instead first have to select a specific $RT$ based on the variables defined by the c-token of m and later return the $RT.find(m)$ result.

## 9.4 Correctness and Complexity Analysis

In this section, we prove the correctness of Algorithm 1 and we discuss the complexity of correlation mechanism based on it. In particular, we show that it produces the minimal number of radix trees needed to guarantee correctness. In the following, as usual, we assume that $V$ is the set of variables of a c-set and that $n = |V|$.

First of all, we show that Algorithm 1 produces a number of radix trees much smaller than $2^n$. With a slight abuse of notation, when no ambiguity arises, we indicate by $radix\_trees(P)$ both the graph produced by the algorithm, with input $P$, the radix tree schemas obtained from this graph according to Definition 9.5, and the concrete radix tree obtained from the schemas as described at the end of previous section. All the proofs of the theorems are reported in Appendix A.

**Theorem 9.1.** *If $W \subseteq \mathcal{P}(V)$ the result of $radix\_trees(W)$ is a graph containing at most $\binom{n}{\lceil n/2 \rceil}$ maximal paths. Hence the algorithm produces at most $\binom{n}{\lceil n/2 \rceil}$ radix trees schemas.*

Next, we show that the algorithm is correct, that is, the number of radix trees produced is sufficient to check correlation.

**Theorem 9.2.** *Let $m$ be a message and $V_1, \ldots, V_k$ be all the subsets of c-set variables that are defined by all the possible c-tokens. Then there exists a radix tree schema produced by $radix\_trees(\{V_1, \ldots, V_k\})$ which allows us to check if the message correlates with a session.*

Finally, we show that the number of radix trees produced by the algorithm is the minimal one which guarantees correctness.

**Theorem 9.3.** *The graph produced by $radix\_trees(P)$ contains the minimal number of maximal paths covering all the nodes in $P$.*

As an obvious consequence of previous theorem, we obtain that if we consider less radix trees than those produced by Algorithm 1 we cannot establish correctly correlation for some kind of messages. Thus, our algorithm cannot be improved with respect to the number of radix trees that one can use to solve this problem.

The complexity of Algorithm 1 is polynomial on the size of $P$. As for the complexity of the operations described in Sect. 9.3.1, assuming that $l$ is the maximum length of a c-set value and $k$ is the number of the sessions that correlate with a message $m$, the (time) complexity of $find\_session(m)$, is $O(n + knl) = O(knl)$. For normal (i.e., non broadcast) messages the complexity of $find\_session(m)$ reduces to $O(nl)$. On the other hand, the (time) complexity of $add\_session(s)$ and $del\_session(s)$ is $O(\binom{n}{\lceil n/2 \rceil} l)$ (for more details see [86]). We would like to underline that, in practice, the number of the c-set variables which are used is very small (less or equal to 5) so, in practice, the complexity of our operations is constant.

Let us now consider the complexity of the fundamental operations of the correlation mechanism as described in Sect. 9.3.1:

- For $find\_session(m)$, it is necessary to retrieve the right radix tree where the find operation can be applied. This operation can be done simply checking the number of the variables defined by the c-token of $m$ and therefore has complexity $O(n)$. If $k$ is the number of the sessions that correlate with $m$ ($k = 1$ if no session correlates with $m$) we have that the cost of $find\_session(m)$ is $O(n + knl) = O(knl)$. Note that for normal messages the cost of $find\_session(m)$ is $O(nl)$ that is also the complexity of a correlation mechanism implemented using a perfect hash table with a key of length $O(nl)$.
- For $add\_session(s)$ we need to add a session pointer to every radix tree. Since adding a new element in a radix tree has cost $O(nl)$ and for Algorithm 1 the number of radix trees is less or equal to $\binom{n}{\lceil n/2 \rceil}$ we have that the cost of $add\_session(s)$ is $O(\binom{n}{\lceil n/2 \rceil} l)$.
- Since adding or removing an element to a radix tree has the same cost of deleting one, also $del\_session(s)$ has complexity $O(\binom{n}{\lceil n/2 \rceil} l)$.

## 9.5 Summary

In this chapter, we have proposed a data structure, based on radix trees, for managing a correlation mechanism which supports also a broadcast communication in the context of languages for service oriented computing. This could be very useful for the definition of the message exchange between the entities of the cloud-based constraint solving proposed in Chap. 7.

We have also described an algorithm that computes the minimal number of radix trees required for handling correctly every normal and broadcast message. The complexity of the correlation operation is constant for normal messages, and linearly dependent with respect to the number of targets for broadcast messages. The operations of session creation and termination have a complexity that depends on the number of different types of broadcast messages. In the worst case (i.e., when an exponential number of broadcast messages is used), it is exponential. The worst-case scenario is however impossible in practice, since real scenarios use few types of broadcast messages. For this reason the complexity of session creation and termination have in practice a constant complexity.

The major drawback of our approach is memory consumption: having more than one radix tree means that we require more memory to store the correlation values. For services that use huge data as correlation values memory consumption could be problematic. Nevertheless, we believe that in practice this is not an issue, since correlation values should be small for minimizing the cost of the message exchange over the network. If a service uses huge data as correlation values, then we argue that it is worth considering the introduction of a new shorter key that can be used as a new correlation variable.

We are currently implementing the data structure and the algorithm in the JOLIE language interpreter. With this new implementation, hopefully we will be able to provide a faster mechanism for the assignment of messages to session.

# Chapter 10
# Interruptible Request Responses in Jolie

In Service-oriented computing (SOC) interaction with remote services may incur in errors of different kinds: the remote service may disconnect, messages may be lost on the net, received data items may be outside the desired range of values, or a client may decide to interrupt the interaction with a remote service exactly in between the request and the corresponding response. To avoid that such an error causes the failure of the whole application, error handling techniques and primitives have been developed. They are commonly based on the concept of fault handler and compensation. A fault handler is a piece of code devoted to take the application to a consistent state after a fault has been caught. A compensation is a piece of code devoted to undoing the effect of a previous activity (e.g., an invocation of another service) because of a later error.

As an example, consider a hotel reservation service that requires the credit card number as a guarantee for the reservation. A reservation can be cancelled, but if it is not annulled the cost of one night will be charged in case of no show. In case the trip has to be annulled, the compensation for the successful hotel reservation has to be executed, thus cancelling the reservation and avoiding the cost of a no show.

Jolie [59] is a language for programming service-oriented applications. A main design choice in Jolie concerns its approach to the request-response interaction pattern. Jolie request-response invocation establishes a strong connection between the caller and callee, and that such a connection should not be disrupted by faults in the caller or in the callee. To this end, callee faults are notified to the caller that can thus manage them. Symmetrically, in case of caller faults the answer from the callee is waited for and used during recovery. This allows, in particular, to compensate successful remote activities which are no more needed because of the local fault. This is the case of the hotel reservation above. If the hotel reservation instead failed on its own (e.g., no room was available), compensation is not needed.

The Jolie approach for interrupting request-response interactions is different from that of WS-BPEL, according to which execution can continue without waiting for the response, and the response is discarded upon arrival. The Jolie approach allows for programming safer applications, including distributed compensation of faults. The

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_10, © Atlantis Press and the authors 2014

fact that the request-response pattern is not disrupted by errors has been formalized and proved in [59], by relying on SOCK [60], a calculus defining the formal semantics of Jolie, including its error handling features [58]. However, a nasty side effect of the Jolie approach is that the client has to wait for answers of request-response invocations before proceeding in its execution. This slows down the caller execution. For instance, referring to the hotel reservation example, the client cannot continue its operations (e.g., organizing a new trip) before the answer from the hotel has been received and (s)he gets stuck whenever the answer is lost (e.g., the hotel server unexpectedly disconnected).

This drawback is unacceptable for programming applications over the net, where communications may be delayed for long time. Such a kind of problem is normally solved using timeouts, but in the current Jolie language timeouts are not available, and one cannot implement them relying on external faults to interrupt a running request-response since execution is restarted only after the response has been received.

We propose here a new approach to error handling in Jolie, allowing on one side to compensate undesired remote side effects, and ensuring on the other side that local computation is not slowed down in case of late answers. In particular, this new approach allows to easily program timeouts.

This new variant of Jolie will be used to improve the implementation of the framework presented in Chap. 7 where the possibility of interrupting request response message is vital to handle the exchange of messages between the entities of the system. In particular, with such mechanism, we plan to create a more reliable and fault tolerant system and also to exploit timeouts to terminate the workers activity after a given time limit.

Going back to our previous example, assume that a client of the hotel reservation service exploits a request-response communication pattern in order to send the credit card number, and to subsequently receive the reservation number. In case the client does not want to wait for the reservation number for a long time, (s)he could be interested in interrupting the request-response interaction after the expiration of a timeout. In this case, according to the Jolie approach, the client is blocked waiting for the answer before being able to start other activities. On the contrary, according to the WS-BPEL approach it is not possible to write code that will be executed upon receipt of the hotel response. The mechanism that we propose in this chapter allows the client to continue immediately after the timeout expires, but it is still possible to program an activity that will be started upon the receipt of the response. This activity will be responsible for cancelling the reservation.

We also analyze how the approach has to be extended to deal not only with simple request-response, but also with invocations of multiple services. This is the case of the so called speculative parallelism, where many services are invoked simultaneously (e.g., many news servers), the first received answer is taken, the others are discarded and the corresponding invocations compensated. Nevertheless, computation in the caller restarts as soon as the first (successful) answer is received.

## 10.1 SOCK

To give a formal presentation of our approach, we first introduce SOCK [60], the calculus that defines the semantics of Jolie [59] programs, and then we extend it to account for request-response and multiple request-response service invocations. SOCK is suitable for illustrating our approach since it has a formal SOS semantics (while WS-BPEL has not), it provides request-response as a native operator, and it has a refined approach to error handling. SOCK is structured in three layers: (i) the service behavior layer, specifying the actions performed by a service, (ii) the service engine layer dealing with state and service instances (called sessions), and (iii) the services system layer allowing different engines to interact. We give a simplified description of SOCK, removing aspects not related to our aim. A description of the full calculus can be found in [60].

### *Service Behavior Layer*

The service behavior layer describes the actions performed by services. Actions can be operations on the state (SOCK is an imperative language), or communications. Basic actions can be composed using composition operators. Services are identified by the name of their operations, and by their location.

SOCK offers strong support for error handling, based on the concepts of *scope*, *fault*, and *compensation*. A scope is a process container denoted by a unique name. A fault is a signal raised by a process towards the enclosing scope when an error state is reached. A compensation is used either to smoothly stop an activity in case of an external fault, or it can be invoked to compensate the effect of the activity after its successful termination (this encompasses both termination and compensation mechanisms according to WS-BPEL terminology). Recovering mechanisms are implemented by exploiting *handlers*, which contain processes defining error recovery policies. Handlers are defined within a scope which represents the execution boundaries for their application. We use *fault handlers* and *compensation handlers*. Fault handlers are executed when a fault is triggered by the internal process of the scope. Compensation handlers are executed when a running scope is reached by an external fault or when its effect has to be annulled because of a later error. In this last case it has to be invoked by another handler.

#### Syntax

SOCK syntax is based on the following (disjoint) sets: *Var*, ranged over by $x$, $y$, for variables, *Val*, ranged over by $v$, for values, $\mathcal{O}$, ranged over by $o$, for one-way operations, $Faults$, ranged over by $f$, for faults, and $Scopes$, ranged over by $q$, for scope names. *Loc* is a subset of *Val* containing locations, ranged over by $l$.

**Table 10.1** Service behavior syntax with faults

| | | | | |
|---|---|---|---|---|
| $P, Q, \ldots ::=$ | $\overline{o}@l(\mathbf{y})$ | output | $o(\mathbf{x})$ | input |
| | $x := e$ | assignment | $P; Q$ | sequential comp. |
| | $P \mid Q$ | parallel comp. | $\sum_{i \in I} o_i(\mathbf{x}_i); P_i$ | external choice |
| | $if\ \chi\ then\ P\ else\ Q$ | det. choice | $while\ \chi\ do\ (P)$ | iteration |
| | $\mathbf{0}$ | null process | $\{P : \mathcal{H} : u\}_{q_\perp}$ | active scope |
| | $\text{inst}(\mathcal{H})$ | install handler | $\text{throw}(f)$ | throw |
| | $\text{comp}(q)$ | compensate | $\langle P \rangle$ | protection |

We denote as $SC$ the set of service behavior processes, ranged over by $P, Q, \ldots$. We use $q_\perp$ to range over $Scopes \cup \{\perp\}$, whereas $u$ ranges over $Faults \cup Scopes \cup \{\perp\}$. Here $\perp$ is used to specify that a handler is undefined. $\mathcal{H}$ denotes a function from $Faults$ and $Scopes$ to processes extended with $\perp$, i.e. $\mathcal{H} : Faults \cup Scopes \rightarrow SC \cup \{\perp\}$. In particular, we write the function associating $P_i$ to $u_i$ for $i \in \{1, \ldots, n\}$ as $[u_1 \mapsto P_1, \ldots, u_n \mapsto P_n]$. Finally, we use the notation $\mathbf{k} = \langle k_0, k_1, ..., k_i \rangle$ for vectors.

The syntax of service behavior processes is defined in Table 10.1. A one-way output $\overline{o}@l(\mathbf{y})$ invokes the operation named $o$ of a service at location $l$, where $\mathbf{y}$ are the variables that specify the values to be sent. Dually, a one-way input has the form $o(\mathbf{x})$ with $\mathbf{x}$ containing variables that will receive the communicated values. Assignment $x := e$ assigns the result of the expression $e$ to the variable $x$ (state is local to each behavior). We do not present the syntax of expressions: we just assume that they include the arithmetic and boolean operators, values in *Val* and variables. $\text{Var}(e)$ computes the set of variables in expression $e$, and $[\![e]\!]$ is the evaluation of ground expression $e$. We use $\chi$ to range over boolean expressions. $P; Q$ and $P \mid Q$ are sequential and parallel composition respectively. $\sum_{i \in I} o_i(\mathbf{x}_i); P_i$ is input-guarded external choice: whenever one of the input operations $o_i(\mathbf{x}_i)$ is invoked, continuation $P_i$ is executed. Iteration is modeled by $while\ \chi\ do\ (P)$ and deterministic choice by $if\ \chi\ then\ P\ else\ Q$. Finally, $\mathbf{0}$ is the inactive process.

We denote with $\{P\}_q$ a scope named $q$ executing process $P$. An active scope has instead the form $\{P : \mathcal{H} : u\}_{q_\perp}$. Here $\mathcal{H}$ defines the fault and compensation handlers defined in the scope. Term $\{P\}_q$ is a shortcut for $\{P : \mathcal{H}_0 : \perp\}_q$, where $\mathcal{H}_0$ is the function that evaluates to $\perp$ for all fault names (i.e. at the beginning no fault handler is defined) and to $\mathbf{0}$ for all scope names (i.e. the default compensation handler has no effect). The third argument, $u$, is the name of a handler waiting to be executed, or $\perp$ if no handler is waiting to be executed. When a scope has failed its execution, either because it has been killed from a parent scope, or because it has not been able to catch and manage an internal fault, it reaches a zombie state. Zombie scopes have $\perp$ as scope name. Primitives $\text{throw}(f)$ and $\text{comp}(q)$ respectively raises fault $f$ and asks to compensate scope $q$. $\langle P \rangle$ executes $P$ in a protected way, i.e. not influenced by external faults. This is needed to ensure that recovery from a fault is completed even if another fault happens. Handlers are installed into the nearest enclosing scope by $\text{inst}(\mathcal{H})$, where $\mathcal{H}$ is the required update of the handler function.

### Well-Formedness Rules

Informally, comp($q$) occurs only within handlers, and $q$ can only be a child of the enclosing scope. For each inst($\mathcal{H}$), $\mathcal{H}$ is undefined on all scope names $q$ but the one of the nearest enclosing scope, i.e. a process can define the compensation handler only for its own scope. Finally, scope names are unique.

### Semantics

The service behavior layer does not deal with state, leaving this issue to the service engine layer. Instead, it generates all the transitions allowed by the process behavior, specifying the constraints on the state that have to be satisfied for them to be performed. The state, and the conditions on it, are substitutions of values for variables. We use $\sigma$ to range over substitutions, and write $[\mathbf{v}/\mathbf{x}]$ for the substitution assigning values in $\mathbf{v}$ to variables in $\mathbf{x}$. Given a substitution $\sigma$, Dom($\sigma$) is its domain.

The semantics follows the idea above: the labels contain all the possible actions, together with the necessary requirements on the state. Formally, let $Act$ be the set of labels, ranged over by $a$. We use structured labels of the form $\iota(\sigma : \theta)$ where $\iota$ is the kind of action while $\sigma$ and $\theta$ are substitutions containing respectively the assumptions on the state that should be satisfied for the action to be performed and the effect on the state. We also use the unstructured labels $th(f)$, $cm(q, P)$, $inst(\mathcal{H})$.

We use operator $\boxplus$, defined as follows, for updating the handler function:

$$(\mathcal{H} \boxplus \mathcal{H}')(u) = \begin{cases} \mathcal{H}'(u) \text{ if } u \in \text{Dom}(\mathcal{H}') \\ \mathcal{H}(u) \text{ otherwise} \end{cases}$$

Intuitively, handlers in $\mathcal{H}'$ replace the corresponding handlers in $\mathcal{H}$. We also use cmp($\mathcal{H}$) to denote the part of $\mathcal{H}$ dealing with terminations/compensations: $\text{cmp}(\mathcal{H}) = \mathcal{H}|_{Scopes}$.

**Definition 10.1 (Service behavior layer semantics).** We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the rules of Tables 10.2 and 10.3, and closed w.r.t. the structural congruence $\equiv$, defined by the axioms at the bottom of Table 10.2.

Table 10.2 contains the standard semantic rules, while Table 10.3 defines the fault handling mechanism. Rule ONE- WAYOUT defines the output operation, where $\mathbf{v}/\mathbf{x}$ is the assumption on the state. Rule ONE- WAYIN corresponds to the input operation: it makes no assumption on the state, but it specifies a state update. The other rules in Table 10.2 are standard, apart from the fact that the label stores the conditions on the state. The internal process $P$ of a scope can execute thanks to rule SCOPE in Table 10.3. Handlers are installed in the nearest enclosing scope by rules ASKINST and INSTALL. According to rule SCOPE-SUCCESS, when a scope successfully ends, its compensation handlers are propagated to the parent scope. Compensation execution is required by rule COMPENSATE. The actual compensation code $Q$ is guessed, and the guess is checked by rule COMPENSATION. Faults are raised by rule THROW. A fault

**Table 10.2** Standard rules for service behavior layer ($a \neq th(f)$)

$$\text{(ONE- WAYOUT)}\ \frac{}{\overline{o}@l(\mathbf{x}) \xrightarrow{\overline{o}(\mathbf{v})@l(\mathbf{v}/\mathbf{x}:\emptyset)} \mathbf{0}} \qquad \text{(ONE- WAYIN)}\ \frac{}{o(\mathbf{x}) \xrightarrow{o(\mathbf{v})(\emptyset:\mathbf{v}/\mathbf{x})} \mathbf{0}}$$

$$\text{(ASSIGN)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(e) \quad [\![e\sigma]\!] = v}{x := e \xrightarrow{\tau(\sigma:v/x)} \mathbf{0}}$$

$$\text{(IF- THEN)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = true}{if\ \chi\ then\ P\ else\ Q \xrightarrow{\tau(\sigma:\emptyset)} P} \qquad \text{(ELSE)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = false}{if\ \chi\ then\ P\ else\ Q \xrightarrow{\tau(\sigma:\emptyset)} Q}$$

$$\text{(ITERATION)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = true}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\emptyset)} P; while\ \chi\ do\ (P)} \qquad \text{(NO- ITERATION)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = false}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\emptyset)} \mathbf{0}}$$

$$\text{(SEQUENCE)}\ \frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q} \qquad \text{(PARALLEL)}\ \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \text{(CHOICE)}\ \frac{o_i(\mathbf{x}_i) \xrightarrow{a} Q_i \quad i \in I}{\sum_{i \in I} o_i(\mathbf{x}_i); P_i \xrightarrow{a} Q_i; P_i}$$

STRUCTURAL CONGRUENCE

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P \qquad \langle \mathbf{0} \rangle \equiv \mathbf{0}$$

is caught by rule CATCH-FAULT when a scope defining the corresponding handler is met. Activities involving the termination of a sub-scope and the termination of internal error recovery are managed by the rules for fault propagation THROW-SYNC, THROW-SEQ and RETHROW , and by the partial function $killable$. Function $killable$ computes the activities that have to be completed before the handler is executed and it is applied to parallel components by rule THROW-SYNC. Moreover, function $killable$ guarantees that when a fault is thrown there is no pending handler update. Technically this is obtained by making $killable(P, f)$ undefined (and thus rule THROW-SYNC not applicable) if some handler installation is pending in $P$. The $\langle P \rangle$ operator (described by rule PROTECTION) guarantees that the enclosed activity will not be killed by external faults. Rule SCOPE-HANDLE-FAULT executes a handler for a fault. A scope that has been terminated from the outside is in zombie state. It can execute its compensation handler thanks to rule SCOPE-HANDLE-TERM, and then terminate with failure (the compensation handler will not be available any more) using rule SCOPE-FAIL. Similarly, a scope enters the zombie state when reached by a fault it cannot handle, as specified by rule RETHROW. The fault is propagated up along the scope hierarchy. Zombie scopes cannot throw faults any more, since rule IGNORE-FAULT has to be applied instead of RETHROW.

## *Service Engine Layer*

Since sessions have a limited impact on error recovery, we will present here only the rules of the service engine layer that handle the state. The syntax of the service engine is:

**Table 10.3** Faults-related rules for service behavior layer ($a \neq th(f)$)

(SCOPE)
$$\frac{P \xrightarrow{a} P' \quad a \neq inst(\mathcal{H}), cm(q', \mathcal{H}')}{\{P : \mathcal{H} : u\}_{q_\perp} \xrightarrow{a} \{P' : \mathcal{H} : u\}_{q_\perp}}$$

(INSTALL)
$$\frac{P \xrightarrow{inst(\mathcal{H})} P'}{\{P : \mathcal{H}' : u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H}' \boxplus \mathcal{H} : u\}_{q_\perp}}$$

(ASKINST) $\text{inst}(\mathcal{H}) \xrightarrow{inst(\mathcal{H})} \mathbf{0}$

(THROW) $\text{throw}(f) \xrightarrow{th(f)} \mathbf{0}$

(COMPENSATE) $\text{comp}(q) \xrightarrow{cm(q,Q)} Q$

(SCOPE- SUCCESS) $\{\mathbf{0} : \mathcal{H} : \perp\}_q \xrightarrow{inst(\text{cmp}(\mathcal{H}))} \mathbf{0}$

(SCOPE- HANDLE- FAULT) $\{\mathbf{0} : \mathcal{H} : f\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(f) : \mathcal{H} \boxplus [f \mapsto \perp] : \perp\}_{q_\perp}$

(COMPENSATION)
$$\frac{P \xrightarrow{cm(q,Q)} P', \mathcal{H}(q) = Q}{\{P : \mathcal{H} : u\}_{q'_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} \boxplus [q \mapsto \mathbf{0}] : u\}_{q'_\perp}}$$

(SCOPE- HANDLE- TERM) $\{\mathbf{0} : \mathcal{H} : q\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(q) : \mathcal{H} \boxplus [q \mapsto \mathbf{0}] : \perp\}_\perp$

(SCOPE- FAIL) $\{\mathbf{0} : \mathcal{H} : \perp\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \mathbf{0}$

(PROTECTION)
$$\frac{P \xrightarrow{a} P'}{\langle P \rangle \xrightarrow{a} \langle P' \rangle}$$

(THROW- SYNC)
$$\frac{P \xrightarrow{th(f)} P', killable(Q, f) = Q'}{P | Q \xrightarrow{th(f)} P' | Q'}$$

(THROW- SEQ)
$$\frac{P \xrightarrow{th(f)} P'}{P; Q \xrightarrow{th(f)} P'}$$

(CATCH- FAULT)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) \neq \perp}{\{P : \mathcal{H} : u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} : f\}_{q_\perp}}$$

(IGNORE- FAULT)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P : \mathcal{H} : u\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} : u\}_\perp}$$

(RETHROW)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P : \mathcal{H} : u\}_q \xrightarrow{th(f)} \langle \{P' : \mathcal{H} : \perp\}_\perp \rangle}$$

where
$killable(\{P : \mathcal{H} : u\}_q, f) = \langle \{killable(P, f) : \mathcal{H} : q\}_\perp \rangle$ if $P \not\equiv \mathbf{0}$
$killable(P \mid Q, f) = killable(P, f) \mid killable(Q, f)$
$killable(P; Q, f) = killable(P, f)$ if $P \not\equiv \mathbf{0}$
$killable(\langle P \rangle, f) = \langle P \rangle$ if $killable(P, f)$ is defined
$killable(P, f) = \mathbf{0}$ if $P \in \{\mathbf{0}, o(\mathbf{x}), \overline{o}@l(\mathbf{x}), x := e, if\ \chi\ then\ P\ else\ Q, while\ \chi\ do\ (P)$
$\sum_{i \in W} o_i(\mathbf{x}_i); P_i, \text{throw}(f), \text{comp}(q)\}$

$$Y ::= (P, \mathcal{S}) \mid Y | Y$$

A service engine can be a session $(P, \mathcal{S})$, where $P$ is a service behavior process and $\mathcal{S}$ is a state, or a parallel composition of them. A state is a substitution of values for variables. A state $\mathcal{S}$ satisfies a substitution $\sigma$, written $\mathcal{S} \vdash \sigma$, if $\sigma$ is a subset of $\mathcal{S}$. We denote with $\oplus$ the update operation on a state. The service engine layer is described by the following rules:

(ENGINE- STATE)
$$\frac{P \xrightarrow{\iota(\sigma:\rho)} P' \quad \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S} \oplus \rho)}$$

(ENGINE- PAR)
$$\frac{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S}')}{(P, \mathcal{S}) | Y \xrightarrow{\iota} (P', \mathcal{S}') | Y}$$

**Table 10.4** Rules for services system layer

$$\text{(Lift)}\ \frac{Y \xrightarrow{\iota} Y'}{Y@l \xrightarrow{\iota} Y'@l} \qquad \text{(Sync)}\ \frac{Y@l' \xrightarrow{\overline{o}(v)@l} Y'@l' \quad Z@l \xrightarrow{o(v)} Z'@l}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l} \qquad \text{(Par-Ext)}\ \frac{E_1 \xrightarrow{\iota} E_1'}{E_1 \parallel E_2 \xrightarrow{\iota} E_1' \parallel E_2}$$

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

### *Services System Layer*

The services system models the composition of different engines into a system. The services system syntax is:

$$E ::= Y@l \mid E \parallel E$$

A service system $E$ can be a located service engine $Y@l$ or a parallel composition of them. The semantics is defined by the rules in Table 10.4 and closed w.r.t. the structural congruence $\equiv$ therein. Rule Lift propagates an action to a located engine. Rule Sync allows to synchronize an output with the corresponding input. Par-Ext deals with parallel composition.

## 10.2 Request–Response Interaction Pattern

A request-response pattern is a bi-directional interaction where a client sends a message to a server and waits for an answer. When a server receives such a message, it elaborates the answer and sends it back to the client. In the literature there are two proposals to deal with a client that fails during a request-response interaction. The WS-BPEL approach kills the receive activity and, when the message arrives, it is silently discarded. In Jolie instead, clients always wait for the answer and exploit it for error recovery. In particular, in case of a successful answer, handlers may be updated by the continuation of the request-response operation.

Here we present an intermediate approach: in case of failure we wait for the answer, but without blocking the computation. Moreover, when the answer is received we allow for the execution of a compensation activity.

We now describe our approach. Let $\mathcal{O}_r$ be the set of request-response operations, ranged over by $o_r$. We define the request-response pattern in terms of the output primitive $\overline{o_r}@l(\mathbf{y}, \mathbf{x}, P)$, also called solicit, and of the input primitive $o_r(\mathbf{x}_1, \mathbf{y}_1, Q)$. When interacting, the client sends the values from variables $\mathbf{y}$ to the server, that stores them in variables $\mathbf{x}_1$. Then, the server executes process $Q$ and, when $Q$ terminates, the values in variables $\mathbf{y}_1$ are sent back to the client who stores them in variables $\mathbf{x}$. Only at this point the execution of the client can restart. If a fault occurs on the client-side (e.g., because of a parallel thread) after the remote service has been invoked, but before the answer is received, we allow the client to handle the fault regardless

of the reply, so that recovery can start immediately. However, we create a receiver for the missing message in a fresh session so that, if later on the message is received, the operation can be compensated. The compensation is specified by the parameter $P$ of the solicit operation. If instead a fault is raised on the server-side during the computation of the answer, the fault is propagated to the client where it raises a local fault. In this case there is no need to compensate the remote invocation, since we assume that this is dealt with by local recovery of the server.

To formally specify the behavior informally described above, we have to update the three layers of SOCK architecture.

## *Service Behavior Calculus: Extension*

To define the behaviour of the request-response pattern, we extend the syntax of the behavioral layer with the request-response primitive and with few auxiliary operators that are used to define its semantics.

| | | | |
|---|---|---|---|
| $\overline{o_r}@l(\mathbf{y}, \mathbf{x}, P)$ | Solicit | $o_r(\mathbf{x}_1, \mathbf{y}_1, Q)$ | Request-Response |
| $Exec(l, o_r, \mathbf{y}, P)$ | Req.-Resp. execution | $Wait(o_r, \mathbf{y}, P)$ | Wait |
| $\overline{o_r}!f@l$ | Fault output | $Bubble(P)$ | Bubble |

$Exec(l, o_r, \mathbf{y}, P)$ is a server-side running request-response: $P$ is the process computing the answer, $o_r$ the name of the operation, $\mathbf{y}$ the vector of variables to be used for the answer, and $l$ the client location. Symmetrically, $Wait(o_r, \mathbf{y}, P)$ is the process waiting for the response on client-side: $o_r$ is request-response operation, $\mathbf{y}$ is the vector of variables to be used for storing the answer and $P$ is the compensation code to run in case the client fails before the answer is received. When a fault is triggered on the server-side, an error notification has to be sent to the client: this is done by $\overline{o_r}!f@l$, where $o_r$ is the name of the operation, $f$ the name of the fault and $l$ the client location. As we have said, if a fault occurs on client-side, we have to move the receipt operation to a fresh, parallel session, so that error recovery can start immediately. This is done by the primitive $Bubble(P)$, which allows to create a new session executing code $P$. We name it "bubble" since we see $P$ as a bubble that goes up in the scope hierarchy, and installs $P$ when it arrives at the top level. This primitive is the key element that allows a failed solicit to wait for a response outside its scope and potentially allowing its termination regardless of the arrival of the answer.

The semantics of the behavioral layer is extended with the rules presented in Table 10.5. Function *killable* is also extended, as follows:

- $killable(Exec(l, o_r, \mathbf{y}, P), f) = killable(P, f) | \langle o_r!f@l \rangle$
- $killable(Wait(o_r, \mathbf{x}, P), f) = Bubble(Wait(o_r, \mathbf{x}, \mathbf{0}); P)$
- $killable(\overline{o_r}!f@l, f) = \overline{o_r}!f@l$
- $killable(Bubble(P), f) = Bubble(P)$

Rules SOLICIT and REQUEST start a solicit-response operation on client and server side respectively. Upon invocation, the request-response becomes an active construct

**Table 10.5** Request-response pattern rules

(SOLICIT)
$$\overline{o_r}@l(\mathbf{y}, \mathbf{x}, P) \xrightarrow{\overline{o_r}(\mathbf{v})@l(\emptyset:\mathbf{v}/\mathbf{x})} Wait(o_r, \mathbf{x}, P)$$

(REQUEST)
$$o_r(\mathbf{x}, \mathbf{y}, P) \xrightarrow{o_r(\mathbf{v})::l(\emptyset:\mathbf{v}/\mathbf{x})} Exec(l, o_r, \mathbf{y}, P)$$

(REQUEST- EXEC)
$$\frac{P \xrightarrow{a} P'}{Exec(l, o_r, \mathbf{y}, P) \xrightarrow{a} Exec(l, o_r, \mathbf{y}, P')}$$

(THROW- REXEC)
$$\frac{P \xrightarrow{th(f)} P'}{Exec(l, o_r, \mathbf{y}, P) \xrightarrow{th(f)} P' \mid \langle \overline{o_r}!f@l \rangle}$$

(REQUEST- RESPONSE)
$$Exec(l, o_r, \mathbf{y}, \mathbf{0}) \xrightarrow{\overline{o_r}(\mathbf{v})@l(\mathbf{v}/\mathbf{y}:\emptyset)} \mathbf{0}$$

(SOLICIT- RESPONSE)
$$Wait(o_r, \mathbf{x}, P) \xrightarrow{o_r(\mathbf{v})(\emptyset:\mathbf{v}/\mathbf{x})} \mathbf{0}$$

(SEND- FAULT)
$$\overline{o_r}!f@l \xrightarrow{\overline{o_r}(f)@l(\emptyset:\emptyset)} \mathbf{0}$$

(RECEIVE FAULT)
$$Wait(o_r, \mathbf{x}, P) \xrightarrow{o_r(f)(\emptyset:\emptyset)} \text{throw}(f)$$

(CREATE BUBBLE)
$$Bubble(P) \xrightarrow{\tau(\emptyset:\emptyset)[\![P]\!]} \mathbf{0}$$

executing process $P$, and storing all the information needed to send back the answer. The execution of $P$ is managed by rule REQUEST-EXEC. When the execution of $P$ is terminated, rule REQUEST-RESPONSE sends back an answer. This synchronizes with rule SOLICIT-RESPONSE on the client side, concluding the communication pattern.

When an executing request-response is reached by a fault, it is transformed into a fault notification (see rule THROW-REXEC and the definition of function $killable$) on server side. Fault notification is executed by rule SEND-FAULT, and it will interact with the waiting receive thanks to rule RECEIVE-FAULT. When received, the fault is ready to be re-thrown at the client side, where it is treated as a local fault.

A fault on client side instead gives rise to a bubble, creating the process that will wait for the answer in a separate session. The bubble is created by rule CREATE BUBBLE, and will be installed at the service engine level. The label for bubble creation has the form $\tau(\emptyset : \emptyset)[\![P]\!]$, where $P$ is the process to be run inside the new session. Here and in the following, to simplify the presentation, we will write $\tau(\emptyset : \emptyset)$ for $\tau(\emptyset : \emptyset)[\![\mathbf{0}]\!]$. The new receive operation inside the bubble has no handler update, since it will be executed out of any scope, and its compensating code $P$ has been promoted as a continuation. In this way, $P$ will be executed only in case of successful answer. In case of faulty answer, the generated fault will have no effect since it is in a session on its own.

## *Service Engine Calculus: Extension*

We have to add to the service engine layer a rule for installing bubbles: when a bubble reaches the service engine layer, a new session is started executing the code inside the bubble.

$$\frac{P \xrightarrow{\tau(\emptyset:\emptyset)[\![Q]\!]} P' \quad Q \neq \mathbf{0}}{(P, \mathcal{S}) \xrightarrow{\tau} (P', \mathcal{S}) \mid (Q, \mathcal{S})} \text{(Engine- Bubble)}$$

## *Service System Calculus: Extension*

The service system calculus is expanded with the rule modeling the request-response communication. The rule uses the relation *comp* to match corresponding input output actions.

$$\frac{Y@l' \xrightarrow{a} Y'@l' \quad Z@l \xrightarrow{a'} Z'@l \quad comp(a, a')}{Y@l' | Z@l \xrightarrow{\tau} Y'@l' | Z'@l} \text{(Request- Response Sync)}$$

where $comp = \{(\overline{o_r}(\mathbf{v})@l,\ o_r(\mathbf{v}) :: l'),\ (\overline{o_r}(\mathbf{v})@l,\ o_r(\mathbf{v})),\ (\overline{o_r}(f)@l,\ o_r(f))\}$.

## *Example and Properties*

We present now an example of usage of the request-response primitive and prove some basic properties. We show a first solution for the hotel reservation example described in the introduction:

```
CLIENT:==
     book_r@hotel_Imperial(⟨CC,dates⟩,⟨res_num⟩,
       annul@hotel_Imperial(⟨res_num⟩));
      P
```

The $book_r$ operation transmits the credit card number `CC` and the dates of the reservation and waits for the reservation number. In case the user wants to cancel the reservation before receiving an answer from the hotel a fault can be used to kill this operation. In such a case the `annul` operation is invoked when the answer is received to compensate the $book_r$ operation. The `annul` operation will be executed in a new session by using our mechanism based on bubbles.

As a more concrete instance, we could consider the case where the user is willing to wait a limited amount of time for the answer from the hotel, after which (s)he will cancel the reservation. This case could be programmed by assuming a service *timeout* that offers a request-response operation that sends back an answer after $n$ seconds.[1] The *timeout* service can be used to add a timeout in our example as follows:

[1] Clearly, because of networks delay the answer may be received later than expected.

```
CLIENT:==
    res_num:= 0;
    {
     inst(f ↦ if res_num==0 then throw(tm));
     (
     timeout_r@timeout(⟨60⟩,⟨ ⟩,0); throw(f)
     |
     book_r@hotel_Imperial(⟨CC,dates⟩,⟨res_num⟩,
       annul@hotel_Imperial(⟨res_num⟩)); throw(f)
     )
    }_q ; P
```

In this scenario the timeout operation is invoked in parallel with the booking service. The first operation that finishes raises the fault $f$ that is caught by the handler of the scope $q$. The fault will kill the remaining operation and if the hotel response has not arrived yet (i.e. the value of res_num is still 0) then the fault $tm$ is raised. $P$ is executed otherwise.

Note that a similar solution is not viable in BPEL: in case the timeout triggers, the booking invocation is killed, and if an answer arrives, it is discarded. Thus one does not know whether the invocation succeeded or not, neither which was the reservation number in case it succeeded.

In Jolie, the answer is used for error recovery. However, in case no answer is received from the booking service, the whole service engine gets stuck. In our approach instead the main session can continue its execution without delays.

It is difficult to apply the proposed solution when two or more solicits install handlers or require compensation. Indeed, if two solicits are executed in parallel, they can receive their answers simultaneously. It is thus difficult to accept one of the answers and compensate the other one. One may try to exploit the handler update primitive, but in this way compensations are executed inside the scope, thus they have to be terminated before execution can proceed. To solve this problem, one may try to change the semantics to execute those handlers in a separate session but this would not be meaningful in general, since the handler may want to update the local state. This is not the case for compensations of request-responses, which only need to produce remote effects. An additional difficulty is that fault notifications from the invoked services should be masked as long as there are invocations that may succeed. These reasons justify the multiple solicit response primitive introduced in the next section.

The fact that a response is always waited for is captured by the proposition below.

**Proposition 10.1.** *Let* $Y \xrightarrow{a_1} Y_1 \xrightarrow{a_2} Y_2 \ldots \xrightarrow{a_n} Y_n$ *be a computation of an engine. Let* $a_1$ *be* $\overline{o_r}(\mathbf{v})@l$*, i.e. the start action in a solicit-response. Then there are two possible cases:*

1. *the response has been received:* $a_i = o_r(\mathbf{v}')$ *or* $a_i = o_r(f)$ *for some* $i$*;*
2. *the process is waiting for the response:* $Y_n \xrightarrow{o_r(\mathbf{v}')} Y'$.

## 10.3 Multiple Request–Response Communication Pattern

The Sect. 10.2 presented the request-response pattern, where one invocation is sent and one answer is received. For optimization reasons, it may be important to invoke many services providing the same facility (e.g., many hotels or news services), and only consider the first received answer. This pattern is known as speculative parallelism.

We model this communication pattern using a dedicated primitive that we call multiple solicit-response (MSR for short) and that can be seen as a generalization of the solicit-response from the Sect. 10.2. The idea is that a MSR consists of a list of solicit-responses, each one equipped with its own continuation. Formally, we define the syntax of the MSR primitive as $MSR\{z_1, \ldots, z_n\}$ where each $z_i$ is a *solicit-response with continuation* written $z_i = \overline{o_{r_i}}@l_i(\mathbf{y}_i, \mathbf{x}_i, P_i) \mapsto Q_i$. Intuitively, the continuation $Q_i$ is executed only when $\overline{o_{r_i}}@l_i(\mathbf{y}_i, \mathbf{x}_i, P_i)$ is the first to receive a successful answer (i.e. not a fault notification). Thus, at most one of the $Q_i$ will be executed (exactly one if a non-faulty answer is received).

In the following we extend the SOCK calculus with the MSR primitive.

### *Service Behavior Calculus: Extension*

We add to the syntax of the service behavior calculus the MSR primitive together with some auxiliary operators that we need in order to define the MSR semantics.

$$
\begin{array}{rll}
P, Q ::= & \ldots & \\
 & MSR\{z_1, \ldots, z_n\} & \text{multiple solicit-response} \\
 & Wait^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m) & \text{multiple wait} \\
z ::= & \overline{o_r}@l(\mathbf{y}, \mathbf{x}, P) \mapsto Q & \text{solicit with continuation} \\
w ::= & Wait(o_r, \mathbf{y}, P) \mapsto Q & \text{wait with continuation}
\end{array}
$$

In a MSR the solicits are sent one after the other, and only when all the requests have been sent the MSR can receive a response. For this reason we introduce the multiple wait $Wait^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m)$ that specifies the solicits that still have to be sent $z_1, \ldots, z_n$, and the ones that will wait for an answer $w_1, \ldots, w_n$. Thus, the MSR primitive $MSR\{z_1, \ldots, z_n\}$ above is a shortcut for $Wait^+(z_1, \ldots, z_n \triangleright)$. Moreover, we have that a multiple wait with only one waiting process is structurally equivalent to a standard wait, as shown at the bottom of Table 10.6. We formally define the behavior of the MSR primitive by extending the service behavior semantics with rules presented in Table 10.6.

The multiple wait executes all the solicit-responses through rule MSR-SOLICIT. Once all the solicits have been sent (and therefore the process $Wait^+(\triangleright w_1, \ldots, w_m)$ is obtained), the multiple wait receives a successful answer through rule MSR-RESPONSE. It continues the execution with the corresponding continuation code, and kills all the other solicits by creating a bubble for each remaining waiting process. If a fault notification arrives as an answer, it is discarded by rule MSR-IGNORE FAULT

**Table 10.6** Multiple request-response pattern rules

(MSR- SOLICIT)

$$\frac{z_1 = \overline{o_r}@l(\mathbf{y}, \mathbf{x}, P) \mapsto Q \quad w_{m+1} = Wait(o_r, \mathbf{y}, P) \mapsto Q}{Wait^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m) \xrightarrow{\overline{o_r}(\mathbf{v})@l(\emptyset:\mathbf{v}/\mathbf{x})} Wait^+(z_2, \ldots, z_n \triangleright w_1, \ldots, w_m, w_{m+1})}$$

(MSR- RESPONSE)

$$\frac{\forall k \in \{1, \ldots, n\} : w_k = Wait(o_{r_k}, \mathbf{y}_k, P_k) \mapsto Q_k \quad i \in \{1, \ldots, n\} \quad J = \{1, \ldots, n\} \setminus \{i\}}{Wait^+(\triangleright w_1, \ldots, w_n) \xrightarrow{o_{r_i}(\mathbf{v})(\emptyset:\mathbf{v}/\mathbf{y_i})} Q_i \,|\, \prod_{j \in J} Bubble(Wait(o_{r_j}, \mathbf{y_j}, \mathbf{0}); P_j)}$$

(MSR- IGNORE FAULT)

$$\frac{n > 1 \quad w_i = Wait(o_{r_i}, \mathbf{y_i}, P_i) \mapsto Q_i \quad i \in \{1, \ldots, n\}}{Wait^+(\triangleright w_1, \ldots, w_n) \xrightarrow{o_{r_i}(f)(\emptyset:\emptyset)} Wait^+(\triangleright w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n)}$$

$$Wait^+(\triangleright Wait(o_r, \mathbf{y}, P) \mapsto Q) \equiv Wait(o_r, \mathbf{y}, P); Q$$

if there is at least another available wait (which may succeed). If instead there is no other solicit waiting for an answer, the last fault received is raised (rule RECEIVE FAULT described in Table 10.5 of the previous section). When an external fault arrives a bubble containing a dead solicit response is created for every solicit that has been sent, as specified by the function *killable* that is extended in the following way:

$$killable(Wait^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m), f) = \prod_{Wait(o_{r_j}, \mathbf{y_j}, P_j) \mapsto Q_j \in \{w_1, \ldots, w_m\}} Bubble(Wait(o_{r_j}, \mathbf{y_j}, \mathbf{0}); P_j)$$

The MSR primitive is perfectly suited to capture speculative parallelism scenarios. Consider, for instance, the parallel requests of news towards different news servers. Suppose that we are interested only in the first answer and that we do not need to compensate the arrival of the other answers. If we have three news servers located at sites A, B and C we can implement this pattern in the following way:

```
NEWS_READER(news_title) :==
  msr {
    get_news_r@siteA(⟨news_title⟩,⟨result⟩,0) ↦ 0
    get_news_r@siteB(⟨news_title⟩,⟨result⟩,0) ↦ 0
    get_news_r@siteC(⟨news_title⟩,⟨result⟩,0) ↦ 0
  }
```

The MSR can also be used to easily solve the hotel reservation problem defined in the introduction. Suppose for instance that you would like to use two booking services for making the hotel reservation and that you would like to get the acknowledgment in 1 min. If the booking services are located at site A and B and if we use the *timeout* service introduced before, this service could be defined as:

```
CLIENT:==
  msr {
```

$\overline{timeout_r}$@timeout(⟨60⟩,⟨⟩,**0**) ↦ throw($tm$)
$\overline{book_r}$@H_1(⟨CC,dates⟩,⟨res_num⟩, annul@H_1(⟨res_num⟩)) ↦ **0**
$\overline{book_r}$@H_2(⟨CC,dates⟩,⟨res_num⟩, annul@H_2(⟨res_num⟩)) ↦ **0**
}

The following proposition extends Proposition 10.1 to deal with MSR. Here either one successful answer is received, or all faulty answers are received.

**Proposition 10.2.** *Let* $Y \xrightarrow{a_1} Y_1 \xrightarrow{a_2} Y_2 \ldots \xrightarrow{a_n} Y_n$ *be a computation of an engine. Assume that the engine contains MSR*$\{z_1, \ldots, z_m\}$ *where* $z_i = \overline{o_{r_i}}@l_i(\mathbf{y}_i, \mathbf{x}_i, \mathcal{H}_i, P_i)$. *If* $a_1 = \overline{o_{r_1}}(\mathbf{v})$, i.e. *the start action in the MSR,*[2] *then there are three possible cases:*

1. *a successful response has been received:* $a_j = o_{r_i}(\mathbf{v})$ *for some* $j > 1$ *and some* $i \in [1, m]$;
2. *the process is waiting for a response:* $Y_n \xrightarrow{o_{r_i}(\mathbf{v})} Y'$ *for some* $i \in [1, m]$;
3. *the process has received all faulty responses:* $a_{j_1} = o_{r_1}(f), \ldots, a_{j_m} = o_{r_m}(f)$ *for some* $j_1, \ldots, j_m > 1$

## 10.4 Related Works and Conclusions

In this chapter we have presented a new approach to model request-response interactions in SOC languages which allows a more natural treatment of faults. According to our proposal, and differently from the case of Jolie, a request-response invocation can be interrupted when a fault occurs, thus avoiding slowing down or blocking of the computation. On the other hand, differently from the case of WS-BPEL, after a request-response is interrupted it is still possible to trigger, upon arrival of a response, a fault-handler process which can close gracefully the conversation with the invoked service. Our approach allows us also to easily program timeouts.

This variant of Jolie will be used to improve the implementation of the framework presented in Chap. 7 exploiting the timeouts and the new request response primitive to handle the communications between the entities of the framework and, at the same time, improving its reliability, its efficiency and fault tolerance.

Technically our proposal has been formalized in terms of the SOCK calculus [60], by defining a new syntax and semantics for the request-response primitive. This allows one to specify the compensation code to be executed when an answer for a request-response whose client already failed arrives. The proposed solution exploits SOCK management of sessions for executing the compensation in a separate concurrent session.

We have also defined multiple solicit-response (MSR), which allows one to invoke many services at once. Also in this case we allow graceful interruption, since, after the first response has been received, the other pending invocations are interrupted and

[2] We assume we have no other similar invocations enabled.

later compensated. This primitive comes handy when developing real applications, especially for programming speculative parallelism scenarios.

Apart from Jolie and WS-BPEL there are several other languages for SOC which have considered the problem of interrupting a pending invocation and/or faults. Nevertheless the solution that we propose is different from the existing ones, both in the way we interrupt a request-response interaction and in the compensation mechanism.

Web-$\pi$ [78, 79] is a language designed for modeling Web transactions and therefore it pays particular attention to compensation mechanisms. However web-$\pi$ has no request-response pattern and its treatment of faults and of scopes is rather different from ours. Orc [71] is a language designed to express orchestrations and wide-area computations in a simple and structured manner. This language has a pruning primitive that is conceptually similar to our MSR and that allows one to wait for a result from one out of two services. However, since Orc has no notion of fault, all the difficulties coming from error management do not emerged in their framework. For instance in Orc implementing "one out of n" speculative parallelism using "one out of two" is trivial, while this is not the case in presence of faults. Finally, we compare our approach with the service oriented calculi CaSPiS [23] and COWS [80]. CaSPiS is based on bi-directional sessions and includes a low-level session closure mechanism: upon closure of one session side, an asynchronous notification is sent to the opposite one. In order to close also the opposite side, it is necessary to explicitly program a corresponding handler. Also the primitives for fault handling and compensations in COWS [80] are more low level than ours, thus leaving to the programmer the responsibility for defining error handling policies. Our approach is the opposite one, since we aim at providing primitives which free the programmer from this burden.

# Chapter 11
# Conclusions

What happens when constraint meets concurrency? In this thesis we give a partial answer showing the benefits of using ideas of concurrency theory in the constraint world and vice versa.

In the first part of the thesis we considered Constraint Handling Rules (CHR), a well-known concurrent language that supports constraints as primitive constructs. We studied its expressive power focusing first on some of its fragments and then considering what happens when priorities are added. In the second part of the thesis we propose instead a framework that is written using a concurrent language and uses a concurrent system to ease the resolution of constraint problems.

The original contributions of this thesis are the following:

- a study of two significant non-Turing powerful fragments of CHR;
- an analysis of the expressive power of static and dynamic priorities in CHR;
- a definition and an implementation in Jolie of a modular and flexible framework that allows to solve CSPs using a distributed system;
- a study of machine learning approaches to improve the CSP solving;
- a new approach based on classifiers to solve a set of CSPs;
- new algorithms for an efficient exchange of broadcast messages in Service-Oriented systems;
- a new approach to handle faults within a Service-Oriented Language.

We have just started to scratch the surface of the benefits deriving from the interaction between ideas coming from concurrency and constraint theory. There are hundreds of concurrent languages that can be enriched with constraint primitives to improve their expressive power. We are experiencing an increasing attention toward such tasks, e.g., [17, 26, 27], and we expect a continuation of this trend in the future.

As far as the use of concurrent system is concerned, we argue that in the future at least part of a constraint solving framework will be written using a concurrent language and then deployed in a concurrent system. Like Gerard Holzmann pointed out in [64] for the model checking community, in the past the tremendous improvements obtained within the constraint programming community were due to the Moore's law and to the improvement of algorithms and heuristics. Unfortunately, even though we

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_11, © Atlantis Press and the authors 2014

cannot be certain, it seems that we are on the verge of the physical limits of the transistor technology and we cannot count on having every year always a faster processor to use. However, we can expect to have systems with more and more computing units. This means that, unless a new paradigm of computation emerges, the only way to significantly improve the current state of the research is to start considering algorithms that exploit concurrent systems. Communities like model checking, SAT solving, and data mining have already started this process, and we are currently seeing a great interest for these themes also in the constraint community. The work described in the second part of the thesis is certainly going toward this direction.

In the remaining parts of these conclusions we will not describe in detail all the results obtained or the possible future directions that could be taken, the interested reader can find them at the end of every chapter. Here we are more interested in presenting a more global, personal, and philosophical view.

Today the research is "specialized": scientists focus on always smaller sectors of the human knowledge. This process is causing the estrangement between physics, mathematics, computer science, and biology and, even within the computer science community, we are experiencing a partition into smaller and smaller communities that are focusing only on their own topics, sometimes without bothering to check if others have interesting ideas that could be used inside their research.

In this thesis we try to go against the tide and see what ideas and concepts can be borrowed from the concurrency and constraint communities to help both of them. It is almost impossible to predict all the possible interactions between these fields. This work is just a first step, and much more could be done also considering the contribution of other areas of research. In this thesis, as an example, we experienced also with machine learning techniques coming from artificial intelligence, portfolio theory studied in economics, game theory, and scheduling heuristics that are well-studied in the operative system community.

Certainly, some readers will say that these kinds of researches are not worth following. They will probably say that these studies will have a small impact, if not none, on the real word or that there is no need to delve into the study of the ideas that have been already extensively studied and understood within a given field. To these readers we would like just to remind that discoveries are unpredictable. There are strong philosophical reasons that support this thesis, see for instance [105]. In this context, we would like just to recall some examples showing that discoveries were not planned in advance and often discoverers were not even addressing the problem they later solved.[1] Cristoforo Colombo, for instance, was not planning to discover America, he was simply trying to find a new route to India. Complex numbers, discovered by Gerolamo Cardano, are now used in many fields of human knowledge but originally, they were invented to solve mathematical conundrums. Arno Penzias and Robert Wilson were in charge of building a very sensitive antenna intended for communicating with satellites. In the process they encountered radio noise which was later identified as the cosmicmicrowave background radiation, one

[1] For a good and narrative presentation of this topic see [74] or [121].

of the best available evidence for the Big Bang theory. Arthur Leonard Schawlow, Nobel Prize winner for the creation of the laser, said "We had no application in mind. If we had, it might have hampered us and not worked out as well." Alexander Fleming discovered the penicillin when he noticed the infection-fighting propriety of a fungus that contaminated some experiments that he was conducting for investigating the properties of staphylococci. He said "When I woke up just after dawn on September 28, 1928, I certainly didn't plan to revolutionize all medicine by discovering the world's first antibiotic, or bacteria killer." Viagra was not discovered with the purpose of treating erectile dysfunction, in fact the Pfizer scientists that discovered the drug were just testing out a novel way to control high blood pressure. And what about the Internet that was created for military reasons?

The list of accidental discoveries that changed the world can go on. But let us not forget that history is full of silly predictions like the one attributed to the commissioner of the US Patent Office that in 1899 said "Everything that can be invented has been invented" or the physicist and Nobel laureate Albert Abraham Michelson, who at the beginning of the twentieth century said, "The most important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplemented by new discoveries is exceedingly remote." Another funny prediction is due to the French philosopher Auguste Comte who said, "Of all objects, the planets are those which appear to us under the least varied aspect. We see how we may determine their forms, their distances, their bulk, and their motions, but we can never know anything of their chemical or mineralogical structure; and, much less, that of organized beings living on their surface." However, the ink was not yet dried when the spectroscope allowing astronomers to identify elements in the solar atmosphere was invented. Famous are also the statements of people who did not understad the greatness of a discovery. For instance, the president of the Linnean Society of London remarked in May 1859 that, the previous year had not been marked by any revolutionary discoveries. Note that in June 1858, Darwin presented his work on the evolution of species and it was presented at the Linnean Society of London.

As opposed to these people we consider more trustworthy people who have an open mind like the polymath John von Neumann, who once said "It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."

Discoveries cannot be predictable, it is not possible to say a priori that something is not worth following. That is its beauty but also the tragedy of research. We need to understand that, be humble enough to consider every possible idea, dream the impossible, and, in case of failure, continue searching. Serendipity[2] is after all everywhere.

[2] Note that the etymology of the word "serendipity" comes from the Persian fairy tale "The Three Princes of Serendip," whose heroes "were always making discoveries, by accidents and sagacity, of things they were not in quest of."

# Appendix A
# Proofs

## A.1 Lemma 5.1

**Lemma 5.1** *$\leq$ is a well-quasi-order on Conf.*

*Proof.* $\leq$ is trivially a transitive and reflexive relation. To prove that $\leq$ is a well-quasi-order we have to prove that, for every infinite sequence $seq = s_0, s_1, \ldots$ of configurations, there exist $i$, $j$ s.t. $i < j$ and $s_i \leq s_j$.

The proof is by contradiction. Suppose there exists a sequence $seq = s_0, s_1, \ldots$ such that there are no $i$, $j$ with $i < j$ and $s_i \leq s_j$. Since the variables and constants in *Conf* are finite, the possible number of built-in stores in a state in *Conf* is finite. Since $seq$ is infinite this implies that there is an infinite subsequence $seq_0 = s_{0,0}, s_{0,1}, \ldots$ of $seq$ such that every state in $seq_0$ has the same built-in store.

Starting with $seq_0$ if $seq_k = s_{k,0}, s_{k,1}, \ldots$ and if $s_{k,0} \not\leq s_{k,1}$ let us define the sequence $seq_{k+1}$ in the following way:

- given a configuration $s = \langle G, S, B\rangle_i$ and a constraint $c$, let $\theta_{\text{goal}}(s, c) = |\{c \in G\}|$ and $\theta_{\text{store}}(s, c) = |\{i \ . \ c\#i \in S\}|$
- let $c$ be a constraint in $s_{k,0}$ s.t. $\theta_{\text{goal}}(s_{k,0}, c) > \theta_{\text{goal}}(s_{k,1}, c)$ or $\theta_{\text{store}}(s_{k,0}, c) > \theta_{\text{store}}(s_{k,1}, c)$ (this constraint exists since $s_{k,0} \not\leq s_{k,1}$)
- let $seq'_k$ be the subsequence of $seq_k$ obtained by deleting all the configurations $s$ in $seq_k$ s.t. $\theta_{\text{goal}}(s, c) = m$ if $\theta_{\text{goal}}(s_{k,0}, c) > \theta_{\text{goal}}(s_{k,1}, c) = m$ or otherwise $\theta_{\text{store}}(s, c) = n$ if $\theta_{\text{store}}(s_{k,0}, c) > \theta_{\text{store}}(s_{k,1}, c) = n$
- if the sequence obtained from $seq_k$ by deleting the configurations in $seq'_k$ is infinite, let $seq_{k+1}$ be this sequence, $seq_{k+1} = seq'_k$ otherwise.

Since the number of variables and constants in *Conf* is finite, the number of different constraints in *Conf* is finite. Therefore, $seq_{k+1}$ is equal to $seq'_k$ only a finite number of times. On the other hand, every configuration contains a finite number of constraints and therefore after a finite number of configuration deletions (i.e. when $seq_{k+1} \neq seq'_k$) we will have that the first configuration of the sequence is smaller than all the others.

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6, © Atlantis Press and the authors 2014

Thus there is an $l$ s.t. $s_{l,0} \leq s_{l,1}$. But this is impossible because for definition of $seq_l$ there exist $i$, $j$ s.t. $i \leq j$, $s_i = s_{l,0}$ and $s_j = s_{l,1}$.

## A.2 Lemma 5.2

**Lemma 5.2** *Given a $CHR^{\omega_a}(C)$ program $P$, $(Conf, \overset{\omega_o}{\rightarrow}_P, \leq)$ is a well-structured transition system with strong compatibility.*

*Proof.* Given Lemma 5.1 it suffices to prove property 2 of Definition 5.1. Suppose that $s_1 \leq t_1$ and $s_1 \overset{\omega_o}{\rightarrow}_P s_2$. There are three possible cases:

1. if $\overset{\omega_o}{\rightarrow}_P$ is a **Solve** transition, then the same transition can be executed from the configuration $t_1$. If $t_1 \overset{\omega_o}{\rightarrow}_P t_2$ we trivially have that $s_2 \leq t_2$.
2. if $\overset{\omega_o}{\rightarrow}_P$ is an **Introduce** transition, then the same transition can be executed from the configuration $t_1$. If $t_1 \overset{\omega_o}{\rightarrow}_P t_2$ we trivially have that $s_2 \leq t_2$.
3. if $\overset{\omega_o}{\rightarrow}_P$ is an **Apply** transition, since every constraint in $s_1$ is also present in $t_1$, a head rule that matches with $s_1$ matches also with $t_1$. Since the built-in stores of $s_1$ and $s_2$ are equal then every guard satisfied in $s_1$ is also satisfied in $t_1$. Therefore from $t_1$ it is possible to fire the same rule fired from $s_1$. If $t_1 \overset{\omega_o}{\rightarrow}_P t_2$, we have that $s_2 \leq t_2$ since the number of constraints added or removed in both transitions are the same.

## A.3 Lemma 5.3

**Lemma 5.3** *Let $X$ be a set of variables and let $s = \langle c_1, c_2\rangle \cdots \langle c_{n-1}, c_n\rangle$ be a strictly increasing sequence with respect to $X$. Then $n \leq |X| + 2$.*

*Proof.* The proof follows by observing that if $s = \langle c_1, c_2\rangle \cdots \langle c_{n-1}, c_n\rangle$, then by definition of strictly increasing sequence, for each $i \in [1, n-1]$ we have that $CT \models d_i \not\rightarrow c_{i+1}$ and $CT \models c_i \not\rightarrow d_i$.

## A.4 Lemma 5.4

**Lemma 5.4** *Let $Const$ be a finite set of constants and let $S$ be a finite set of variables such that $u = |Const|$ and $w = |S|$. The set of sequences $s$ which are strictly increasing with respect to $S$ (up to logical equivalence) is finite and has cardinality at the most*

$$\frac{2^{w(u+w)(w+3)} - 1}{2^{w(u+w)} - 1}.$$

*Proof.* The proof follows from the following observations:

- For each variable in $X \in S$, we have at the most $u + w$ possible instantiations and therefore for the variables in $S$, we have at the most $w(u+w)$ different combinations of instantiations of variables.
- Each constraint $c$ such that $Fv(c) \subseteq S$ can be viewed as a subset of all the possible combinations of instantiations of the variables in $S$ and therefore there are at the most $k = 2^{w(u+w)}$ different constraints (up to logical equivalence) such that $Fv(c) \subseteq S$.
- By Lemma 5.3, if $s$ is a strictly increasing sequence with respect to $S$, then $s$ contains at the most $w+2$ constraints. Moreover, by the previous point for $i \leq w+2$ the number of distinct strictly increasing sequences (up to logical equivalence) is at the most $k^i$.

Then the number of sequences $s$ which are strictly increasing with respect to $S$ (up to logical equivalence) is at most

$$\sum_{i=0}^{w+2} k^i = \frac{k^{w+3} - 1}{k - 1} = \frac{2^{w(u+w)(w+3)} - 1}{2^{w(u+w)} - 1}.$$

## A.5 Lemma 5.5

**Lemma 5.5** *Let $\delta$ be a terminating computation for the goal $G$ in the $CHR_1^{\omega_a}(C)$ program $P$. Assume that $F_\delta$ is $l$-repetitive with $p = dg(F_\delta)$ and assume that there exist an $l$-repetitive sc-computation $\sigma$ of $F_\delta$ and a repetition $k\#l^i \in \sigma$ such that $l = |\{h\#n^j \in \sigma \mid h\#n^j == k\#l^i\}|$.*

*Moreover, assume that there exist two distinct nodes $n$ and $n'$ in $\sigma$ such that $n'$ is a node in $T_\delta(n)$, $A_{F_\delta}(n) == k\#l^i$, $A_{F_\delta}(n') == k'\#{l'}^{i'}$ and $\rho$ is a renaming such that $S_{F_\delta}(n) = S_{F_\delta}(n')\rho$ and $k = k'\rho$.*

*Then there exists a terminating computation $\delta'$ for the goal $G$ in the program $P$, such that either $F_{\delta'}$ is $l'$-repetitive with $l' < l$, or $F_{\delta'}$ is $l$-repetitive and $dg(F'_\delta) < p$.*

*Proof.* Let $A_{F_\delta}(n) = k_s\#{l_s}^{i_s}$ and let $A_{F_\delta}(n') = k_t\#{l_t}^{i_t}$. From $F_\delta$ we can construct a new forest $F'$ by replacing the subtree $T_\delta(n)$ with the subtree $T' = T_\delta(n')$ and we can define two functions $A_{F'}$ and $S_{F'}$ such that for each node $v$ in $F'$

- if $v \notin T'$, then $A_{F'}(v) = A_{F_\delta}(v)$ and $S_{F'}(v) = S_{F_\delta}(v)$,
- if $v \in T'$ and $A_{F_\delta}(v) = k\#{l'}^{i'}$, with $l \neq l_t$, then $A_{F'}(v) = A_{F_\delta}(v)\rho$ and $S_{F'}(v) = S_{F_\delta}(v)\rho$,
- if $v \in T'$ and $A_{F_\delta}(v) = k_t\#{l_t}^i$, then $A_{F'}(v) = k_s\#{l_s}^{i-i_t+i_s}$ and $S_{F'}(v) = S_{F_\delta}(v)\rho$.

By construction since $F'$ is obtained from $F_\delta$ by deleting at least a constraint $h\#l'^z$ such that $h'\#l'^z == k\#l^i$ (and therefore $|\{h\#n^j \in d \mid h\#n^j == h'\#l'^z\}| = l$) we have that either $F'$ is $l'$-repetitive with $l' < l$, or $F'$ is $l$-repetitive and $dg(F') < p$. Moreover, since $S_{F_\delta}(n) = S_{F_\delta}(n')\rho$, all the possible interactions between the constraints of input–output in $T_\delta(n)$ follow the same pattern of interactions between the constraints of input-output in $T_\delta(n')$. Then there exists a terminating computation $\delta'$ for the goal $G$ in the program $P$ such that $\delta'$ obtained from $\delta$ by replacing all the applications of rule in $T_\delta(n)$ with only the rules in $T_\delta(n')$. Therefore, by construction, $F_{\delta'} = F'$, $A_{F_{\delta'}} = A_{F'}$, and $S_{F_{\delta'}} = S_{F'}$ (up to renaming of identifiers) and then the thesis.

## A.6 Theorem 6.1

**Theorem 6.1** *The triple* $(\alpha(), \mathcal{INP}(), \mathcal{OUT}())$ *provides an acceptable encoding from* $static\ CHR^{\omega_p}$ *into* $static\ CHR_2^{\omega_p}$.

*Proof.* By definition, we have to prove that for all $static\ CHR^{\omega_p}$ programs $P$ and goals $G$,

$$\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\alpha(P)}(\mathcal{INP}(G)))$$

holds.

By construction, the functions $\mathcal{INP}()$ and $\mathcal{OUT}()$ are compositional and defined as

$$\mathcal{INP}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

$$\mathcal{OUT}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = new_{\text{ak}}(V, \bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = ak(\bar{t}). \end{cases}$$

Let $P$ be a $static\ CHR^{\omega_p}$ program and let $G$ be a goal. From definition of $\mathcal{INP}()$, we have that the predicate symbols $id$, $end$, $rC[N]_i$, $rA_i$, $new_k$ (with $k \in \mathcal{INP}(Head(P))$) cannot be in the encoded goal $\mathcal{INP}(G)$.

Therefore if $G = \emptyset$ or $G$ does not contain predicate symbols that are in $Head(P)$, we have that $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\alpha(P)}(\mathcal{INP}(G)))$ since no rule from both the programs can be applied. If, however, the goal $G$ contains a predicate symbol in $Head(P)$ (and therefore $\mathcal{INP}(G)$ contains a predicate symbol in $\mathcal{INP}(Head(P))$) then $rule_{(2,k)} \in \alpha(P)$ is fired first. At this point all the constraints $k(\bar{t})$ in $G$, such that $k \in Head(P)$, are transformed by rules $rule_{(1,k)}$ into the constraint $new_{\mathcal{INP}(k)}(n, \bar{t})$ in $\mathcal{INP}(G)$, where $n$ is a unique identifier (intuitively this identifier can be considered as the identifier assigned to the original constraint by the Introduce transition step). Let us define the mapping between the original constraint $k(\bar{t})$ with the corresponding $n$ identifier of the $new_{\mathcal{INP}(k)}(n, \bar{t})$ constraint as $\varphi$.

After this phase, we obtain a new goal $G'$ in $\alpha(P)$ and the rules $rule_{(2,k)}$ and $rule_{(1,k)}$ are no longer used in this derivation in $\alpha(P)$. Since there is no $end$ or $rA_i$ predicate symbol in $G'$, the next rules that are applied in the derivation in $\alpha(P)$ are rules $rule'_{(i,N)}$. By definition of these rules a constraint $rC[N]_i(V_1, \ldots, V_N, \bar{t})$ is generated if in the original program the constraints $\varphi^{-1}(V_1), \ldots, \varphi^{-1}(V_N)$ in $G$ can be used as a match for the application of the rule $rule_i$ in $P$. Thus a constraint $rC[r_i]_i(\bar{V}, \bar{t})$ is created for every possible match of constraints that can fire rule $rule_i$.

When all the possible $rule'_{(i,N)}$ have fired there are two possibilities:

1. if in the original program a rule can fire than at least one rule $rule_{(7,i)}$ fires. The firing of this rule corresponds to the firing of a rule $rule_i$ in the original program. For every constraint $k(\bar{t})$ in the body of the original rule a new $new_{\mathcal{INP}(k)}(V, \bar{t})$ constraint is added to the store of the derivation in the encoded program, with its new unique identifier $V$. This rule also adds to the store the constraint $rA_i(\bar{V})$ where $\bar{V}$ are the identifiers $\varphi(k(\bar{t}))$ of all the constraints $k(\bar{t})$ that are removed from the store by the application of $rule_i$ in the original program $P$. The removal of this constraints in the encoded program is done by rules $rule_{(4,i,k)}$ that are eventually fired immediately after rule $rule_{(7,i)}$. Rules $rule_{(5,j,i,k)}$ are then fired for removing all the constraints $rC[N]_i$ that have no more sense to exist since one of the constraints identified by their arguments has been removed. After that, the constraint $rA_i(\bar{V})$ is no longer useful and it is removed by rule $rule_{(6,i)}$. When the constraint $rA_i(\bar{V})$ is removed other rules $rule'_{(i,N)}$ can fired (new $new_{\mathcal{INP}(k)}(\_)$ constraints have potentially been added to the store by $rule_{(7,i)}$) repeating the cycle.
2. if in the original program no rule can fire then no rule $rule_{(7,i)}$ in $\alpha(P)$ can fire and therefore $rule_8$ fires. This removes the constraint $id$ and adds the constraint $end$ that triggers the rules $rule_{(3,i,N)}$. These rules remove all the constraints $rC[N]_i$ and when all these constraints are removed the $end$ constraint is removed too by $rule_9$. After the firing of this rule, no rule of the program can fire anymore.

For every rule $rule_i$ that can fire in the original program there is a corresponding rule $rule_{(7,i)}$ that can fire in the encoded program. Moreover, for every CHR constraint $k(\bar{t})$ in every configuration during the execution of the goal $G$ in $P$ we have two possibilities. If $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal $G$, then there is a $\mathcal{INP}(k)(\bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\alpha(P)$. If $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition step, then there is a $new_{\mathcal{INP}(k)}(V, \bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\alpha(P)$. The built-in constraints are not modified and are processed in the same way by both the programs.

When the encoded program terminates no $id$, $end$, $rC[N]_i$, and $rA_i$ are in the store.[1] Hence, applying the decoding function to the qualified answer of the encoded program produces the equivalent qualified answer of the original program.

[1] Technically speaking rules $rule_{(3,i,N)}$, $rule_8$ and $rule_9$ are not needed because the constraint can be removed using the decoding function. We chose to add them to exploit the same encoding also for the Theorem 6.2

## A.7 Lemma 6.1

**Lemma 6.1** *Let $P$ be a static $CHR^{\omega_p}$ program and let $q$ be a predicate symbol. For every goal $G$, if $G$ does not contain the predicate symbol $q$ then $\mathcal{SA}_P(G) = \mathcal{SA}_{\beta(P,q)}(G)$, $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$ otherwise.*

*Proof.* By our assumptions $start$ and $init$ are not contained in $Head(P)$. Moreover, by construction, $f$ is a function that maps predicate symbols into fresh predicate symbols (i.e., not in $Pred(P) \cup \{start, init, q\}$).

The proof is by cases on the form of the goal $G$.

If $G = \emptyset$ or $G$ does not contain predicate symbols that are in $Head(P)$, we have that $\mathcal{SA}_P(G) = \emptyset$. Moreover, since in $\beta(P, q)$ there is no rule which produces an atom of the form $k(\bar{t})$, with $k \in Head(P)$, we have that $rule_{(m+9,k)}$ cannot be used and therefore $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$.

Now, let us to assume that the goal $G$ contains a predicate symbol in $Head(P)$. We have the following cases:

$(G = start, G')$ In this case, since by our assumptions $start \notin Head(P)$ we have that $\mathcal{SA}_P(G) = \emptyset$.

Moreover, we have the following possibilities:

1. $(init \in G'$ **or** $q(\_) \in G'$ **or** $f(k)(\_) \in G'$, **with** $k \in Head(P))$. In this case

$$\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrow^{\omega_p}_{\beta(P,q)}{}^{*} \langle \emptyset, G'', false, T\rangle_n$$

   by using one of the three clauses with priority 1. Therefore, $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.
2. $(init \notin G',\ q(\_) \notin G',\ f(k)(\_) \notin G'$ **with** $k \in Head(P)$, **and** $start \in G')$. In this case

$$\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrow^{\omega_p}_{\beta(P,q)}{}^{*} \langle \emptyset, (G'', start\#l, start\#p), B, T\rangle_k \rightarrow^{\omega_p}_{\beta(P,q)}$$
$$\langle \emptyset, (G'', start\#l, init\#n), B, T'\rangle_{n+1} \rightarrow^{\omega_p}_{\beta(P,q)}{}^{*} \langle \emptyset, G'', false, T''\rangle_{n+1}$$

   by using in the order the rules $rule_{m+4}$ and $rule_{m+3}$ and therefore $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.
3. $(G' = k_1(\bar{t}_1), \ldots, k_r(\bar{t}_r)$, **with** $k_i \in Head(P)$, **for** $i = 1, \ldots, r)$. In this case, after some **Solve** and **Introduce** transition steps and an **Apply** transition step we have that

$$\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrow^{\omega_p}_{\beta(P,q)}{}^{*} \langle \emptyset, G', B, T\rangle_n$$

where

- either $G'$ is of the form $(G'', start\#l, start\#p)$ if the **Apply** transition step uses $rule_{(m+5,k)}$

- or $G'$ is of the form $(G'', start\#l,\ f(k'(\bar{t}'))\#p)$ if the **Apply** transition step uses a rule $rule_{(m+6,k,k')}$.
  By using the same arguments of the cases 1 and 2, we have that $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.

$(start \notin G)$ We have two further cases.

1. ($G$ **contains an atom of the form** $init$ **or** $f(k(\bar{t}))$ **with** $k \in Head(P)$ ). Since by our hypothesis $init,\ f(k) \notin Head(P)$ we have that $\mathcal{SA}_P(G) = \emptyset$. Moreover, since $G$ contains at least an atom of the form $k(\bar{t})$, with $k \in Head(P)$, it is easy to check that

$$\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrow_{\beta(P,q)}^{\omega_p} {}^{*} \langle \emptyset, (G', start\#p), B, T\rangle_n$$

   by using some **Apply** transition steps with $rule_{(m+6,k,k')}$ and then an **Apply** transition step with $rule_{(m+5,k)}$, where $G'$ contains an atom of the form $init$ or $f(k(\bar{t}))$ with $k \in Head(P)$. In this case, analogously to point 1 of the case $(G = start, G')$, we have that the derivation ends in a failed configuration and then $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.
2. ($G$ **contains an atom of the form** $q(\bar{t})$). Then, analogously to the previous point, we have that the derivation of $G$ in $\beta(P, q)$ ends in a failed configuration and then $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$.
3. $(G = k_1(\bar{t_1}), \ldots, k_n(\bar{t_n}))$. Let us consider a derivation $\delta$ for $G$ in $\beta(P, q)$. We distinguish two cases:

   (the first **Apply** transition step uses a rule $rule_{(m+6,k,k')}$). In this case, analogously to point 3 of the case **(start** $\in$ **G)**, we have that $\delta$ ends in a failed configuration.
   (the first **Apply** transition step uses a rule $rule_{(m+5,k)}$). Without loss of generality, we can assume that $rule_{(m+5,k_l)}$ rewrites an atom of the form $k_l(\bar{t_l})$. Then we have that

$$\begin{aligned}\delta = \langle G, \emptyset, true, \emptyset\rangle_1 &\rightarrow_{\beta(P,q)}^{\omega_p} {}^{*} \langle \emptyset, G', B, \emptyset\rangle_n \rightarrow_{\beta(P,q)}^{\omega_p} \\ &\langle \emptyset, (G', start\#n), B, \{[s, rule_{(m+5,k_l)}]\}\rangle_{n+1} \\ &\langle \emptyset, (G', init\#n + 1), B, \{[s, rule_{(m+5,k_l)}],\ [n, rule_{m+4}]\}\rangle_{n+2} \cdot \delta'\end{aligned}$$

   Now, we have two further possibilities.

   a. There exists an atom in $G'$, which is rewritten by using a clause $rule_{(m+5,j)}$. In this case

$$\begin{aligned}\delta' = \langle \emptyset, (G', init\#n + 1), B, \{[s, rule_{(m+5,k_l)}],\ [n, rule_{m+4}]\}\rangle_{n+2} \\ \rightarrow_{\beta(P,q)}^{\omega_p} {}^{*} \langle \emptyset, (G'', init\#n + 1, start\#n'), B', T'\rangle_{n'+1} \\ \rightarrow_{\beta(P,q)}^{\omega_p} {}^{*} \langle \emptyset, G_1, false, T'\rangle_{n'+1}\end{aligned}$$

      where the last **Apply** transition step uses either $rule_{(m+2,k)}$ or $rule_{m+3}$

b. There exists no atom in $G'$, which is rewritten by using a clause $rule_{(m+5,k)}$. In this case

$$\langle \emptyset, (G', init\#n+1), B, \{[s, rule_{(m+5,k_l)}],\ [n, rule_{m+4}]\}\rangle_{n+2}$$
$$\rightarrow_{\beta(P,q)}^{\omega_p\ *} \langle \emptyset, (G'', init\#n+1, k_l(\bar{t}_l)\#s, B', T'\rangle_{n''}$$

where $\mathtt{chr}(G'') = f(k_1(\bar{t}_1)), \ldots, f(k_n(\bar{t}_n))$, all the **Apply** transition steps except the last one use one of the rules $rule_{(m+6,k,k')}$, $[s, rule_{(m+5,k_l)}] \in T'$ and the last **Apply** transition step rewrites the atom $k_l(\bar{t}_l)\#s$ by using the rule $rule_{(m+7,k_l)}$ (and therefore $[s, rule_{(m+7,k_l)}] \in T'$). From this point the only applicable rules are $rule'_i$, $rule_{(m+8,k)}$ and $rule_{(m+9,k)}$.

Then the proof is immediate by previous results and by definition of $rule_i$, $rule'_i$, $rule_{(m+8,k)}$ and $rule_{(m+9,k)}$.

## A.8 Theorem 6.3

**Theorem 6.3** *The triple ($\mathcal{T}()$, $\mathcal{INP}()$, $\mathcal{OUT}()$) provides an acceptable encoding between $CHR^{\omega_p}$ and static $CHR^{\omega_p}$.*

*Proof.* By definition, we have to prove that for all $CHR^{\omega_p}$ programs $P$ and goals $G$,

$$\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{T}(P)}(\mathcal{INP}(G)))$$

holds.

By construction, the functions $\mathcal{INP}()$ and $\mathcal{OUT}()$ are compositional and defined as:

$$\mathcal{INP}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

$$\mathcal{OUT}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = \text{new}_{\text{ak}}(V,\bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = \text{ak}(\bar{t}). \end{cases}$$

Let $P$ be a $CHR^{\omega_p}$ program and let $G$ be a goal.

For the definition of $\mathcal{INP}()$, we have that the constraints $start$, $id$, $end$, $instance_i$, $highest\_priority$, $new_{\text{ak}}$ (where $ak \in \mathcal{INP}(Head(P))$) cannot be in the encoded goal $\mathcal{INP}(G)$.

Therefore if $G = \emptyset$ or $G$ does not contain constraints that are in $Head(P)$ we have that $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{T}(P)}(\mathcal{INP}(G)))$ since no rule from both the programs $P$ and $\mathcal{T}(P)$ can be applied. If, however, the goal $G$ contains a constraint in $Head(P)$ then $rule_{(2,k)}$ is fired first. At this point each constraint $ak(\bar{t})$ in $\mathcal{INP}(G)$ (corresponding to a constraint $k(\bar{t})$ in $G$) such that $k \in Head(P)$ is transformed by rules $rule_{(1,k)}$ into the constraint $new_{\text{ak}}(V, \bar{t})$ where $V$ is a unique identifier (intuitively,

this identifier can be considered as the identifier assigned to the original constraint by the Introduce transition step). Let us define the mapping between the original constraint $k(\bar{t}) \in G$ with the corresponding $V$ identifier of the $new_{\text{ak}}$ constraint as $\varphi$.

After this phase the rules $rule_{(2,k)}$ and $rule_{(1,k)}$ are no longer used in a derivation in $\mathcal{T}(P)$ and the configuration $S$ is generated. Since there is no $start$, $end$, or $instance_i$ constraint in $S$, (they cannot be in the encoded goal $\mathcal{INP}(G)$ due to the goal encoding function) the next rules that are applied in $\mathcal{T}(P)$ are rules in EVALUATE_PRIORITIES($i$). By definition of these rules, if in the original program $P$ it is possible to fire the $j$-th rule starting from $G$, the constraint $highest\_priority(p_j)$ can added to the CHR store of $S$ in $\mathcal{T}(P)$ after all the possible rules in
EVALUATE_PRIORITIES($i$) have fired.

Note that, after all the possible rules in EVALUATE_PRIORITIES($i$) (for $i = 1, \ldots, n$) have fired at most a constraint $highest\_priority(p_j)$ is present in the constraint store.

When all the possible EVALUATE_PRIORITIES($i$) (for $i = 1, \ldots, n$) have fired there are two possibilities:

1. if in the original program a rule can fire then at least one rule ACTIVATE_RULE($i$) (for $i = 1, \ldots, n$) fires. The firing of the rule $rule_{(10,j)}$ in $\mathcal{T}(P)$ corresponds to the firing of the $j$-th rule in the original program $P$. Moreover, the application of the rule $rule_{(10,j)}$ in $\mathcal{T}(P)$ uses the atoms $p_1(V_1, \bar{t}_1), \ldots, p_m(V_m, \bar{t}_m), highest\_priority(p_j), id(l)$ if and only if in the original program $rule_j$ in $P$ can fire by using the atoms $\varphi^{-1}(V_1), \ldots, \varphi^{-1}(V_m)$. For every constraint $k(\bar{t})$ in the body of the original rule a $new_{\text{ak}}(V, \bar{t})$ constraint is added with its new unique identifier $V$. This rule also adds to the store the constraint $highest\_priority(inf)$ and then the computation starts from EVALUATE_PRIORITIES($i$) (for $i = 1, \ldots, n$), repeating the cycle.
2. if in the original program no rule can fire then no rule EVALUATE_PRIORITIES ($i$) (for $i = 1, \ldots, n$) can fire and therefore $rule_9$ fires. This removes the constraints $highest\_priority(inf)$ and $id(V)$ from the constraint store. It also adds the constraint $end$ that triggers the rules $rule_{(4,i)}$ (for $i = 1, \ldots, n$). These rules remove all the constraints $instance_i(\bar{V})$ and when all these constraints are removed the $end$ constraint is alto removed by $rule_5$. After the firing of this rule no rule of the program can fire anymore.

For every rule $rule_i$ that can fire in the original program $P$ there is a corresponding rule $rule_{(10,i)}$ that can fire in the encoded program $\mathcal{T}(P)$. Moreover for every CHR constraint $k(\bar{t})$ in every configuration during the execution of the goal $G$ in $P$ we have two possibilities. If $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition step then there is a $new_{\text{ak}}(V, \bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$. If $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal $G$ then there is a $ak(\bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$. The built-in constraints are not modified and are processed in the same way by both the programs. When the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$ terminates

no $id$, $start$, $highest\_priority$, $end$, $instance_i$ are in the store.[2] Hence applying the decoding function to the qualified answer of the encoded program produces the equivalent qualified answer of the original program.

## A.9 Theorem 9.1

**Theorem 9.1** *If $W \subseteq \mathcal{P}(V)$ the result of $radix\_trees(W)$ is a graph containing at maximum $\binom{n}{\lceil n/2 \rceil}$ maximal paths. Hence the algorithm produces at maximum $\binom{n}{\lceil n/2 \rceil}$ radix trees schemas.*

*Proof.* The worst case scenario is obtained when $W = \mathcal{P}(V)$. In this case we have that:

- $|level_W(i)| \leq |level_W(i+1)|$ for every $i \in [0, \lfloor n/2 \rfloor]$
- $|level_W(i-1)| \geq |level_W(i)|$ for every $i \in [0, \lceil n/2 \rceil]$

If $i \in [0, \lfloor n/2 \rfloor]$ and we are computing the maximal matching between $radix\_trees(level_W(0) \cup \cdots \cup level_W(i))$ and $level_W(i+1)$, every node of $level_W(i)$ will have an outgoing arc. At the same time if instead $i \in [\lceil n/2 \rceil, n]$ we have that a node of $level_W(i)$ has an incoming arc. This means that the number of the non-extensible paths is equal to the cardinality of the level having more elements. Since $\lceil n/2 \rceil$ level is the level with the maximal number of elements (this is a basic combinatoric result, see [36] for more details) and $|level_W(\lceil n/2 \rceil)| = \binom{n}{\lceil n/2 \rceil}$ we have the thesis.

## A.10 Theorem 9.2

**Theorem 9.2** *Let $m$ be a message and $V_1, \ldots, V_k$ be all the subsets of c-set variables that are defined by all the possible c-tokens. Then there exists a radix tree schema produced by $radix\_trees(\{V_1, \ldots, V_k\})$ which allows us to check if the message correlates with a session.*

*Proof.* Since every node of the graph is covered at least by a path having length 0 we have that for every set $V_i$ there exist a radix tree schema $RT(seq_1, \ldots, seq_m)$ covering it. $RT(seq_1, \ldots, seq_m)$ is therefore the schema required.

## A.11 Theorem 9.3

**Theorem 9.3** *The graph produced by $radix\_trees(P)$ contains the minimal number of maximal paths covering all the nodes in $P$.*

---

[2] Technically speaking rules $rule_{(4,i)}$, $rule_5$ and $rule_9$ are not needed because the constraint can be removed using the decoding function.

*Proof.* The proof is by induction on the number of the levels of $P$. First of all, let us underline that the minimal number of maximal paths covering all the nodes in a graph never decreases when new nodes are added to the graph.

Let us suppose that the first nonempty level is $k_0$ (i.e., the smallest set of $P$ has $k_0$ elements).

- if $i < k_0$ the propriety is trivially satisfied because $level_P(0) \cup \cdots \cup level_P(i) = \emptyset$.
- if $i = k_0$ $radix\_trees(level_P(k_0))$ will be the graph containing only the nodes of level $k_0$. Since there are no arcs between these nodes $radix\_trees(level_P(k_0))$ produces the optimal solution (in this case for covering the $level_P(k_0)$ we need $|level_P(k_0)|$ paths of length 0).
- if $i > k_0$ let suppose for inductive hypothesis that $(V, E) = radix\_trees(level_P(0) \cup \cdots \cup level_P(i-1))$ produces the minimal amount of maximal walks covering all the nodes in $level_P(0) \cup \cdots \cup level_P(i-1)$. A maximal matching is computed to select the arcs added to the $radix\_trees(level_P(0) \cup \cdots \cup level_P(i))$. This maximal matching is performed on the bipartite graph between the nodes of $V$ not having an outgoing arc and the nodes of level $i$. Now:
  - if $V'$ are the nodes of level $i$ having an incoming arc and $E'$ are the arcs selected by the matching algorithm arriving in a $V'$ node we have that the graph $(V \cup V', E \cup E')$ is optimal since it has the same number of maximal paths of the graph $(V, E)$
  - if $V''$ are the nodes of level $i$ that have no incoming arc because no node of $V''$ has a subset in $V$ we have that for covering a $V''$ node a new path of length 0 is required. The graph $(V \cup V' \cup V'', E \cup E')$ is therefore optimal
  - if $V'''$ are the remaining nodes of the $i$ level we have that $u$ belongs to $V'''$ iff $u$ has no incoming arcs and there exists a node $v \in V$ that is a subset of $u$. This means that the arc $(v, u)$ has not been selected by the maximal matching algorithm and that for every node $v'$ that is a subset of $u$ there is a node $u' \in V'$ s.t. $(v', u') \in E \cup E'$. Since there are no arcs between nodes of the same level we have that for covering $u$ a new path is required. Therefore $(V \cup V' \cup V'' \cup V''', E \cup E')$ is one of the optimal solutions Since $radix\_trees(level_P(0) \cup \cdots \cup level_P(i)) = (V \cup V' \cup V'' \cup V''', E \cup E')$ we have proven our initial claim.

# References


1. Fourth International CSP Solver Competition. http://www.cril.univ-artois.fr/CPAI09/ (2012). Accessed 1 Feb 2012
2. JSR 331: Constraint Programming API. http://jcp.org/en/jsr/summary?id=331 (2012). Accessed 1 Feb 2012
3. MiniZinc Challenge 2011. http://www.g12.csse.unimelb.edu.au/minizinc/challenge2011/challenge.html (2011). Accessed 1 Feb 2012
4. Standardization of Constraint Programming. http://4c110.ucc.ie/cpstandards/ (2012). Accessed 1 Feb 2012
5. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (2012). Accessed 1 Feb 2012
6. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96), pp. 313–321 (1996)
7. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference (AFIPS '67 Spring), April 18–20, New York (1967), pp. 483–485. http://doi.acm.org/10.1145/1465482.1465560
8. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Proceedings of the Conference on the Principles and Practice of Constraint Programming, Lisbon, Portugal, pp. 142–157 (2009)
9. Barros, A.P., Decker, G., Dumas, M., Weber, F.: Correlation patterns in service-oriented architectures. In: Fundamental Approaches to Software Engineering (FASE). Braga, Portugal, pp. 245–259 (2007)
10. Barták, R.: Constraint programming. In Pursuit of the Holy Grail. In: Proceedings of Week of Doctoral Students (WDS99), pp. 555–564 (1999)
11. Becket, R.: Specification of flatzinc. version 1.3. http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-1.3/flatzinc-spec.pdf (2012). Accessed 1 Feb 2012
12. Ben, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, Boston (2006)
13. Betz, H.: Relating coloured Petri nets to constraint handling rules. In: Djelloul, K., Duck, G.J., Sulzmann, M. (eds.) 4th Workshop on Constraint Handling Rules. Porto, Portugal, pp. 33–47 (2007)
14. Betz, H., Raiser, F., Frühwirth, T.W.: A complete and terminating execution model for constraint handling rules. Theor. Prac. Logic Program. **10**(4–6), 597–610 (2010)
15. Bistarelli, S., Frühwirth, T.W., Marte, M.: Soft constraint propagation and solving in CHRs. In: Symposium on Applied Computing (SAC), pp. 1–5 (2002)



DOI: 10.2991/978-94-6239-067-6, © Atlantis Press and the authors 2014

16. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Trans. Comput. Log. **7**(3), 563–589 (2006)
17. Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language for SLA negotiation. Electr. Notes Theor. Comput. Sci. **236**, 147–162 (2009)
18. de Boer, F.S., Gabbrielli, M., Meo, M.C.: A timed concurrent constraint language. Info. Comput. **161**(1), 45–83 (2000)
19. de Boer, F.S., Gabbrielli, M., Meo, M.C.: A temporal logic for reasoning about timed concurrent constraint programs. In: Time, 227–233 (2001)
20. de Boer, F.S., Palamidessi, C.: On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In: International Conference on Concurrency Theory (CONCUR). Amsterdam, Netherlands, pp. 99–114 (1990)
21. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. Info. Comput. **108**(1), 128–157 (1994)
22. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI). Pasadena, California, pp. 443–448 (2009)
23. Boreale, M., Bruni, R., Nicola, R.D., Loreti, M.: Sessions and pipelines for structured service programming. In: International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08). Oslo, Norway. LNCS, vol. 5051, pp. 19–38. Springer (2008)
24. Borning, A., Duisberg, R., Freeman-Benson, B.N., Kramer, A., Woolf, M.: Constraint hierarchies. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). Portland, Oregon, pp. 48–60 (1987)
25. Buscemi, M.G., Montanari, U.: CC-Pi: A constraint-based language for specifying service level agreements. In: European Symposium on Programming (ESOP). Braga, Portugal, pp. 18–32 (2007)
26. Buscemi, M.G., Montanari, U.: Open bisimulation for the concurrent constraint pi-calculus. In: S. Drossopoulou (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 4960, pp. 254–268. Springer (2008)
27. Buscemi, M.G., Montanari, U.: A survey of constraint-based programming paradigms. Comput. Sci. Rev. **2**(3), 137–141 (2008)
28. Busi, N., Gabbrielli, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: International Colloquium on Automata, Languages and Programming (ICALP). Turku, Finland, pp. 307–319 (2004)
29. Carletta, J.: Assessing agreement on classification tasks: the kappa statistic. CoRR cmp-lg/9602004 (1996)
30. Carzaniga, A., Wolf, A.L.: Forwarding in a content-based network. In: SIGCOMM Conference. Karlsruhe, Germany, pp. 163–174 (2003)
31. Di Giusto, C., Gabbrielli, M., Meo, M.C.: Expressiveness of multiple heads in CHR. CoRR abs/0804.3351 (2008). To appear in TOCL
32. Di Giusto, C., Gabbrielli, M., Meo, M.C.: Expressiveness of multiple heads in CHR. In: Proceedings of the SOFSEM, Spindleruv Mlyn, Czech Republic, pp. 205–216 (2009)
33. Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: International Conference on Logic Programming (ICLP). Saint Malo, France, pp. 90–104 (2004)
34. Elfatatry, A.: Dealing with change: components versus services. Commun. ACM **50**(8), 35–39 (2007)
35. Eoin O'Mahony Emmanuel Hebrard, A.H.C.N., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
36. Erickson, M.J.: Introduction to combinatorics, series in discrete mathematics and optimization, vol. 42. Wiley-Interscience, Las Vegas (1996)
37. Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe. In: SIGMOD Conference. Santa Barbara, California, pp. 115–126 (2001)

38. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theor. Comput. Sci. **256**(1–2), 63–92 (2001)
39. Freuder, E.C.: Synthesizing constraint expressions. Commun. ACM **21**(11), 958–966 (1978)
40. Freuder, E.C., Wallace, R.J.: Partial constraint satisfaction. Artif. Intell. **58**(1–3), 21–70 (1992)
41. Frühwirth, T.: Temporal reasoning with constraint handling rules. Technical report ECRC-94-5, European Computer-Industry Research Centre, Munchen, Germany (1994)
42. Frühwirth, T.: Constraint handling rules (2009). http://www.constraint-handling-rules.org
43. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Logic. Program. **37**(1–3), 95–138 (1998)
44. Frühwirth, T.W.: As time goes by II: more automatic complexity analysis of concurrent rule programs. Electr. Notes Theor. Comput. Sci. **59**(3) (2001)
45. Frühwirth, T.W.: As time goes by: automatic complexity analysis of simplification rules. In: International Conference on Knowledge Representation. Toulouse, France, pp. 547–557 (2002)
46. Frühwirth, T.W., Abdennadher, S.: The munich rent advisor: a success for logic programming on the internet. Theor. Pract. Logic Program. **1**(3), 303–319 (2001)
47. Frühwirth, T.W., Brisset, P.: Optimal placement of base stations in wireless indoor telecommunication. In: International Conference on Constraint Programming. Udine, Italy, pp. 476–480 (1998)
48. Gabbrielli, M., Mauro, J., Meo, M.C.: On the expressive power of priorities in CHR. In: Principle and Practice of Declarative Programming (PPDP). Coimbra, Portugal, pp. 267–276 (2009)
49. Gabbrielli, M., Mauro, J., Meo, M.C., Sneyers, J.: Decidability properties for fragments of CHR. Theor. Pract. Logic Program. **10**(4–6), 611–626 (2010)
50. Gabbrielli, M., Palamidessi, C., Valencia, F.D.: Concurrent and reactive constraint programming. In: 25 Years GULP, pp. 231–253 (2010)
51. Gallaire, H.: Logic programming: further developments. In: Symposium on Logic Programming. Boston, Massachusetts, pp. 88–96 (1985)
52. Gaschnig, J.G.: Performance measurement and analysis of certain search algorithms. Ph.D. thesis, Carnegie-Mellon University (1979)
53. Gebruers, C., Hnich, B., Bridge, D.G., Freuder, E.C.: Using CBR to select solution strategies in constraint programming. In: International Conference on Case-Based Reasoning. Chicago, Illinois, pp. 222–236 (2005)
54. Glover, F.: Tabu search-part I. INFORMS J. Comput. **1**(3), 190–206 (1989)
55. Glover, F.: Tabu search-part II. INFORMS J. Comput. **2**(1), 4–32 (1990)
56. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
57. Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In: European Conference on Artificial Intelligence (ECAI). Valencia, Spain, pp. 475–479 (2004)
58. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: International Conference on Application of Concurrency to System Design (ACSD'08). Xi'an, China, pp. 190–198. IEEE Computer Society Press (2008)
59. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamentae Informaticae **95**(1), 73–102 (2009)
60. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A calculus for service oriented computing. In: International Conference on Service Oriented Computing (ICSOC). Berlin, Germany (2006)
61. Guidi, C., Montesi, F.: Reasoning about a service-oriented programming paradigm. In: YR-SOC, pp. 67–81 (2009)
62. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM **31**(5), 532–533 (1988)
63. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. **14**(3), 263–313 (1980)
64. Holzmann, G.J.: Formal software verification: how close are we? In: J. Hatcliff, E. Zucca (eds.) FMOODS/FORTE, Lecture Notes in Computer Science, vol. 6117, p. 1. Springer (2010)

65. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. (JAIR) **36**, 267–306 (2009)
66. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Symposium on Principles of Programming Languages (POPL). Orlando, Florida, pp. 111–119 (1987)
67. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Logic. Program. **19**(20), 503–581 (1994)
68. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable distributed depth-first search with greedy work stealing. In: International Conference on Tools with Artificial Intelligence (ICTAI 2004). Boca Raton, Florida, pp. 98–103 (2004)
69. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC-instance-specific algorithm configuration. In: European Conference on Artificial Intelligence (ECAI). Valencia, Spain, pp. 751–756 (2010)
70. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artif. Intell. **45**(3), 275–286 (1990)
71. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The orc programming language. In: FMOODS/FORTE'09, LNCS, vol. 5522, pp. 1–25. Springer (2009)
72. Kiziltan, Z., Mandrioli, L., Mauro, J., O'Sullivan, B.: A classification-based approach to managing a solver portfolio for CSPs. In: Proceedings of the 22th Irish Conference on Artificial Intelligence and Cognitive Science (2011)
73. Kiziltan, Z., Mauro, J.: Service-oriented volunteer computing for massively parallel constraint solving using portfolios. In: A. Lodi, M. Milano, P. Toth (eds.) CPAIOR, Lecture Notes in Computer Science, vol. 6140, pp. 246–251. Springer (2010)
74. Koestler, A.: The sleepwalkers: a history of man's changing vision of the universe. Penguin, New York (1990)
75. Koninck, L.D., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: Principles and Practice of Declarative Programming. Wroclaw, Poland, pp. 25–36 (2007)
76. Kumar, V.: Algorithms for constraint-satisfaction problems: a survey. AI Mag. **13**(1), 32–44 (1992)
77. Laburthe, F., Jussien, N., Rochart, G., Cambazard, H., Prud'homme, C., Malapert, A., Menana, J.: Choco solver. http://www.emn.fr/z-info/choco-solver/index.html. Accessed 01 Feb 2012
78. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Foundations of Software Science and Computation Structure (FoSSaCS). LNCS, vol. 3441, pp. 282–298. Springer (2005)
79. Laneve, C., Zavattaro, G.: Web-pi at work. In: Trustworthy Global Computing (TGC'05), LNCS, vol. 3705, pp. 182–194. Springer (2005)
80. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: European Symposium on Programming (ESOP). Braga, Portugal. LNCS, vol. 4421, pp. 33–47. Springer (2007)
81. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: a survey. Oper. Res. **14**(4), 699–719 (1966)
82. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In: Principles and Practice of Constraint Programming, pp. 556–572 (2002)
83. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**(1), 99–118 (1977)
84. Maher, M.J.: Logic semantics for a class of committed-choice programs. In: International Conference on Logic Programming (ICLP). Melbourne, Victoria, pp. 858–876 (1987)
85. Mauro, J., Gabbrielli, M., Guidi, C., Montesi, F.: An efficient management of correlation sets with broadcast. In: W.D. Meuter, G.C. Roman (eds.) COORDINATION, Lecture Notes in Computer Science, vol. 6721, pp. 80–94. Springer (2011)
86. Mauro, J., Gabbrielli, M., Guidi, C., Montesi, F.: An efficient management of correlation sets with broadcast. Technical report (2011). www.cs.unibo.it/~jmauro/papers/tech_report_coordination_2011 (2011). Accessed 1 Feb 2012
87. McDermott, J.P., Nilsson, N.J.: Principles of artificial intelligence. Artif. Intell. **15**(1–2), 127–131 (1980)

88. Michel, L., See, A., Hentenryck, P.V.: Transparent parallelization of constraint programming. INFORMS J. Comput. **21**(3), 363–382 (2009)
89. Mila, D.P., Maurizio, G., Ivan, L., Jacopo, M., Zavattaro, G.: Graceful interruption of request-response service interactions. In: International Conference on Service Oriented Computing (ICSOC). Paphos, Cyprus (2011)
90. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. I. Info. Comput. **100**(1), 1–40 (1992)
91. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. II. Info. Comput. **100**(1), 41–77 (1992)
92. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artif. Intell. **58**(1–3), 161–205 (1992)
93. Mitchell, T.M.: Machine Learning. McGraw-Hill, McGraw Hill series in computer science, New York (1997)
94. Montesi, F., Guidi, C.: Jolie project. http://www.jolie-lang.org/ (2012). Accessed 1 Feb 2012
95. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a java orchestration language interpreter engine. Electr. Notes Theor. Comput. Sci. **181**, 19–33 (2007)
96. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: European Conference on Web Services (ECOWS). Halle, Germany, pp. 13–22 (2007)
97. Morrison, D.R.: PATRICIA-practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968)
98. Mühl, G.: Generic constraints for content-based publish/subscribe. In: International Conference on Cooperative Information Systems (CoopIS). Trento, Italy, pp. 211–225 (2001)
99. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: C. Bessiere (ed.) Proceeding of the CP, Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer (2007)
100. Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding random SAT: beyond the clauses-to-variables ratio. In: Principles and Practice of Constraint Programming, pp. 438–452 (2004)
101. Organising Committee of the Third International Competition of CSP Solvers: XML representation of constraint networks format XCSP 2.1. http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf (2009). Accessed 1 Feb 2012
102. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Math. Struct. Comput. Sci. **13**(5), 685–719 (2003)
103. Palamidessi, C., Valencia, F.D.: A temporal concurrent constraint programming calculus. In: Principles and Practice of Constraint Programming, pp. 302–316 (2001)
104. Pierce, B.C.: Foundational calculi for programming languages. In: The Computer Science and Engineering Handbook, pp. 2190–2207 (1997)
105. Popper, K.R.: The Poverty of Historicism. Routledge, New York (2002)
106. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: Principles and Practice of Constraint Programming, pp. 574–589 (2007)
107. Pulina, L., Tacchella, A.: Time to learn or time to forget? Strengths and weaknesses of a self-adaptive approach to reasoning in quantified Boolean formulas. In: CP Doctoral Program (2008)
108. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
109. Saraswat, V.A., Jagadeesan, R., Gupta, V.: Default timed concurrent constraint programming. In: Symposium on Principles of Programming Languages (POPL). San Francisco, California, pp. 272–285 (1995)
110. Saraswat, V.A., Rinard, M.C.: Concurrent constraint programming. In: Symposium on Principles of Programming Languages (POPL). San Francisco, California, pp. 232–245 (1990)
111. Schulte, C., Lagerkvist, M., Tack, G.: Gecode project. http://www.gecode.org/ (2012). Accessed 1 Feb 2012

112. Selman, B., Kautz, H.A.: Domain-independent extensions to GSAT: solving large structured satisfiability problems. In: International Joint Conference on Artificial Intelligence (IJCAI). Chambery, France, pp. 290–295 (1993)
113. Shapiro, E.Y.: The family of concurrent logic programming languages. ACM Comput. Surv. **21**(3), 413–510 (1989)
114. Sklower, K.: A tree-based packet routing table for berkeley unix. In: Proceedings of the Usenix Winter 1991 Conference. Dallas, Texas, pp. 93–104 (1991)
115. Smolka, G.: The Oz programming model. In: Computer Science Today, pp. 324–343 (1995)
116. Sneyers, J.: Turing-complete subclasses of CHR. In: International Conference on Logic Programming (ICLP). Udine, Italy, pp. 759–763 (2008)
117. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of constraint handling rules. ACM Trans. Program. Lang. Syst. **31**(2) (2009)
118. Streeter, M.J., Golovin, D., Smith, S.F.: Combining multiple heuristics online. In: Association for the Advancement of Artificial Intelligence (AAAI). Atlanta, Georgia, pp. 1197–1203 (2007)
119. Stuckey, P.J., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. Constraints **15**(3), 307–316 (2010)
120. Sutherland, I.E.: Sketchpad, A man-machine graphical communication system. In: Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York (1963)
121. Taleb, N.N.: The black swan: the impact of the highly improbable. Random House, Allen Lane (2007)
122. Ueda, K.: Guarded horn clauses. In: Proceedings of the 4th Conference on Logic programming. Tokyo, Japan, pp. 168–179 (1985)
123. Vaandrager, F.W.: Expressive results for process algebras. In: REX Workshop, pp. 609–638 (1992)
124. Verfaillie, G., Jussien, N.: Constraint solving in uncertain and dynamic environments: a survey. Constraints **10**(3), 253–281 (2005)
125. Versari, C., Busi, N., Gorrieri, R.: On the expressive power of global and local priority in process calculi. In: International Conference on Concurrency Theory (CONCUR). Lisbon, Portugal, pp. 241–255 (2007)
126. Vigliotti, M.G., Phillips, I., Palamidessi, C.: Tutorial on separation results in process calculi via leader election problems. Theor. Comput. Sci. **388**(1–3), 267–289 (2007)
127. Vovk, V., Gammerman, A., Shafer, G.: Algorithmic learning in a random world. Springer, New York (2005)
128. Wallace, R.J., Freuder, E.C.: Supporting dispatchability in schedules with consumable resources. J. Sched. **8**(1), 7–23 (2005)
129. Waltz, D.: Understanding line drawings of scenes with shadows. In: P. Winston (ed.) The Psychology of Computer Vision, pp. 19–91. McGraw-Hill, New York (1975)
130. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Association for the Advancement of Artificial Intelligence (AAAI). Atlanta, Georgia (2010)
131. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In: Constraints Programming, pp. 712–727 (2007)
132. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) **32**, 565–606 (2008)
133. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: formalization and algorithms. IEEE Trans. Knowl. Data Eng. **10**(5), 673–685 (1998)
134. Zoeteweij, P., Arbab, F.: A component-based parallel constraint solver. In: Coordination. Pisa, Italy, pp. 307–322 (2004)