MONITORING COMPLEX PROCESSES TO VERIFY SYSTEM CONFORMANCE

A DECLARATIVE RULE-BASED FRAMEWORK

STEFANO BRAGAGLIA

MARCH 2013

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
DEIS – DEPARTMENT OF ELECTRONICS, COMPUTER SCIENCE AND SYSTEMS

Ph.D. Course in
ELECTRONICS, COMPUTER SCIENCE AND TELECOMMUNICATIONS
Cycle XXV

Examination Sector 09/H1
Scientific Disciplinary Sector ING-INF/05

MONITORING COMPLEX PROCESSES TO VERIFY SYSTEM CONFORMANCE

A DECLARATIVE RULE-BASED FRAMEWORK

Candidate:
STEFANO BRAGAGLIA

Supervisor:
FULL PROF. PAOLA MELLO

Advisor:
DR. FEDERICO CHESANI

Ph.D. Course Coordinator:
FULL PROF. ALESSANDRO VANELLI CORALLI

Final Examination Year 2013

Stefano Bragaglia: *Monitoring Complex Processes to Verify System Conformance,* A Declarative Rule-based Framework © March 2013.

WEBSITE:
http://ai.unibo.it/People/StefanoBragaglia

E-MAIL:
stefano.bragaglia@unibo.it

«Success consists of going from failure to failure without loss of enthusiasm.»

— Winston Churchill

Questa tesi è dedicata alla mia famiglia che è per me fonte costante di ispirazione. Li ringrazio per avermi dato disciplina e supporto incondizionato per affrontare qualsiasi compito con entusiasmo e determinazione. Senza il loro affetto e sostegno, non mi sarebbe stato possibile conseguire questo risultato.

This page is intentionally left blank.

SOMMARIO

Negli ultimi 60 anni, i computer e i programmi hanno favorito incredibili avanzamenti in ogni campo. Oggigiorno, purtroppo, questi sistemi sono così complicati che è difficile – se non impossibile – capire se soddisfano qualche requisito o mostrano un comportamento o una proprietà desiderati. Questa tesi introduce un approccio a posteriori Just-In-Time (JIT) per effettuare il controllo di conformità ed identificare appena possibile ogni deviazione dal comportamento desiderato, ed eventualmente applicare qualche correzione.

Il framework dichiarativo che implementa il nostro approccio – interamente sviluppato su una promettente piattaforma open source di Production Rule System (PRS) chiamata Drools – si compone di tre elementi: 1. un modulo per il monitoraggio basato su una nuova implementazione efficiente di Event Calculus (EC), 2. un modulo generale per il ragionamento ibrido (il primo del suo genere) che supporta ragionamento temporale, semantico, fuzzy e a regole, 3. un formalismo logico basato sul concetto di aspettativa che introduce le Event-Condition-Expectation rules (ECE-rules) per valutare la conformità globale di un sistema. Il framework è anche accompagnato da un modulo opzionale che fornisce Probabilistic Inductive Logic Programming (PILP).

Spostando il controllo di conformità da dopo l'esecuzione ad appena in tempo, questo approccio combina i vantaggi di molti metodi a posteriori e a priori proposti in letteratura. Si noti che, se le azioni correttive sono fornite esplicitamente, la natura reattiva di questo metodo consente di conciliare le deviazioni dal comportamento desiderato non appena questo viene rilevato. In conclusione, la metodologia proposta introduce alcuni avanzamenti per risolvere il problema del controllo di conformità, contribuendo a colmare il divario tra l'uomo e la tecnologia, sempre più complessa.

SUMMARY

Over the last 60 years, computers and software have favoured incredible advancements in every field. Nowadays, however, these systems are so complicated that it is difficult – if not challenging – to understand whether they meet some requirement or are able to show some desired behaviour or property. This dissertation introduces a Just-In-Time (JIT) a posteriori approach to perform the conformance check to identify any deviation from the desired behaviour as soon as possible, and possibly apply some corrections.

The declarative framework that implements our approach – entirely developed on the promising open source forward-chaining Production Rule System (PRS) named Drools – consists of three components: 1. a monitoring module based on a novel, efficient implementation of Event Calculus (EC), 2. a general purpose hybrid reasoning module (the first of its genre) merging temporal, semantic, fuzzy and rule-based reasoning, 3. a logic formalism based on the concept of expectations introducing Event-Condition-Expectation rules (ECE-rules) to assess the global conformance of a system. The framework is also accompanied by an optional module that provides Probabilistic Inductive Logic Programming (PILP).

By shifting the conformance check from after execution to just in time, this approach combines the advantages of many a posteriori and a priori methods proposed in literature. Quite remarkably, if the corrective actions are explicitly given, the reactive nature of this methodology allows to reconcile any deviations from the desired behaviour as soon as it is detected. In conclusion, the proposed methodology brings some advancements to solve the problem of the conformance checking, helping to fill the gap between humans and the increasingly complex technology.

This page is intentionally left blank.

ABSTRACT

Over the last 60 years, computers and software have favoured incredible advancements in every field. Computerised systems have allowed to provide answers to increasingly complex problems in less and less time. Nowadays, however, these systems are so complicated that it is difficult – if not challenging – to understand whether they meet some requirement or are able to show some desired behaviour or property. Therefore, the problem of verifying whether a system conforms to a set of requirements is very common.

Solutions present in Literature and their limitations

In literature several solutions already exists, however modern systems are distributed, heterogeneous and hybrid. It means that components may have distinct natures, making it difficult to understand whether a property is maintained across the whole system. They can be localised elsewhere or be provided as black boxes so that checking whether they are operating properly could be rather complicated. They can also be almost freely assembled together or replaced at run time, thus defeating the ability to verify anything in advance. On one hand, it follows that sometimes a priori methods can not be applied simply because, for the above reasons, some critical information on the system is missing before it is started. A posteriori methods, on the other hand, are sometimes inadequate because they asses the conformance after execution, when some valuable resources may have been already depleted by an undesired behaviour that is emerging on the system. Last but not least, modern systems start to consider the users as integral part of the workflow by delegating actions to them and relying on their feedback. This introduces a new level of complexity to the problem since humans deal with information with a precision and a detail level which is somehow different from those of computers.

The proposed approach

This thesis aims to perform a conformance check that still qualifies as an a posteriori approach, but in a way that allows to identify the deviations from the desired behaviour of the system as early as possible. The approach follows the philosophy of JIT, widely used in industries and already borrowed by computer science (see JIT compilers): rather than deferring the whole check after execution, the analysis is performed step-by-step as soon as something happens. This still allows to identify all the deviations, but it also gives a chance to intervene almost in time and possibly spare precious resources. Such a goal is achieved by employing a monitoring tool that assists the system during execution and a Decision Support System (DSS) that observes any complex process that is running on the system to produce a conformance score and possibly apply some corrections or rewards. The

resulting framework is the first contribution of this thesis: the monitor consists of an efficient and unobtrusive novel implementation of the EC based on forward rules, while the DSS revolves around *expectations* – a concept introduced to define the desired behaviour of a system – and uses ECE-RULES (an adaptation of EVENT-CONDITION-ACTION RULES (ECA-RULES) towards *expectations*) to reason about them.

Much emphasis has also been given to the expressive power of the framework to ease the tasks where human interaction is involved. Humans, for example, have a different understanding of time than computers: a deadline a few hours away is still considered roughly met, even if it was missed by a couple of minutes. Generally speaking, they tend to express qualitative opinions on things, rather than precisely quantify their extension. Humans are also naturally inclined to use specific terms in place of more general ones and still apply their thinking to the latter. The second contribution of the thesis is a general-purpose reasoning module that we have specifically adopted to improve the reasoning power of the conformance framework. This module which is based again on REACTIVE RULES (RR) to favour its coupling with the rest of the framework, takes advantage of (some concepts of) FUZZY LOGIC (FL), TEMPORAL LOGIC (TL) and DESCRIPTION LOGIC (DL) to respectively deal with imprecise definitions, to take care of time and to correctly subsume concepts. To the best of our knowledge, no other tool is available with a similar potential.

Impact of the method

By shifting the conformance check from after execution to just in time, this approach combines the advantages identified in literature of a posteriori methods by properly addressing all the deviations surfacing on the system with the predisposition that is typical of a priori methods to possibly prevent inappropriate behaviours. One of the most remarkable consequences arising from the adoption of this methodology is that it becomes possible to intervene on the domain by exploiting the reactive nature of the framework as soon as some change on the system takes place. In the domain of SERVICE-ORIENTED ARCHITECTURE (SOA), for example, the QUALITY OF SERVICE (QOS) could be imposed whenever the SERVICE LEVEL AGREEMENT (SLA) is about to be violated if the appropriate corrective actions are available. The same principle could be exploited to trigger processes of KNOWLEDGE BASE REVISION (KBR) on the domain, since any violation of an expected behaviour could be interpreted as the result of a misleading, imprecise or incorrect implementation of a norm. One thing of no less importance is the expressive richness of the tool that enables to deal with interesting socio-technical scenarios such as COMPUTERISED CLINICAL GUIDELINES (CCG). In these contexts, in fact, users expect the computers to understand them and reason with their own modalities, while apparently they have been programmed in a more rational way to simplify their operation or improve their raw computational efficiency.

PUBLICATIONS

Some ideas and figures have previously and partially appeared in the following publications, however they have been further developed and processed before being included in this dissertation.

In particular, [7, 10, 14] have contributed in part to the writing of Chapter 2 on page 21 on EVENT CALCULUS (EC), [3, 5, 6, 8, 9] to the Chapter 3 on page 77 on fuzzy semantic rule-based hybrid reasoning, [4, 6] to the Chapter 4 on page 91 on EVENT-CONDITION-EXPECTATION RULES (ECE-RULES) and the idea of *global conformance*, [1, 12–14] to the Chapter 5 on page 111 describing some use cases and, finally, [2, 15, 16] to the Appendix B on page 183 on probabilistic graph problems.

[1] BRAGAGLIA, STEFANO. *Service Oriented Architectures and Cloud Computing: Models and Tools Based on Declarative Approaches*. In F. Chesani, M. Milano, and A. Saetti, editors, *Abstract Booklet of the First AI*IA Doctoral Consortium*, volume 1, pages 11–16. AI*IA, 2010. URL: http://aixia10.ing.unibs.it/images/dcaiia2010.pdf.
(Cited on page xv)

[2] BRAGAGLIA, STEFANO. *Reasoning on Logic Programs with Annotated Disjunctions*. *Intelligenza Artificiale*, 6(1):77–96, 2012. ISSN: 2211-0097. URL: http://dx.doi.org/10.3233/IA-2012-0029.
(Cited on page xv)

[3] BRAGAGLIA, STEFANO AND CHESANI, FEDERICO AND CIAMPOLINI, ANNA AND MELLO, PAOLA AND MONTALI, MARCO AND SOTTARA, DAVIDE. *An Hybrid Architecture Integrating Forward Rules with Fuzzy Ontological Reasoning*. In M. Graña Romay, E. Corchado, and M. Garcia Sebastian, editors, *Hybrid Artificial Intelligence Systems*, volume 6076 of *Lecture Notes in Computer Science*, pages 438–445. Springer Berlin / Heidelberg, 2010. ISBN: 978-3-642-13768-6. URL: http://dx.doi.org/10.1007/978-3-642-13769-3_53.
(Cited on page xv)

[4] BRAGAGLIA, STEFANO AND CHESANI, FEDERICO AND FRY, EMORY AND MELLO, PAOLA AND MONTALI, MARCO AND SOTTARA, DAVIDE. *Event Condition Expectation (ECE-) Rules for Monitoring Observable Systems*. In F. Olken, M. Palmirani, and D. Sottara, editors, *Rule - Based Modeling and Computing on the Semantic Web*, volume 7018 of *Lecture Notes in Computer Science*, pages 267–281. Springer Berlin / Heidelberg, 2011. ISBN: 978-3-642-24907-5. URL: http://dx.doi.org/

10.1007/978-3-642-24908-2_28.
(Cited on page xv)

[5] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Montali, Marco and Sottara, Davide. *Forward Rules and Fuzzy Ontological Reasoning: Heading toward Tight Integration.* In M. Gavanelli and T. Mancini, editors, *R.i.C.e.R.c.A. 2010: RCRA Incontri E Confronti*, 2009. URL: http://rcra.aixia.it/rcra2010/workshop-programme.
(Cited on page xv)

[6] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Montali, Marco and Sottara, Davide. *Fuzzy Conformance Checking of Observed Behaviour with Expectations*. In R. Pirrone and F. Sorbello, editors, *AI*IA 2011: Artificial Intelligence Around Man and Beyond*, volume 6934 of *Lecture Notes in Computer Science*, pages 80–91. Springer Berlin / Heidelberg, 2011. ISBN: 978-3-642-23953-3. URL: http://dx.doi.org/10.1007/978-3-642-23954-0_10.
(Cited on page xv)

[7] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Montali, Marco and Torroni, Paolo. *Reactive Event Calculus for Monitoring Global Computing Applications*. In A. Artikis, R. Craven, N. Kesim Çiçekli, B. Sadighi, and K. Stathis, editors, *Logic Programs, Norms and Action*, volume 7360 of *Lecture Notes in Computer Science*, pages 123–146. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29413-6. URL: http://dx.doi.org/10.1007/978-3-642-29414-3_8.
(Cited on page xv)

[8] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Sottara, Davide. *A Rule-Based Implementation of Fuzzy Tableau Reasoning*. In M. Dean, J. Hall, A. Rotolo, and S. Tabet, editors, *Semantic Web Rules*, volume 6403 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2010. ISBN: 978-3-642-16288-6. URL: http://dx.doi.org/10.1007/978-3-642-16289-3_5.
(Cited on page xv)

[9] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Sottara, Davide. *A Step toward Tight Integration of Fuzzy Ontological Reasoning with Forward Rules*. In P. Hitzler and T. Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 227–230. Springer Berlin / Heidelberg, 2010. ISBN: 978-3-642-15917-6. URL: http://dx.doi.org/10.1007/978-3-642-15918-3_20.
(Cited on page xv)

[10] Bragaglia, Stefano and Chesani, Federico and Mello, Paola and Sottara, Davide. *A Rule-Based Calculus and Processing of Complex Events*. In A. Bikakis and A. Giurca, editors, *Rules on the Web: Research and Applications*, volume 7438 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-32688-2. URL: http://dx.doi.org/10.1007/978-3-642-32689-9_12.
(Cited on page xv)

[11] Bragaglia, Stefano and Luccarini, Luca and Mello, Paola and Pulcini, Dalila and Sottara, Davide. *Monitoring the performance of Soft Sensors used in WWTPs by means of Formal Verification*. In *Proceedings of 6th International Congress on Environmental Modelling and Software*. iEMSs, 2012. URL: http://www.iemss.org/sites/iemss2012//proceedings/G1_2_0924_Luccarini_et_al.pdf.

[12] Bragaglia, Stefano and Luccarini, Luca and Mello, Paola and Pulcini, Dalila and Sottara, Davide. *Ontologies, Rules, Workflow and Predictive Models: Knowledge Assets for an EDSS*. In *Proceedings of 6th International Congress on Environmental Modelling and Software*. iEMSs, 2012. URL: http://www.iemss.org/sites/iemss2012//proceedings/A3_0926_Sottara_et_al.pdf.
(Cited on page xv)

[13] Bragaglia, Stefano and Luccarini, Luca and Mello, Paola and Pulcini, Dalila and Sottara, Davide. *Optimising the Management of SBRs using Formal Verification and Monitoring Techniques*. *Water Research*, submitted.

[14] Bragaglia, Stefano and Mello, Paola and Sottara, Davide. *Towards an Interactive Personal Care System driven by Sensor Data*. In M. Baldoni, F. Chesani, B. Magnini, P. Mello, and M. Montali, editors, *Proceedings of PAI 2012 Workshop and Prize for Celebrating 100th Anniversary of Alan Turing's Birth*, pages 54–59. CEUR-WS, 2012. URL: http://www.ceur-ws.org/Vol-860/paper17.pdf.
(Cited on page xv)

[15] Bragaglia, Stefano and Riguzzi, Fabrizio. *Approximate Inference for Logic Programs with Annotated Disjunctions*. In M. Gavanelli and T. Mancini, editors, *R.i.C.e.R.c.A. 2009: RCRA Incontri E Confronti*, 2009. URL: http://rcra.aixia.it/workshops/ricerca-2009/bragaglia.
(Cited on page xv)

[16] Bragaglia, Stefano and Riguzzi, Fabrizio. *Approximate Inference for Logic Programs with Annotated Disjunctions*. In P. Frasconi and F. Lisi, editors, *Inductive Logic Programming*, volume 6489 of *Lecture Notes in Computer Science*, pages 30–37. Springer Berlin / Heidelberg, 2011. ISBN: 978-3-642-21294-9. URL: http://dx.doi.org/10.

1007/978-3-642-21295-6_7.
(Cited on page xv)

CONTENTS

ACRONYMS

ADS	Active Database System	94
ADT	Abstract Data Type	56
AI	Artificial Intelligence	184
ALP	Abductive Logic Programming	93
AOP	Aspect Oriented Programming	85
API	Application Programming Interface	124
ASP	Answer Set Programming	149
AST	Abstract Syntax Tree	98
BDD	Binary Decision Diagram	186
BEC	Basic Event Calculus	26
BLP	Bayesian Logic Program	184
BPMS	Business Process Management System	91
BPM	Business Process Modelling	143
BP	Backpropagation	114
BRMS	Business Rule Management System	78
CCG	Computerised Clinical Guidelines	143
CC	Cloud Computing	143
CDSS	Clinical Decision Support System	117
CEC	Cached Event Calculus	27
CEP	Complex Event Processing	157
CO	Custom Operator	183
CRG	Compliance Rule Graph	11
CS	Cognitive Science	154
CTL	Computational Tree Logic	10
CV	Computer Vision	112
CWA	Closed World Assumption	145
DBMS	Data Base Management System	149
DD	Deductive Database	79
DLP	Description Logic Program	79
DL	Description Logic	144
DM	Data Mining	112
DO	Dissolved Oxygen	138

UI	User Interface	78
UML	Unified Modelling Language	158
VM	Virtual Machine	124
W3C	World Wide Web Consortium	78
WM	Working Memory	187
WSML	Web Service Modelling Language	79
WSMO	Web Service Modelling Ontology	79
WS	Web Service	143
WWTP	Waste-Water Treatment Plant	111
XML	eXtensible Markup Language	23

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

> *«There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things.»*
>
> — Niccolò Machiavelli
> Italian writer and statesman, Florentine patriot,
> author of '*The Prince*', 1469-1527

This year we are celebrating the centenary of life and work of Alan M. Turing, one of the putative fathers of Artificial Intelligence (AI). In such an occasion, it is natural to look back and see what scientific results have been achieved since the foundation of this discipline in 1950 [179]. In this time period, the disciplines that are related to AI have multiplied and many important research results were obtained. Thanks to a virtuous loop, the complexity of the problems and software systems that solve them has grown pairwise with the technological advances: the more impressive were the results and the more ambitious were the new challenges to address.

Nowadays software is so complicated that it is difficult to determine if it complies with the requirements and expectations of both programmers and final users, even with the help of appropriate automated tools. This depends on the heterogeneous, distributed, open nature of their architecture and on the complex, possibly disclosed or even malicious nature of single components. This issue is so generalised and widespread to become itself a reference problem that justifies a new specific discipline. The results of this research field are particularly important because they can be applied to several domains.

Technical domains

An application field that is often cited in this regard is that of Web Services (WSs) and Service-Oriented Architecture (SOA) in general. These technologies represented a real revolution, back in the '70s as the software was considered as monolithic until that point in computer science history. Today those seminal ideas are still changing the world if we consider the great interest that Cloud Computing (CC) is attracting. CC, in fact, still relies on basically the same technologies but it proposes them in an innovative, and for some aspects, even more extreme way. WSs become the atoms for huge architectures and services that are highly pervasive, continuous, distributed and even more heterogeneous – as it considers various possible interaction models. The typical SOA's problems of Quality of Service (QoS)

versus Service Level Agreement (SLA) become more complicated as they now also include the handling of points of variability, (semi-automatic) service composition and orchestration. If we consider the underlying hardware, it also raises problems of load balancing, clustering, service migration both at a level of Platform as a Service (PaaS) and Infrastructure as a Service (IaaS), in contrast with the former vision of Software as a Service (SaaS).

Socio-technical domains

The socio-technical domain is another interesting field. This term is used to refer to systems in which both humans and machines are expected to interact to reach a common goal.

A typical use-case is Computerised Clinical Guidelines (CCG). These systems aim to perform the same tasks that were once performed only by humans. They try to assist the humans in their duties by simplifying, speeding up, double checking the human contribution. These improvements balance in a way the better understanding that humans have which allow them to effectively take proper shortcuts, but they also cut out any biased opinion that they could have. Although the general concept is clear, guidelines are usually considered as very complicated to handle. They can specify which sequence of steps to take to reach a goal, possibly allowing some of the steps to be skipped, moved, added or modified. At other times, they express more or less precisely the domain thus limiting the degrees of freedom of the human actor and expressly specifying which steps must be avoided at any cost.

It is mandatory to stress the difference between machine-oriented and human-oriented tasks: in the former case, the performance of the actor are quite regular and any requirement or expectation about its behaviour may be precisely quantified and planned in time; in the latter case, however, this is not true anymore as humans consider deadlines and quantities in a more gradual, relaxed way. On top of that, the domain knowledge is sometime only partially explicated, making the problem even harder. In addition, sometimes the guidelines involve more actors with different roles – doctor, patient and administrative/legal employee, for instance – whose understanding and judgement on a specific instance may be conflicting. Therefore the technical part of the system has to consider and reconcile all of the above aspects.

These considerations also apply to other fields like eTourism, business processes or plant semi-automation just to name a couple of them, where sometimes the domain constraints and implications are less critical, but not less important.

In these use-cases, the ultimate goal may vary from the minimisation of costs to the maximisation of profit, from the reduction of pollutant to favouring some positive side effects. No matter the objective, these systems must also ensure some results for the humans that are taking part into the supplying or fruition of the service as the

satisfaction of the eTourism customer or the personal safety of the workers of a plant.

In case of frequently repeated tasks, the technical part that assists humans could also analyse the executions to infer relations between tasks and properly rephrase or rearrange the sequence of steps of a task to improve it. In a plant or hospital, for example, long shifts may imply a degradation of the quality of the final product or service as humans get tired so the machine could suggest to perform first the most delicate steps when humans are still focused and well motivated.

Complex processes

All the problems that we have sketched in the above paragraphs require the possibly reiterated execution of a sequence of steps by one or more actors with different roles to produce a concerted effort in a specific direction. As we have seen, the forms and methods of alteration that are both admissible or undesired in the sequence are many. For this reason, we decided to call *complex processes* these executions of tasks. For a complete and detailed definition of complex processes, we refer to the work by Urovi and Stathis [180].

In this dissertation we propose an improvement to one of the possible approaches that tries to determine whether a complex process is carried out in a satisfactory manner. This method qualifies as *a posteriori*, but it rather aims to provide its contribution Just-In-Time (JIT), as long as the process evolves. This kind of methods is composed of two main components: the first one is a monitoring module which observes what is happening on the underlying domain to figure out which is its current state, the second is a comparison module in which the current state is confronted with the desired outcome to determine whether the complex process is progressing nicely.

Solutions present in Literature

This issue has already been addressed in literature and several solutions have already been proposed. They can be grouped into two main categories. A first group of solutions proposes to perform an *a priori* static analysis of the systems under investigation to determine if they were built in a way to meet some specific rules, paradigms or standards. This approach is typically named *compliance checking*. The solutions of the second group, instead, are designed to analyse *a posteriori* the execution log of the systems being verified to state if their manifested behaviour had fulfilled some specific rules, paradigms or standards. We call this approach *conformance checking*.

Our own view

Many of them already tackle the problem in a rigorous and precise way, and few are already exploring the most promising direction to take. Our contribution, instead, focuses on flexibility and expressivity. We have seen, on one hand, that socio-technical systems require a more sophisticated semantics to cope with the processes. These systems, in fact, must not only be able to handle both machine and human tasks, but also to adapt to any specific use-cases where the permitted amount of relaxation and graduality varies. In addition, we

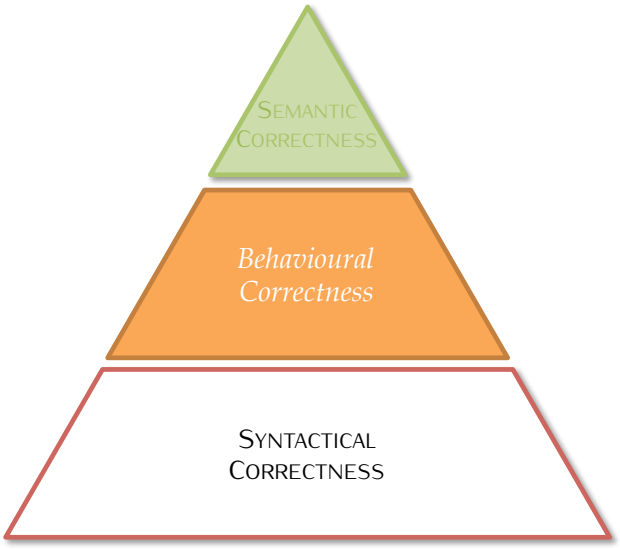


Figure 1.1: Pyramid of process correctness, as devised in [92].

strive to give the domain modellers an effective tool to represent the problem and tune the sensitivity of the method to the domain needs.

The remainder of this introductory Chapter is organised as follows. We first introduce the idea of process correctness and then we provide some terminology for the domain. In the following, we review the state-of-the-art approaches that are being researched in the literature, organising them in cross-organisational processes, process flexibility and business process compliance works. The concluding part of the Chapter is devoted to present our approach in more detail, taking special care in presenting what are the main contributions of this work.

1.1 PROCESS CORRECTNESS

As suggested by Knuplesch, Reichert, Mangler, Rinderle-Ma, and Fdhila in [92] for Business Process Modelling (BPM), the correctness of complex processes is assessed when three conditions are met. These conditions are layers that build one on top of each other, forming a pyramidal structure as in Figure 1.1. The shape of this structure is justified by the fact that the conditions of each level act as prerequisite for the following level. These levels of correctness – namely *syntactical correctness*, *behavioural correctness* and *semantic correctness* – are discussed below.

Syntactical Correctness

SYNTACTICAL CORRECTNESS This layer refers to the correct usage and composition of the items of the underlying meta-model. Examples of syntactical constraints include the existence of start and end events, as well as the proper use of the possible kinds of edges such

as the control flow edges that may only connect flow elements like tasks, gateways and events, or data flow edges that connect tasks with data objects. As pointed out in the opening of this section, the syntactical correctness of a process is a prerequisite for the behavioural correctness, since the behaviour of a syntactically incorrect model is not defined.

Behavioural Correctness

BEHAVIOURAL CORRECTNESS This layer requires a process model to be executable therefore it involves properties like the absence of deadlocks and livelocks. Moreover it depends upon the proper flow of data: data objects, for instance, must be written first before they can be read. With respect to the (dynamic) process changes, the state and data consistency must be preserved: an instance of a running process must not incur in any deadlock, lifelock, or data-flow error when it dynamically migrates its execution to a new version of the process model. In the context of cross-organisational processes, the compatibility between the public processes of the different partners requires their composition to be a behaviourally correct process. The conformance requires the private process of a partner to implement the behaviour of his public process. Also in this case, the behavioural correctness is prerequisite for the semantic correctness.

Semantic Correctness

SEMANTIC CORRECTNESS This layer requires that a process model complies with any rule that is stemming from regulations, standards and laws and imposed to the model. The business process compliance, for instance, is an example of semantic correctness. Therefore each possible execution of a process must not violate any compliance rule. In case of a set of compliance rules, the consistency of each single rule is not enough as their conjunction has to be satisfiable at the same time. Notice, in fact, that it is possible to model a process that complies with each rule of the compliance rules set.

1.2 DOMAIN TERMINOLOGY

In [52], Depaire et al. introduce a precise formal terminology to unambiguously address the problem. Processes implicitly define an expected behaviour, however deviations may happens due to both need for flexibility or incomplete description of the process. From this standpoint, deviations are not always undesired departures from a standard behaviour or norm. The identification of these kinds of deviation and their organisation in a hierarchy of concepts is every day increasingly important for *conformance checking* to help determine to which extent reality conforms to the designed process model and to which extent it deviates, cf. as a reference [2, 153, 154, 183, 194].

Deviations

The term *deviation* is used to denote a process execution that is

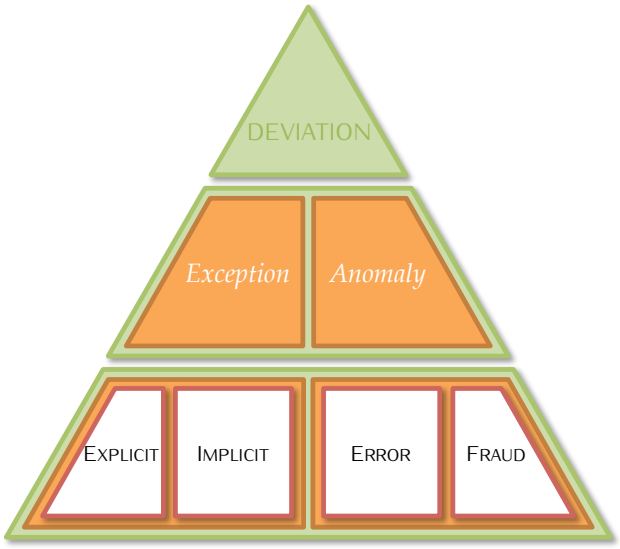


Figure 1.2: A hierarchical view of the several classes of process deviation, as devised in [52].

somehow not conforming to the normative process model. It is possible to identify several kinds of deviations that can be collected in a suitable hierarchy, as shown in Figure 1.2. A first distinction is based on the deviation's acceptability. As mentioned, in fact, not every deviation is necessarily despicable.

Exceptions and anomalies

In effect, on one hand we have deviations that are desirable because they improve the process by guaranteeing - for instance - the flexibility that is necessary to react fast and operate efficiently. These deviations are called *exceptions*. On the other hand, the unforeseen deviations from the process model that provoke an undesirable effect which results detrimental are denoted as *anomalies*.

Explicit and implicit exceptions

EXCEPTIONS This kind of deviations can be further divided into *explicit exceptions* and *implicit exceptions* (Figure 1.2). The difference between them is that the former are well known and precisely codified, while the latter are typically not depicted in the model or described as rules and in general tentatively adopted. Although the implicit exceptions typically retain their provisional nature and they rarely become regular procedures, depending on how much responsive and dynamic is the system or the organisation, they can of course be promoted to explicit exceptions or even to norms or rules. In both cases, they bring desirable improvements on the evolving of the process as they provide mechanics to treat degenerate cases or shortcuts to reach the goal faster or better. If we consider socio-technical processes [1], for instance, explicit exceptions are usually well consolidated within the framing system or organisation: they are often expressed as rule-

1 A socio-technical process is a process that involves some human task, action or feedback.

-bases and they represent best practices to handle specific sub-tasks within the general guidelines. An employee that asks the permission to skip a less relevant activity to a supervisor, for instance, is a clear example of implicit exception that is occasionally allowed to operate more efficiently and reach a goal quickly.

Notice that the distinction between an explicit exception and a full--fledged guideline is typically a rather subjective choice. In general terms, in fact, it is up to the system or to the organisation itself to determine what to consider as a norm or as an exception within the processes. In this regard, an indicator that is often used is frequency since it could indicate the most probable nature of a path.

Operational errors, errors and frauds

ANOMALIES As mentioned above, anomalies are undesirable deviations that can be divided into *operational errors* and *frauds* (Figure 1.2). It is evident that, given their unwanted nature, anomalies are never depicted in the process models and many efforts are usually made to avoid them as they are potentially harmful. Operational errors, or simply errors, typically are the mistakes that happen during a process execution that are caused by flaws of the underlying information system, by the unintentional human misuse or both. In a purely technical process as a Web service, for example, a programming "bug" could deviate the flow of execution from the expected behaviour. A simple misunderstanding perpetrated by a human that is completing a task within a socio-technical system or an organisation leads to an error as well. Frauds refer to any deliberate action deceptively performed by an actor which is purposely executed for personal gain against the system or the organisation. As the reader may guess, it is the worst kind of process deviation.

1.3 STATE OF THE ART

The domain in review

In this section we report the state-of-the-art research works as brilliantly outlined in [92]. The authors have identified three main areas of interest – *cross-organisational processes*, *process flexibility* and *business process compliance* – that will be addressed below. These works and their relationships are summarised in Figure 1.3 on the next page.

1.3.1 Cross-organisational Processes

The modelling of cross-organisational processes have been discussed for several years now. Widespread standards such as WS-BPEL or BPEL4WS [2], WS-CDL [3] and RosettaNet [4] are already available, as

2 https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
3 http://www.w3.org/2002/ws/chor/edcopies/cdl/cdl.html
4 http://www.rosettanet.org/

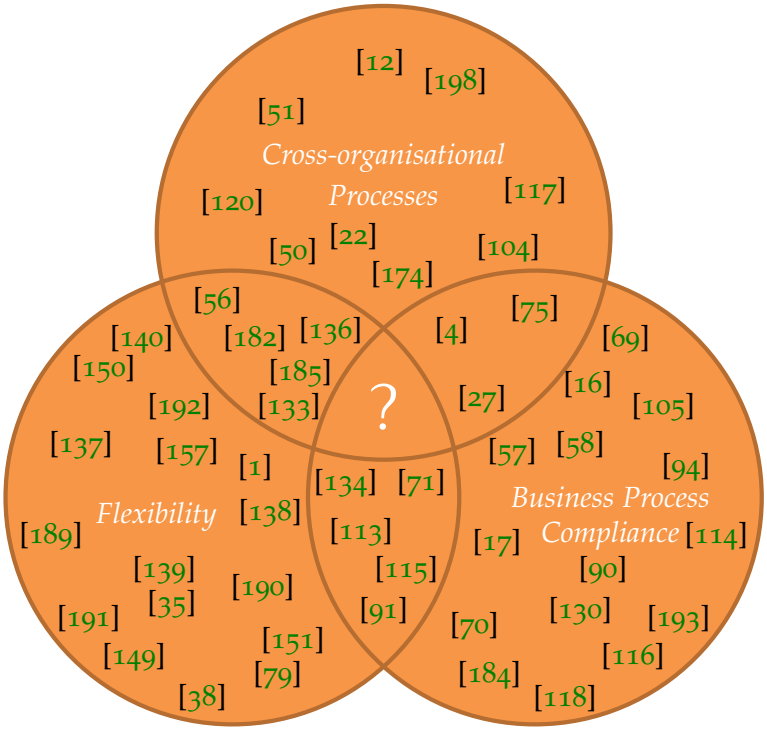


Figure 1.3: Summary on the state of the art works on processes, as devised in [92].

well as powerful modelling frameworks and formalisms like LET'S DANCE [198], iBPMN [51] and BPMN 2.0 [11].

Interaction patterns are further identified in [22] where well-defined patterns for message exchanges between partner processes are described. For privacy reasons, the definition of a choreography is usually restricted to those activities that are relevant for the message exchanges between the involved partners. In particular, the partners publish narrowed views on their own private processes [104, 117].

Several top-down approaches have been proposed to determine whether private processes comply with the corresponding public processes starting from a choreography of public ones [50, 120, 182]. In addition, a set of transformation rules that allows to obtain the private process corresponding to a public one thanks to a stepwise enrichment process that preserves inheritance is included in [185]. These rules can also be used to evolve the private processes without changing their public view. Works like the one in [174] enable the opposite practice to interconnect existing partner processes, for instance, where a bottom-up approach is defined to check whether processes can interact successfully or not. Furthermore, it proposes a solution that is suitable for both central and distributed architecture that allows for both the dynamic matching and the execution of cross-organisational processes thanks to a shared registry for public processes.

Finally, the Formal Contract Language independently addresses the scenario of the modelling of both process choreographies and private processes, permitting the conformance check between choreographies and processes [75].

1.3.2 Process Flexibility

Any issue that is related to process flexibility has been debated for more than a decade [56, 137, 150, 192]. Existing approaches, however, mainly consider flexibility for processes orchestration like in the case of those workflows implementing a private process and being executed by a single process engine. Pockets of Flexibility [157] and Worklets [1], for instance, are approaches in which the processes are executed on the basis of a loosely or partially specified model that is fully specified only at run-time. In this context, the relevant techniques are called *late modelling* and *late composition* of process fragments as well as *declarative modelling* [134].

Dynamic Process Adaptation

The so-called Dynamic Process Adaptation (DPA), in turn, represents the ability of any implemented process to cope with exceptional situations. On the one hand, this includes the handling of expected exceptions, which can be anticipated and thus be captured within the process model [138]. On the other hand, it also covers the handling of non-anticipated exceptions, which are usually addressed through structural adaptations of single process instances based on well-defined change patterns (for instance by adding, deleting or moving activities) [189]. A particularly interesting problem is to ensure the behavioural correctness of a process instance in terms of state and data consistency [150]. The approaches that are similar to ADEPT [139] guarantee the behavioural correctness of the modified process model instead.

Approaches based on changes

Besides that, there are some methods to support the end users in reusing ad-hoc changes [190] and in limiting changes to authorised users [191]. Another central aspect concerns process schema changes [35, 149, 151] such as the ability of the implemented process to change when the business process evolves. Congruent problems in this context affect the handling of the instances of the running processes that were created according the obsolete version of the model but are also required to use the fresh specification from that point on [35, 151]. Notice that the issue of behavioural correctness is rather critical as it is testified by the generally huge number of active instances that are affected by a change in a given process. The traceability of changes and the mining of dynamic processes are relevant issues as well. They are closely related to the evolution of processes. Both issues are addressed in a few existing frameworks [38, 79].

Changing processes and choreographies

There are only a few approaches that handle the changes of distributed processes and choreographies. The rationale behind [136],

for example, is that partitioned workflows can be changed in a controlled way. In [133], service changes in the context of choreographies are distinguished in shallow and deep. On one hand, the effects of shallow changes like changes in the services version, interface or operations are restricted to the single service. On the other hand, deep changes have cascading effects that are disseminated in the whole choreography. Unfortunately, no approach for a comprehensive solution is provided.

1.3.3 Business Process Compliance

Logic based approaches

In many domains, the execution of processes is subject to compliance rules and restrictions that stem from laws, regulations, or guidelines such as the *Basel* or *Sarbanes-Oxley* acts. The lifecycle phase in which the compliance check is performed and a strategy applied is the element that discerns the existing approaches that allow ensuring compliance of business processes with imposed compliance rules. In addition, several other paradigms and formalisms define process models and compliance rules [58]. The compliance rules are often considered as restrictions to the extent in which process activities may be executed. In this context, there are a few formalisations for compliance rules in temporal logic, namely Linear Temporal Logic (LTL) [16] and Computational Tree Logic (CTL) [69]. Other formalisations emphasise the modalities of compliance rules by means of Deontic Logic (obligations and permissions) [4, 71]. Since these approaches are rather complicated, a pattern-based method is addressed to encapsulate logic [57]. Some graphical notation to model compliance rules also exists [16, 105, 114]. The integration of compliance rules throughout the lifecycle of processes is discussed in [91, 113, 115].

State space approaches

The verification of whether process models fulfil compliance rules at design-time is addressed by several model checking techniques [16, 69, 94, 105]. These approaches depend on the exploration of the state space of the process models, thus the likely explosion of the states' search space represents a big obstacle for any practical application. In order to deal with this issue, graph reduction and sequencing techniques for parallel flows and predicate abstraction methods were proposed [16, 90, 105]. Among these approaches, there are a few that do not rely only the control-flow perspective: a method that exploits state-based data objects is suggested in [17], while [90] enables data-aware compliance checking for larger data-domains and [94] considers additional temporal constraints.

Compliance Rule Graphs

A few algorithms were also introduced to allow a more efficient compliance design-time verification than model checking for cycle-free processes [75, 193]. The run-time checking and monitoring of compliance is covered by the following methods. Process models, for instance, are enriched with a semantic layer of internal controls

in [130]. Another framework for the compliance monitoring based on common event standards is presented in [70]. In [27], instead, the monitoring and enforcement of compliance within process collaborations is outlined. Furthermore, the fine-grained compliance diagnostics is enabled at run-time by means of Compliance Rule Graphs (CRGs) in [116] and coloured automata in [118].

Declarative approaches

In order to complement the design-time and run-time compliance checking, some formulas in LTL were used to check process logs for compliance, proposing backward compliance checking of logs as a method [184]. Declarative approaches [4, 71, 134] were proposed as well to ensure compliance in a formal elegant way. In these systems, processes are described by means of a set of rules, therefore the rules to impose compliance simply need to be added to the process definition to ensure the business process compliance.

Notice that the procedure for the conformance checking that is outlined in this dissertation is affine to some of the works presented in this section.

To summarise, there are several approaches that are mixing aspects of both compliance of cross-organisational processes, as well as their changes. Only a few of them, however, discuss issues related to flexibility in the contest of business process compliance. Even fewer solutions address the business process compliance of cross-organisational processes. Ultimately, the interplay of change and compliance in the context of cross-organisational processes has not been practically addressed yet, as reported in Figure 1.3 on page 8.

1.4 MAIN CONTRIBUTIONS OF THE DISSERTATION

As suggested at the beginning of this Chapter, our approach fits in well with the conveyed concept of process correctness, placing itself in the group of Figure 1.3 on page 8 for *Business Process Compliance* somewhere near the boundary with *Cross-Organisational Processes*. Also notice that it tries to address all the kinds of process deviations that have been identified.

The general picture

Our methodology is based on two main steps and a few other additions to further improve the expressivity and capabilities of the approach. The two mandatory steps consist in a monitoring framework to assist the domain and in a tool to measure the distance between the state of a process and its ideal state. The optional addendum includes additional evaluators – such as semantic, fuzzy and probabilistic – that bring the adaptability and accuracy of the whole method to a new level by decoupling the problem of assessing the conformance of a complex process from the one of measuring how distant it is from the expected. The two steps that are needed to address this kind of problem do not present any particular novelty with respect to what is

already present in literature, however the choice to implement it in a Production Rule System (PRS) environment is rather fresh since it has a few implications that still need to be partially investigated. The way in which the collateral evaluators support the main process, instead, is rather new as well as the technical choices that we have made to implement them. The following paragraphs describe in details the main contributions of this dissertation.

Monitoring and Event Calculus

In Chapter 2 on page 21 we present the Event Calculus (EC), a wellknown theoretical framework to reason about actions and their effects on a domain, which is often adopted to monitor systems. The Chapter contains an introduction, a brief survey and an analysis of the most peculiar variants of EC. Since our goal is to implement a JIT framework for the evaluating the conformance of a complex process almost in real time, we paid particular attention to efficiency issues and solutions to overcome them. Then we have introduced a general architectural pattern to implement the EC in a PRS in a robust way. We also provided a guideline for users to successfully model any domain. The last section of the Chapter is devoted to the implementations: we have identified some property of the EC and we show how to exploit them to realise efficient declarative forward-chaining implementations. In particular we introduce first a Boolean version of EC as well as a generic multi-valued version which interfaces with the Fuzzy Logic (FL) extension of the adopted PRS thus providing the fuzzy EC. We have also compared our efficient variants of EC with other similar proposals that are already available in literature, obtaining comparable results.

Event-Condition-Expectations rules

The second mandatory step for a tool assessing the conformance of complex processes relates to handling the expected behaviour of these processes. In Chapter 4 on page 91 we introduce *expectations*: the concept that we use to express the desired behaviour of processes to be compared with their current state.

The Chapter opens with a brief survey on this context and then it continues presenting the concept of Event-Condition-Expectation rules (ECE-rules). These rules are forward rules in which we directly express and manage the concept of expectation of a complex process. They conceptually derive from Event-Condition-Action rules (ECA-rules), that were first conceived to address the idea of *active* rules. The central part of the Chapter focuses on the practical details to enable such rules in a modern PRS.

In this regard, we present a possible grammar to tackle them and we outline the architectural choices that are needed to empower them. We also describe the meta-model behind expectations – showing their possible states and patterns of evolution – and the additional reactive rules and optimisations that are needed "under the hood" to manage them. The last part of the Chapter focuses on the concept of *global conformance*.

Expectations introduce the idea of fulfilment, violation and repaired violation however they produce a conformance evaluation which can be considered as dull in many cases. Socio-technical systems, for instance, require at least a more gradual evaluation of deadlines depending on whether they refer to a human or synthetic actor. In this vein, other finegrained evaluators may be introduced to ease the reasoning and more precisely asses the conformance of the complex processes.

These evaluators are implemented by the optional steps discussed above and will be described in the following paragraphs. The results provided by these evaluators are combined together in the way that the domain modeller has suggested to provide a global score of conformance for execution being monitored of a complex process. The Chapter also contains an ongoing example that shows how the modelling of a domain improves thanks to the idea of global conformance with respect to the plain and simple case in which ECE-RULES are only used.

Fuzzy semantic rulesbased hybrid reasoner

Chapter 3 on page 77 contains the first corollary contribution that enhances the process of determining the global conformance of processes.

In particular, this Chapter is about the implementation of a single component that is capable of semantic, fuzzy and rule-based reasoning at the same time. The presentation starts with a review of the works and implementations that are currently available. According to the findings of our search, our component is the first to include all these kinds of reasoning at the same time but other similar works are appearing now.

The Chapter encompasses the description of the architecture of a first prototype which adopted distinct external tools to provide the several flavours of reasoning. This choice was motivated by the fact that dedicated tools generally allow deeper reasonings, however, they typically produce much overhead to keep their single knowledge base synchronised. The Chapter therefore also includes a description of tightly-coupled and slightly less expressive variant of the reasoner. This variant implements a fuzzy version of the *Tableaux* algorithm by means of REACTIVE RULES (RR) to provide the functionalities that are expected by DESCRIPTION LOGIC (DL) and FUZZY LOGIC (FL). As the reader will see, this implementation also includes ideas coming from MIXED-INTEGER LINEAR PROGRAMMING (MILP) and OBJECT-ORIENTED PROGRAMMING (OOP) to overcome some practical threats.

The Chapter is concluded by an explanation of the meaning behind the augmented expressivity of the tool and a necessary consideration about its usage. This tools, in practice, merges two parts that operate according to contrasting principles: the OPEN WORLD ASSUMPTION (OWA) and CLOSED WORLD ASSUMPTION (CWA). The adoption of the more ductile fuzzy semantics works around this limitation, how-

ever specific operators are needed to interpret the results according to those assumptions, as needed. This is the reason why no tool allows yet to mix those assumptions in a single reasoning, but component like our contribution allow to seamlessly manage together reasonings of different nature.

Finally, the Chapter proposes a long example which should help to better understand the capabilities of this reasoner.

Probabilistic graph-based problems with rules

A second corollary contribution is presented in Appendix B on page 183 and focuses on Probabilistic Inductive Logic Programming (PILP). In this Appendix we briefly introduce some well known formalisms to reason about probability with rules and, in particular, Logic Programs with Annotated Disjunctions (LPADs). We basically show how this kind of problems usually consists in a search of paths within connected graphs or, more properly, in determining the probability that at least a path exists between two given nodes of the graph.

The remaining part of the Appendix is devoted to some original declarative implementations realised within a PRS. The first proposal is named *topological approach*, as it tries to identify specific topological patterns in the graph and to replace them with probabilistically equivalent edges. Unfortunately, there are a few topological patterns that cannot be tackled in this way, so we propose a second method named *flow approach* as it resembles some well-known flow algorithms. The idea behind this approach is to propagate the probability through the network as the water in the pipes of an aqueduct. Unfortunately this method fails on graphs with undirected edges which, on the contrary, are quite common. We also propose a more *classical approach* which first finds all the paths between the chosen terminal nodes, then build a diagrammatic representation – using Binary Decision Diagrams (BDDs) for instance – and finally efficiently compute the equivalent probability by visiting such a diagram. This method is general and solves any kind of problem, however it uses a strategy that is more suitable for backward-chaining environments rather than forward-chaining ones.

Notice that this kind of reasoning is not strictly necessary to compute the global conformance of a complex process, however we believe that it could bring some interesting contributions. Therefore we have decided to include it as an Appendix. Also notice that in the conclusions of this work we have mentioned a few suggestions for future works that should address and overcome the current limitations of this approach.

Background information on Production Rule Systems

Appendix A on page 153 contains an introduction to PRSs where we describe their semantics and principles of operation. The Appendix also contains several examples to assist the reader that is not familiar with this kind of tools in learning *Drools* – a rather interesting and widespread open source implementation of a PRS – and get started with its language. The Appendix includes similar discussions and

examples to start using *Drools'* temporal and fuzzy extensions: *Drools Fusion* and *Drools Chance*. Notice that this Appendix probably does not contain any theoretical or practical original contribution, however it has a long series of commented original examples and explanations that testimony at least a considerable effort for teaching.

Applications and conclusions

Last but not least – before the final Chapter 6 on page 143 in which we draw and comment some conclusion about the work sketched in this dissertation and we propose a few ideas to further refine it with future work – Chapter 5 on page 111 contains a showcase of practical applications that have been developed using the tools presented in this thesis.

1.5 ON THE CHOICE OF THE DEVELOPMENT TOOL

Most of the research carried out within the conformance checking domain – and virtually any contribution from our research group – involves logical tools and, specifically, PROLOG. These softwares are particularly interesting because they rigorously represent and enforce the reasoning by means of rules. According to our experience, however, such aspects are very appraised in academy but not in industry. This is probably due to to the fact that the latter are more compelled by compatibility issues with the rest of the existing software infrastructure rather than fascinated by the possibility to demonstrate the soundness of the computation. It follows that, although the business world is rich of interesting use cases, it is often difficult to involve corporations on research topics like that. Conversely, PRSs originate from the business sector and are viewed with suspicion by some research groups as partially unsound applications that could wrongly supersede the former tools. Probably, however, the truth lies somewhere between these two positions.

Despite both systems are based on the concept of *rule*, they differ in some implementation details and for their more congenial areas of usage. PROLOG, for instance, is a tool that tries to determine whether it is possible to demonstrate a goal by concatenating the available rules and facts. PRSs, instead, have a generative behaviour that tries to determine which rules are activated by the current facts' status to assert new facts (possibly with a cascading effect). The differences between specific technical aspects of these software like *backward-chaining*, *forward-chaining*, *backtracking*, *unification* and *pattern matching* are discussed in Appendix A.1 on page 153. The point is that PRSs are typically based on the same tools that are used by corporations (such as *C/C#* and *Java*). These environments host several packages, libraries and tools which can be easily integrated, thus opening new possibilities. Consider, for instance, that there are several packages to provide semantic capabilities to a *Java* application, while there are

almost no similar extension in PROLOG [5] (see Chapter 3 on page 77 for more details). Moreover, most of the PRSs have embraced OOP and COMPLEX EVENT PROCESSING (CEP) allowing a direct interaction with the corporations' DOCUMENT OBJECT MODELS (DOMs) while responding to the events that are going on their application domains.

The motivation behind the choice of a PRS as the implementation platform is to seize the opportunity to acknowledge all the possible industrial use cases and involve corporations into the sound practice of conformance checking, while taking advantage of the large amount of software available to identify new applications and opportunities.

With respect to the choice of the specific PRS, we have considered several alternatives. At the moment, there are a few commercial platforms and a plethora of implementations for educational purposes or at hobby level. The latter are so many that it is almost impossible to list and categorise all of them. Most of them are basic non optimal implementations of the RETE algorithm based on several programming languages and lacking many characteristics of modern PRSs. (Those characteristics are introduced and discussed in the following paragraph.) The reader that wants to evaluate those lesser implementations can simply find them by querying any search engine with keywords like "`production rule systems`", "`rete implementations`", "`forward chaining rule engines`" or similar.

Commercial PRSs includes CLIPS [6] (a public domain software tool for building expert systems, DROOLS [7] (a *Java* open source BUSINESS RULE MANAGEMENT SYSTEM (BRMS)), DTRULES [8] (a open source rule engine for *Java* based on DECISION TABLES), ILOG RULES [9] (IBM's own BRMS), JESS [10] (a rule engine for *Java* that is a superset of CLIPS), LISA [11] (a rule engine written in *Common Lisp*) and OPENL TABLETS [12] (a business centric open source BRMS) (more pointers to some of those systems are provided in Appendix A on page 153). Among these, only a few support all the features that are expected in a modern PRS like the support to DECISION TABLES/SPREADSHEETS, TEMPORAL LOGIC (TL), CEP and FL, for instance. The leading PRS is perhaps RETE-NT, a very efficient implementation in *C* by Forgy, the man who first proposed the RETE algorithm [63]. According to some benchmarks published by *Sparkling Logic* [13], the company that Forgy joined as investor and strategic advisor, RETE-NT is deemed to be a few hundreds times faster than the original algorithm and generally faster than any

5 At the moment there are a few projects that aim to do so but, to the best of our knowledge, the only comprehensive work is the M.Sc. thesis by Herchenröder [81].
6 http://clipsrules.sourceforge.net/
7 http://www.jboss.org/drools/
8 http://dtrules.com/
9 http://www-01.ibm.com/software/websphere/ilog/
10 http://herzberg.ca.sandia.gov/
11 http://lisa.sourceforge.net/
12 http://openl-tablets.sourceforge.net/
13 http://my.sparklinglogic.com/index.php/about-sparkling-logic

other implementation. Over the past few decades, while at *Carnagie Mellon University*, Forgy has also contributed to various versions of OPS (which is said to be the short for "*Official Production System*"), a milestone implementation (formerly in *Lisp* and later in *BLISS* [14]) of the RETE algorithm that – back in time – was able to scale up to larger problems involving hundreds or thousands of rules. A version of OPS maintained by *RuleWorks* [15] which includes several optimisations is also available, however both *Sparkling Logic*'s and *RuleWorks*' implementations are commercial.

Our choice was guided by the availability of the source code, modern features and corollary projects. A vibrant community supporting the open source development of the tool was also regarded as a plus. With respect to all these aspects, DROOLS was the best candidate. Therefore throughout this thesis we assume that the reader is at least familiar with the main concepts of PRSs and this software in particular. The reader that is not accustomed with them may find an introductory presentation in Appendix A on page 153.

14 Much less famous than *Lisp*, *BLISS* is considered one of the best known systems programming language until *C*; more information on this language is available on *Wikipedia*: http://en.wikipedia.org/wiki/BLISS_programming_language.

15 http://www.ruleworks.co.uk/uguide/rwug1.html

Part I

DEVELOPMENT FRAMEWORK CONTRIBUTIONS

2 EVENT CALCULUS

«The best way to keep good actions in memory, is to refresh them with new.»

— Thomas Carlyle,
Scottish Historian and Essayist,
leading figure in the Victorian era, 1795–1881

In this Chapter we will introduce the Event Calculus (EC) – a well known formalism to reason about events and their effects on a domain, and we will present our own implementations based on Reactive Rules (RR). These tools will be used to monitor systems as the representation of their state will be used later to assess their conformance.

In particular, we provide a brief introduction to EC where we discuss its philosophy and modes of operations, as well as some of its variants. Then we outline a classical problem of EC as an example in order to clarify its principles of operation. In the following section we finally present the general architecture of our tool and we discuss the process that led us to define different versions of the calculus. All these versions are optimised to process the occurring events as fast as possible, and they include modes of operations that are suitable for contexts where events are supposed to be notified in reasonable time or with a delay that scrambles the expected chronological order. We have also adapted these efficient modes of operations to work in a context where only Boolean values are used to model the status of the domain, and in a more general one where any kind of data is allowed. The latter case, combined with the fuzzy extension of our reference Production Rule System (PRS), allows to make use of fuzzy linguistic variables on any of these domains (i. e. *empty* or *full* to describe the filling level of a tank whose capacity is 100 litres). Last but not least we report the extensive test case that we have conducted on these implementations and some experimental evidence of their performances.

Please notice that we will often refer to the architecture and the way of functioning of a PRS during this Chapter. Providing a comprehensive presentation about this topic is out of the scope of this dissertation, however, the key concepts are summarised in Appendix A on page 153 for the sake of the reader's understanding.

2.1 FUNDAMENTALS OF EVENT CALCULUS

The EVENT CALCULUS (EC) is a well known logic-based formalism to reason about actions and their effects on a domain. It was introduced for the first time by Kowalski and Sergot in 1986 [98] and after more than 25 years is still subject of much research. It has been adopted in a variety of domains, such as cognitive robotics [162], planning [165], service interaction [119] and composition [152], active databases [61], workflow modelling [42], legal reasoning [60], business process management [196], computerised clinical guidelines [175], service-oriented computing [83] and multi-agent systems [195].

The basics: event and fluent

As for other similar languages – the most prominent of which is probably the SITUATION CALCULUS (SC) [123], it is essentially based on only two concepts: the *event* and the *fluent*. An event is defined as any thing that occurs at any given time on a domain that causes at least a partial change of its state. In a way, the events are notifications of the occurrence of some actions on the domain of relevance. For this reason, several authors use the terms "event" and "action" interchangeably [1]. Consequently, the notification of events causes some modifications that are specific of the single actions as they occur on the domain. A fluent instead is any single measurable aspect of the domain that is subject to change over time. Roughly speaking, they may be considered as variables that may take different values in the course of the time. Therefore, the set of all the variables that are needed to define a problem also provides an operational description. If we consider it as a FINITE STATE MACHINE (FSM), any full assignment of these variables identifies one of its possible states. When contextualising these assignments in time, we can define peculiar states – such as the *initial state* or the *current state*, or even consider them in sequences to describe the behaviour of the domain.

Some properties:
- simplicity

As the reader may see, being based on only two concepts makes the formalism very simple, but not simplistic. Simplicity is a good property for a language, provided that it does not limit its extent. Simple languages, in fact, generally require less resources to be understood and interpreted. The resulting implementations are usually simpler to maintain and also more robust. Complicated tools, in fact, generally can perform more sophisticated tasks but they tend to be easier to break or to be misused. This formalism is also regarded as quite versatile because of its modular approach that decomposes the domain state into simpler fluents. Complex or large problems, in fact, can be destructured in smaller pieces – up to the detail level of the individual fluents – and be handled seamlessly. In addition to these properties, it is also sound and reliable. In effect, unlike other similar formalisms (like some initial implementations of SC which is arguably

- robustness

- versatility

- soundness

1 A few authors, instead, distinguish between *actions* as events perpetrated by some subject, and *events* that are originated by no specific source.

the oldest special-purpose knowledge representation formalism, designed to axiomatise knowledge of actions and their effects) it is not affected by the frame problem. This is a quite common problem in Artificial Intelligence (AI) that arises when a dynamical domain is going to be expressed in logic without explicitly specifying the conditions that are not affected by each action. Its name is supposed to derive from the technique used by animated cartoonists called "framing" where the currently moving parts of the cartoon that are used to display actions are superimposed on the still picture which depicts the background of the scene. The way in which the EC overcomes this problem has been deeply investigated [163]: it has been recognised to be similar to the *successor state axioms* of the more recent versions of SC [141], and also formalised in terms of *circumscriptions* [122].

Formalisation of the calculus

Most of the implementations are realised in PROLOG (the only exception that we are aware of is the implementation formalised in eXTensible Markup Language (XML) and coded in *Java* that was developed by Farrell, Sergot, Sallé, and Bartolini in 2005 to to represent contracts and automatically keep track of their normative state), the language that was originally used to define the EC. In those works, the EC is introduced by using statements that pertains to the *Horn* subset of classical logic augmented with Negation as Failure (NAF). This subset includes clauses – precisely *Horn* clauses – that are disjunctions of literals with at most one positive literal [82]. NAF is the non-monotonic inference rule that is used in logic programming to derive the negation of a predicate from the failure to derive the predicate itself [43].

Core axiom

The essence itself of EC is contained in the following sentence that describes its main operating principle:

- a fluent is TRUE in a given time instant t *if and only if*
  - 1. it was initially TRUE, *or*
    2. it has been made TRUE in the past
  - *and* has not been made FALSE in the meantime.

This statement, commonly called *core axiom*, translates into the following predicates which respectively recall the meaning of the three conditions above:

$$\begin{aligned}
&\mathtt{holdsAt}(F,T) \leftarrow \\
&\quad \mathtt{initially}(F), T_0 < T, \neg \mathtt{clipped}(F, T_0, T). \\
&\mathtt{holdsAt}(F,T) \leftarrow \\
&\quad \mathtt{happens}(E_i, T_i), \mathtt{initiates}(E_i, F), T_i < T, \neg \mathtt{clipped}(F, T_i, T). \\
&\mathtt{clipped}(F, T_s, T_f) \leftarrow \\
&\quad \exists E, T : [\mathtt{happens}(E,T), \mathtt{terminates}(E,F), T_s < T, T < T_f].
\end{aligned}$$

In this formalisation, E, F and T (and its siblings T_0, T_i, T_s and T_f) are terms respectively indicating an *event*, a *fluent* and a *time* (*initial time*,

AXIOM	MEANING
holdsAt(F, T)	*Fluent* F *holds at time* T
initially(F)	*Fluent* F *holds from the initial time*
happens(E, T)	*Event* E *happens at time* T
initiates(E, F, T)	*Event* E *initiates fluent* F *at time* T
terminates(E, F, T)	*Event* E *terminates fluent* F *at time* T
clipped(F, T_1, T_2)	*Fluent* F *is terminated by an event in* (T_1, T_2)

Table 2.1: The EVENT CALCULUS ontology.

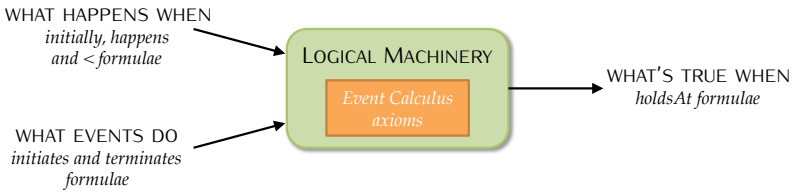


Figure 2.1: Operating diagram of the EVENT CALCULUS machinery.

generic subsequent time, *start time* and *final time*). $<$ is simply a total ordering relation over time instants. Generally the time is represented by numbers in $\mathbb{R}$, $\mathbb{Z}$ or $\mathbb{N}$.

Auxiliary axioms

As suggested by the core axiom formalisation itself, additional auxiliary domain-dependent axioms are needed to complete the definition of any problem. These auxiliary axioms include *temporal information* about occurring events provided by the happens(E, T) predicate and knowledge about the starting configuration of the domain thanks to the initially(F) predicate, as well as *causal information* binding events with their effects on fluents by means of the initiates(E, F, T) and terminates(E, F, T) predicates. In particular, these predicates are used to express that an event E is potentially responsible of the switching of a fluent F state, by respectively starting or ending at time T a time interval in which the fluent holds. The clipped(F, T_1, T_2) predicate is used instead to determine whether a fluent has been set to FALSE by an event within a given timeframe. Last but not least, the holdsAt(F, T) predicate is used to query the KNOWLEDGE BASE (KB) to understand whether a fluent holds at a given instant. All these axioms are collected and briefly explained in Table 2.1 that authors often call EVENT CALCULUS ONTOLOGY. Notice that the above formalisation has been defined in terms of Boolean values, but it may be generalised. Some details about a possible generalisation are provided in Chapter 2.3.4 on page 54.

General mode of operation

The general mode of operation of the EC is sketched in Figure 2.1. In the following, we use the typical predicate/arity notation of PRO-

LOG to refer to the axioms. It recalls a definition of EC as "*a logical mechanism that infers* WHAT'S TRUE WHEN *given* WHAT HAPPENS WHEN *and* WHAT ACTIONS DO" [164]. This machinery, in practice, uses initially/1 statements coming from the WHAT HAPPENS WHEN input to define the initial state of the domain. Then, any time a happens/2 predicate from the same input source is notified, it looks up for the corresponding initiates/3 or terminates/3 statement. They are provided as behavioural information on the WHAT EVENTS DO input. Once they have been identified, their effects are applied to the current state to build the representation for the new state. The machinery finally uses the holdsAt/2 predicate to determine if a fluent holds in the state of the states sequence that is associated with the given time t. Such information representing WHAT'S TRUE WHEN is returned as output. Notice that the "<" axiom is required to determine temporal precedence: its definition, however, is not relevant provided that it satisfies integrity constraints like *transitivity* and *anti-symmetry* [158].

Deductive reasoning

When we use WHAT HAPPENS WHEN and WHAT EVENTS DO as input and WHAT'S TRUE WHEN as output, like in the case that we have just described, the machinery performs *deductive reasoning*. Typical tasks of this reasoning style include temporal *projection* or *prediction*: they are generally used to determine the outcome of a known sequence of actions. This is the kind of reasoning that we will exploit to perform monitoring. As an additional evidence of the versatility of EC, consider that other reasoning styles can be obtained just by permuting the input and output of the machinery. For instance, if WHAT EVENTS DO and WHAT'S TRUE WHEN serve as input and WHAT HAPPENS WHEN as output, the machinery performs *abductive reasoning*. Abduction is generally used to hypothesise sequences of actions that lead to a desired state. Generally speaking, however, it solves tasks referring to temporal *explanation* or *postdiction*, certain kinds of *diagnosis* and *planning*.

Abductive reasoning

Inductive reasoning

Finally, *inductive reasoning* focuses on WHAT'S TRUE WHEN and WHAT HAPPENS WHEN to return WHAT EVENTS DO. It performs certain kinds of *learning*, *scientific discovery* and *theory formation* and, generally, it is used to supply a set of rules or a theory by considering the effects of actions that accounts for the observed data.

We will now see a few variants of the EC, paying particular attention to their different behaviour that has a sensible impact on performances.

2.1.1 Families of Event Calculus Variants

The EC family tree

Over the years, a large number of variants of the EC has been proposed, each with its own characteristics and peculiarities. The family tree in Figure 2.2 on the next page collects a few remarkable versions that we believe to be relevant from the standpoint of systems

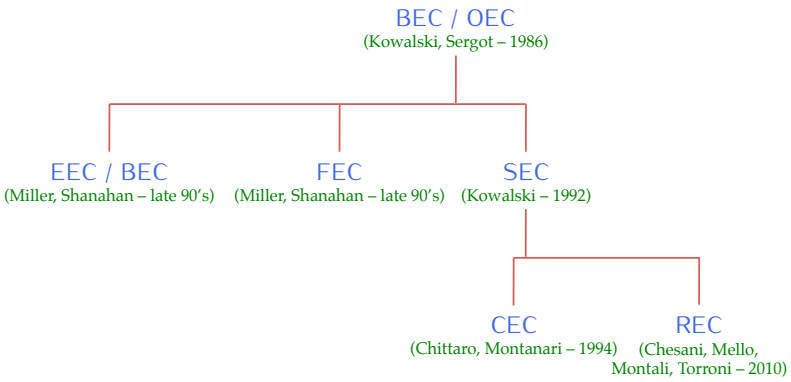


Figure 2.2: A (very partial) family tree of the EVENT CALCULUS variants.

monitoring, and shows their mutual relationships. These variants are summarised and compared in the following paragraphs.

Original EC

The root ancestor on top of the tree is the formalism introduced in the first place by Kowalski and Sergot in 1986 [98]. Some authors refer to it as BASIC EVENT CALCULUS (BEC) [36] or ORIGINAL EVENT CALCULUS (OEC) [129].

Classic variants: SEC, FEC and EEC

A second variation that directly stems from this one was suggested in 1992 by Kowalski [95]. Compared to the previous one, this is a reformulation based on time instants rather than intervals. It includes the notion of time instants, of course, as well as the idea of event types, incompatible fluents and initial state of fluents (specifically, the axiom $\mathtt{initially}(F)$) [129]. Shifting the focus from the intervals to the instants, it is considered a simplified version of EC to the point that it is conveniently referred as SIMPLE EVENT CALCULUS (SEC) [2] [129, 158, 164]. In the following years, other authors have contributed to the development of the EC by proposing new variants. Miller and Shanahan, for example, have introduced the FULL EVENT CALCULUS (FEC) and EXTENDED EVENT CALCULUS (EEC) (sometimes called BEC [36], but it should not be confused with the OEC discussed above). The FEC owes its name to the fact that it considers a more *complete* formalisation: with respect to the BEC, in fact, it includes some axioms such as $\neg\mathtt{initially}(F)$, $\neg\mathtt{holdsAt}(F,T)$ or $\mathtt{declipped}(F,T_1,T_2)$ that provide the dual formulation of the axioms of EC and a new axiom $\mathtt{releases}(E,F,T)$ which is used to state that a fluent is not subject to inertia after the happening of an event [164]. Similarly, the EEC has been called in this way because it is an extension of the FEC: it adds a couple of more axioms – namely $\mathtt{cancels}(E_1,E_2,F)$, $\mathtt{cancelled}(E,F,T_1,T_2)$ and $\mathtt{trajectory}(F_1,T,F_2,D)$ – that ease the handling of concurrent events and continuous changes [164]. Both these features may play a key role in treating complex processes across systems in some domains like SERVICE-ORIENTED ARCHITECTURE (SOA) where events with contrasting effects on a fluent's state may occur simultaneously or one during

2 Sometimes some authors use *simplified* or *simpler* in place of *simple*.

the other's progressing [3]. Regardless of the number and type of available axioms, all these variants operate in the same way: any time the occurrence of a new event is notified, the history of the changes occurred on the fluents' values is discarded and computed again from scratch. This is due to the fact that, in logic, predicates can not be "notified", but simply be TRUE or FALSE. Therefore, each time the KB is changed, the resolution mechanism that is typical of the PROLOG EC implementations needs to find a *derivation* that explains why the conclusion holds. This mechanism tries to unify the current goal with the clauses and eventually the facts that are asserted in the KB. In practice, every time it rebuilds the same derivation, eventually appending something to address the new knowledge.

Incremental variants: CEC and REC

To address this inefficiency, a new group of variants has been introduced. Any time we can assume that *the goal remains fixed* (the goal is always determining when fluents hold) and *the narrative grows towards the future* (new events are generally appended to the current end of the history), in fact, the domain is said to be *causal* [4] and we can conclude that small changes in the input cause only small alterations on the output [128]. Such idea goes by the name of *common-sense law of inertia* and tells us that only a few fluents are affected by the occurrence of an event. In other words, it suggests that *updating* the fluents' history requires much less computational resources than *recomputing* it. A first example that derives from SEC (see Figure 2.2 on the facing page) is the CACHED EVENT CALCULUS (CEC). This variant by Chittaro and Montanari introduces MAXIMAL VALIDITY INTERVALS (MVIs) – axioms in the form $mvi(F, T_1, T_2)$ representing an uninterrupted interval in which a fluent holds – to decouple the current state of a fluent from the fluent itself and to provide a general data structure in which to *cache* its history [40]. Upon its introduction, an MVI is typically *open* which means that its initial time is bound while its final time is not [5]. Unbound final times may be set in a second time. In any case, the MVIs whose final time is bound are called *closed*. In principle, the notification of the occurrence of a new event causes the closure and/or the introduction of one or more MVIs. Notice that it is possible to determine whether a fluent holds in a given time by properly querying its set of MVIs: if there is an MVI that includes that instant, then the fluent holds at that time [6]. The REACTIVE EVENT CALCULUS (REC) is an advanced, more efficient implementation of the CEC that has been introduced in 2010 [39]. It is also based on MVIs and their assertion

3 One basic assumption of the EC is that events can not occur at the same time; some domains like SOA, however, may be so *dense* (of events) and the time granularity so *poor* that it is not possible to determine the exact order of events.

4 A domain is *causal* when the clipping and declipping of a fluent only depends on past or present events.

5 The closing time of an *open* MVI is usually considered equivalent to *infinite*.

6 Given a fluent, for each instant of time there is at most one MVI that includes it.

EVENT	DESCRIPTION w.r.t. Fluent F initially FALSE
e_1	*Declipping* event at T_1 ($> T_0$)
e_2	*Clipping* event at $T_2 > T_1$ ($> T_0$)
e_3	*Declipping* event at $T_3 > T_2 > T_1$ ($> T_0$)
e_4	Delayed *declipping* event at T_4, $T_3 > T_4 > T_2 > T_1$ ($> T_0$)

Table 2.2: Narrative of events used to compare the modes of operation of SEC, CEC and REC.

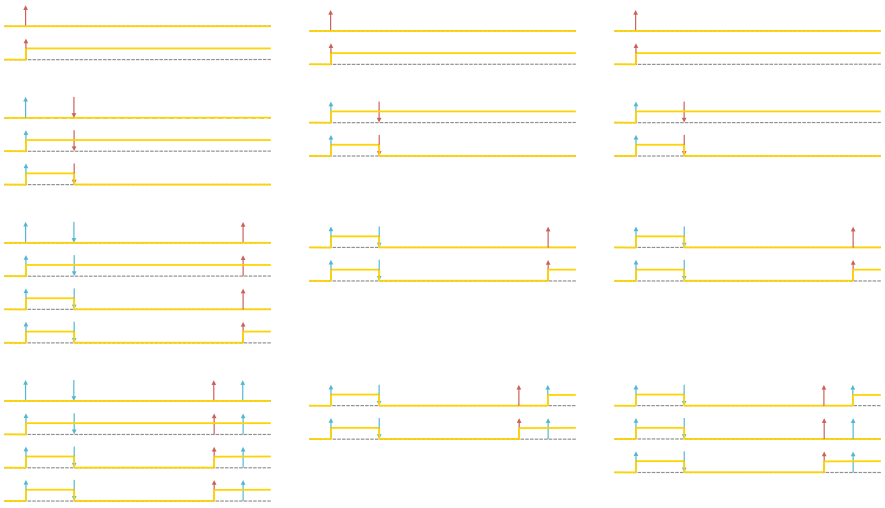
and retraction [7] from the KB to keep the history updated, however it adopt a more streamlined and light-weight algorithm. The difference between the two approaches becomes evident when considering any application domain where, for some reason, the events may be reported with a not neglectable delay that prevents their processing in strict chronological order. On one hand, CEC is a bit cumbersome but robust to unordered events; on the other hand REC is faster but it becomes extremely inefficient when it deals with delayed events. Every time there is a delay in the reporting of an event, a backtracking procedure triggers: the last few operations are cancelled until the system state preceding the delayed event is reverted, then the effects of all the abrogated events are applied again but in the right order and the correct state of the system is restored. Notice that the backtracking procedure may introduce too much overhead in presence of several delayed events, thus it is advisable to adopt the REC only in those domains where it is reasonably safe to believe that events are notified in time.

Comparison between SEC, CEC and REC

In the next few lines, we will present an example to compare the modes of operation of SEC, CEC and REC. We will consider a simple narrative of events that affect a single fluent that is initially FALSE. In order of notification, these events are a *declipping* event, a *clipping* event, another *declipping* event and, finally, a further *delayed declipping* event that actually occurred before the previous event but was notified only after it. These events are summarised and explained in Table 2.2.

Figure 2.3 on the facing page shows the different behaviour of the three variants. Each part of the Figure is devoted to a variant and it is divided into blocks. Each block is dedicated to the processing of an event and it shows the steps that the variant takes to apply the effects of the notified event to the fluent history. The first step only introduces the event that is being notified while the other steps represent the atomic operations that are needed to properly update the

7 A criticism that is moved against these variants is that they are *destructive* as they also relay on `retract/1` – and not only on `assert/1` – to operate: by purging some information from the KB, in fact, it becomes impossible to guarantee any formal property of the calculus.



(a) Simple EC **(b)** Cached EC **(c)** Reactive EC

Figure 2.3: Comparison between the modes of operation of some variants of EVENT CALCULUS.

history of the fluent. Roughly speaking, the more steps are present in a block, the more inefficient the variant is.

Graphic language: - time

Each step is sketched with a compact graphical representation of the events and fluents. The horizontal dashed gray line is the *time axis* in which the time values grow towards right. The events are depicted as vertical cyan arrows: downward and upward arrows respectively represent clipping and declipping events. The event to be processed that has just been notified is coloured in dark red. Finally the history of a fluent is drawn with a segmented yellow line. This line can jump between two levels which represent respectively the TRUE (top) and FALSE (bottom) values. Notice that we do not adopt a special graphic convention to represent delayed events. However it is very easy to spot delayed events because they are coloured in dark red and they have at least another (cyan) event on their right.

- events

-fluents

First event

The notification of the first event produces the same result with all the three variants: SEC, CEC and REC simply change the otherwise empty history of the fluent by setting it to true starting from the time of occurrence of the event.

Second event

The behaviour of the variants starts to differ when the second event is notified. It is a clipping event so we expect to interrupt the validity interval that started earlier. SEC drops the previous result and recreates from scratch an updated history by replaying all the events in the right temporal order: in practice, with a first step it applies again the effects of the previous event, and then the effects of the current event with an additional step. Both CEC and REC, instead, directly update the former state by applying the effects of the second event only. Notice that they require less steps to process the event.

Third event

The third event is a declipping event so it will start another validity interval for the fluent. The behaviours manifested by variants during the processing of the previous event is confirmed and, in a sense, emphasised. Upon the notification of the occurrence of the event, SEC clears the history and plays the whole narrative one event at a time to build the updated history of the fluent. Once again, CEC and REC simply update the past history of the fluent. This brings us to a consideration: while CEC and REC always require a single step to compute the new result, each time SEC takes a number of steps that is proportional to the length of the narrative. Roughly speaking, the longer is the narrative and the more ineffective is the basic variant. Narratives usually tend to become very long.

Fourth event

When we consider events that may be notified with some important delay, the behaviour of CEC and REC also changes. The only effect of the notification of the delayed event is to anticipate the opening of the validity interval that was introduced by the previous event. SEC insists with its usual behaviour: the fluent history is first cleared and then rebuilt by applying one by one the effects of the events that make up the narrative. Now CEC and REC operations are different with respect to the past. On one hand we have CEC which is robust to delays. It manifest the same behaviour as before: the effects of the current event are addressed in a single passage. REC, on the other hand, needs to restore the fluent's state history prior to the time when the delayed event actually occurred. Then it applies back the effects of the delayed event and cancelled events, taking care to following the correct temporal order – much like the SEC generally does. So, generally speaking, REC is usually faster than CEC because it makes less checks thanks to its additional assumptions. However, it becomes slower in a measure that is proportional to the amount of delay with whom the event is notified. An attentive reader may wonder why REC generally outperforms CEC: it is due to the fact that events are generally notified in time and when their notification is in late, the delay is usually small.

Some considerations

Notice that REC handles delayed events by triggering a backtracking procedure. This procedure relies of course on the *backtrack* feature that is available in all the *backward-chaining* tools like PROLOG. The reference platform that we are going to use to implement our versions of EC, however, is a *forward-chaining* tool so it does not provide backtracking. PRSs, in fact, have a shared memory in which they store the facts about the domain and another memory in which they cache rules. The rules are triggered by the stored facts and, as a result, they just change the set of facts. The mechanism with whom PRSs determine if some rule is activated by the facts in memory is extremely efficient, thus we can conclude that approaches similar to CEC are preferable when they are implemented on efficient forward-chaining platforms. In that case it is better to identify all the boundary conditions that may

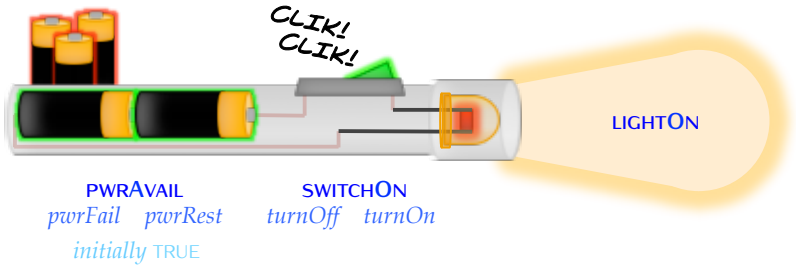


Figure 2.4: A simple flashlight with batteries, a switch and a led diode to project light and the EVENT CALCULUS terms to represent it.

occur during the event notifications and address them with specific rules whose consequence is to apply the required change of the fluent history. Such rules will be discussed in Chapter 2.3.4 on page 44.

Finally, very recently we became aware of an interesting new work on EC by Artikis, Sergot, and Paliouras [15]: due to its recency, we had not the opportunity to set up a proper analysis and performance comparison, however it will be subject of future work.

2.2 THE FLASHLIGHT EXAMPLE

In this section we present a simple problem that has been frequently used in literature as a reference example. It is indeed very compact, but it includes every possible aspect of the EC therefore it qualifies in all respects as a full example.

Description of the problem

This problem is presented in graphical form in Figure 2.4. As the reader can see, the flashlight contains a set of batteries that are wired to a switch and a led diode. When the batteries are charged and the switch is set to on, the led is expected to project out some light. If we want to model this example in EC, we have to provide a complete description of our domain. In other words, we have to identify the complete list of events and fluents that are needed to depict the domain, and we have to determine the initial state of each fluent. If any piece of this information is lacking, the EC can not guarantee meaningful results [8]. The information needed to address this problem is already included in Figure 2.4. The actions with whom we can interact with the flashlight are the following:

PWRFAIL – the batteries are depleted,

PWRRSTR – the batteries have been replaced or recharged,

TURNOFF – the switch has been pressed to open the circuit,

8 It has been shown, in fact, that the EC may return incorrect results if the information about the domain is incomplete [164].

TURNON – the switch has been pressed to close the circuit.

The fluents, whose set of values defines the state of the domain are:

PWRAVAIL – the batteries have some charge,

SWITCHON – the switch is completing the circuit to the flashlight,

LIGHTON – the led is projecting light.

Notice that Figure 2.4 on the preceding page also mentions the initial state of the PWRAVAIL fluent that is TRUE: so the batteries are initially charged. By convention, it is assumed that the initial state of every fluent whose initial value is not explicitly stated is FALSE. Therefore both SWITCHON and LIGHTON are FALSE, meaning that the circuit is initially open and the light is off. Then we have to state all the causal relations between the events and the fluents' values that the events are trying to set. A TURNON event, for example, *initiates* a time interval in which SWITCHON and LIGHTON (but only if contextually PWRAVAIL is *true*) hold. A PWRFAIL event, instead, always terminates both PWRAVAIL and LIGHTON.

Best practices

A table is a convenient way to keep track of this data and to reason about a domain's model: the rows of the table refer to fluents, and columns to events. In addition, the first column always refer to the initial state of the fluents. Now imagine to be a knowledge engineer who is going to define a problem. His work starts by setting up a list of all the fluents that he believes are needed to model the state of the domain. The elements of this list will be the headers of the row of the table. Than he puts the initial value [9] beside each fluent. Later, he prepares as many columns as the number of events that he thinks that may happen on the domain. Each column is labelled with one of the names of these events. Finally, he fills in the cells of these columns. He uses a dash to indicate that the event of that column has no effect on the fluent on the row. An upward or downward arrow tells that the event of that column respectively initiates or terminates the fluent on the row. Any additional condition that must be met in order to apply the effects of an event to a fluent is added beside the arrow. The name of a fluent, for example, means that we expect the given fluent to hold at the time in which the event happens. A preceding exclamation mark negates such condition. Other conditions may be specified as well, such as time instant, intervals or delay (opportunely included in parenthesis). The table that is obtained by applying this procedure to this example's domain is shown in Table 2.3 on the next page. Notice that this best practice can be very tedious, especially for larger domains, but it helps to make fewer modelling mistakes.

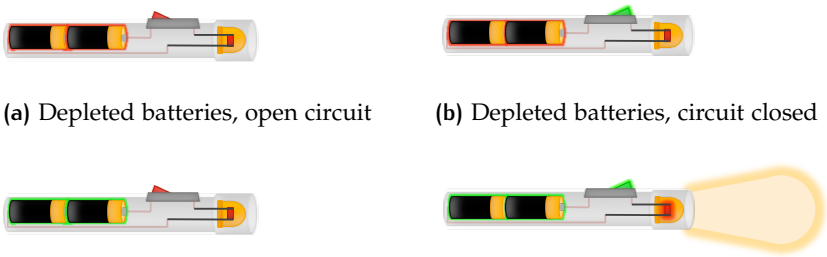
FOL formalisation of the problem

The resulting theory, expressed in terms of First Order Logic (FOL)

9 Notice that this step may be postponed to the end of the modelling process.

		EVENTS			
FLUENTS	INITIALLY	*pwrFail*	*pwrRstr*	*turnOff*	*turnOn*
lightOn	false	↓	↑, *switchOn*	↓	↑, *pwrAvail*
pwrAwail	true	↓	↑	–	–
switchOn	false	–	–	↓	↑

Table 2.3: Synoptic diagram depicting the model of the flashlight's domain.



(a) Depleted batteries, open circuit

(b) Depleted batteries, circuit closed

(c) Charged batteries, circuit open

(d) Charged batteries, circuit closed

Figure 2.5: Admissible configurations on the flashlight's domain.

predicates, is the following:

$$
\begin{aligned}
&\mathtt{initially}(\mathtt{pwrAvail}).\\
&\mathtt{initiates}(\mathtt{turnOn}, \mathtt{switchOn}, \mathtt{T}) \leftarrow\\
&\quad \mathtt{happens}(\mathtt{turnOn}, \mathtt{T}).\\
&\mathtt{terminates}(\mathtt{turnOff}, \mathtt{switchOn}, \mathtt{T}) \leftarrow \mathtt{happens}(\mathtt{turnOff}, \mathtt{T}).\\
&\mathtt{initiates}(\mathtt{pwrRest}, \mathtt{pwrAvail}, \mathtt{T}) \leftarrow \mathtt{happens}(\mathtt{pwrRest}, \mathtt{T}).\\
&\mathtt{terminates}(\mathtt{pwrFail}, \mathtt{pwrAvail}, \mathtt{T}) \leftarrow \mathtt{happens}(\mathtt{pwrFail}, \mathtt{T}).\\
&\mathtt{initiates}(\mathtt{turnOn}, \mathtt{lightOn}, \mathtt{T}) \leftarrow\\
&\quad \mathtt{happens}(\mathtt{turnOn}, \mathtt{T}), \mathtt{holdsAt}(\mathtt{pwrAvail}, \mathtt{T}).\\
&\mathtt{terminates}(\mathtt{turnOff}, \mathtt{lightOn}, \mathtt{T}) \leftarrow \mathtt{happens}(\mathtt{turnOff}, \mathtt{T}).\\
&\mathtt{initiates}(\mathtt{pwrRest}, \mathtt{lightOn}, \mathtt{T}) \leftarrow\\
&\quad \mathtt{happens}(\mathtt{pwrRest}, \mathtt{T}), \mathtt{holdsAt}(\mathtt{switchOn}, \mathtt{T}).\\
&\mathtt{terminates}(\mathtt{pwrFail}, \mathtt{lightOn}, \mathtt{T}) \leftarrow \mathtt{happens}(\mathtt{pwrFail}, \mathtt{T}).
\end{aligned}
$$

The problem as a FSM

As the reader may guess, the set of combinations of all the possible values of the fluents represents the set of its states. The configurations that the flashlight may reach – *depleted batteries, open circuit* (Figure 2.5a), *depleted batteries, closed circuit* (Figure 2.5b), *charged batteries, open circuit* (Figure 2.5c) and *charged batteries, closed circuit* (Figure 2.5d) – are summarised in Figure 2.5. In this fashion, the domain is really like a FINITE STATE MACHINE (FSM) and the events are the ac-

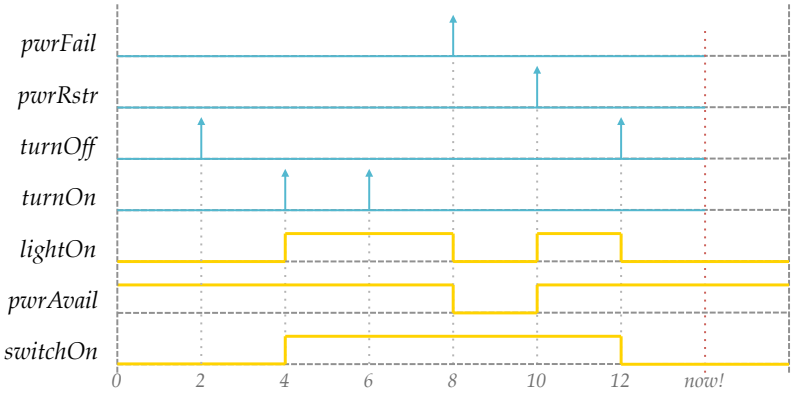


Figure 2.6: A possible trace of execution on the flashlight's domain.

tions that shift the domain from one state to another. The history of the domain, in this case, is the collection of single fluents' histories that are update after the notification of each event.

Graphic language

The Figure 2.6 shows the effects of a sequence of actions on the domain. The Figure is composed of several stripes: each stripe has a tag on the left that tells which entity it relates to. If a strips is coloured in cyan, it relates to one event; if it is coloured in yellow, to a fluent. An event in more detail contains a cyan horizontal line that represents the time flowing (from left to right). From time to time, a vertical upwards arrow resembling a Dirac delta function appears. It indicates that an event of the given type occurred in that very moment. A fluent in more detail contains instead a segmented yellow line. This line runs between two values: FALSE (the lowest) and TRUE (the other). Therefore, depending on the causal information given in Table 2.3 on the preceding page, the occurrence of events can make the fluents to change state.

An instance of the problem

In particular, we start from a configuration in which only the PWRAVAIL fluent is holding (equivalent to the state described by Figure 2.5c on the previous page) and then, every 2 time units, we apply one of the following event: TURNOFF, TURNON, TURNON, PWRFAIL, PWRRSTR and TURNOFF.

1. As the first TURNOFF event occurs, nothing happens since the circuit is already open and we stay in the former state (see again Figure 2.5c on the preceding page).

2. When the first TURNON event occurs, the SWITCHON fluent becomes obviously TRUE as well as the LIGHTON fluent because the batteries are charged; the state in Figure 2.5d on the previous page becomes then the current state.

3. When the second TURNON event occurs, we simply reaffirm the current state [10].

10 Notice that, in general, this is not always true.

4. Then a PWRFAIL occurs and the batteries become depleted (resulting in PWRAVAIL to become FALSE) and the led is consequently off (LIGHTSON is also FALSE); this state is depicted by Figure 2.5b on page 33.

5. A PWRFAIL is the next event to occur so the changes that we have just made are reverted and the current state is again the one in in Figure 2.5d on page 33.

6. The last event of our trace is a TURNOFF which opens the circuit (SWITCHON is FALSE) and turns the light off (LIGHTON is FALSE), like in Figure 2.5d on page 33.

The final state is curiously equal to the initial one (see Figure 2.5c on page 33). It is obviously not necessary that initial state and final state coincide, but it may be a desirable property in certain domains. Also notice that the domain is not passed through all the possible states (the state described in Figure 2.5a on page 33, for example, is never reached), but this is also a coincidence.

2.3 ARCHITECTURAL OUTLINE OF THE TOOL

The following sections provide a detailed description of the architecture of the tool. First we introduce some guidelines that we considered to design the system architecture. Then we explain why we have organised it into two cascading stages and how the component is supposed to work. After that we describe in details the rule sets that are used in both the transformation stage and operational stage. With respect to this second stage, we introduce the case with only Boolean variables and the one with any kind of variables. In both cases, we show how to deal with narratives that may include substantial delay in the notification of events.

2.3.1 Stratification of Terms

Although the EC variant that we have identified is sufficiently streamlined, robust and versatile, it can still be misused and lead to unwanted errors.

Misusing a system

This issue, in particular, is not specific to our implementation, but rather a common problem in computer science. In every software system, in fact, there is sensitive data upon which the application depends to decide what actions to take and which results to return. If this data is not properly protected, it may be changed without complying with the assumptions made for the proper functioning of the system and almost certainly produce unpredictable outcomes. In our case, for example, we use the `clipped/3` predicate to keep track of

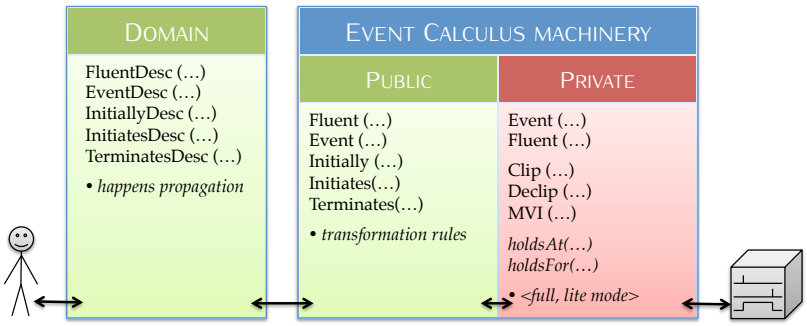


Figure 2.7: Stratification of EVENT CALCULUS terms and definitions for a proper use of the machinery.

any event occurrences that set the state of a fluent to FALSE. Determining whether an event holds in a given time – the kind of answer that we expect to receive from our system – directly depends on this information. So, if this data is left somehow unattended, the user could interfere with it without doing it on purpose. Notice that, with the term "user", we mean both the *final user of an application* and the *application developer* who may inadvertently tamper the calculus by improperly using its objects. This could also affect the *tool developer* since otherwise the bugs could have more serious effects.

For instance, the user could add insignificant data or introduce incorrect changes or even arbitrarily delete some of the available information. In all these cases, the procedure that computes the output of the system would have received an improper set of `clipped/3` notifications, with the result that a plausible but wrong answer is returned. Notice that the user might not notice the error because of the plausible answer and assume the wrong data as valid.

Limiting the misuse of systems

Stratification or, more precisely *local stratification*, is a technique that is used to control the access to the sensitive data of a system [41]. Literally, it refers to the building up of anything in layers. The idea is that each layer provides an interface to the preceding layer by means of which it can be used. In turn, each layer relies on the interface of the layer that follows to perform its tasks. In each layer the variables are local, so their value is not visible from the preceding layer. Therefore, the most sensitive data and the most delicate operations are implemented in the innermost layer which is protected from the user's direct interaction by the more external layers of the system. In this way, the user can only interact with the application through the methods provided by the interface of the outermost layer. This layer is strictly programmed to interact with its inner layer in the proper way and, ultimately, to safely access the sensible data in the inner layers as the chain of calls crosses the layers.

Stratifying the EC

The layers that we intend to use for handling our implementation are sketched in Figure 2.7. As the reader can see, we have 3 layers: the layer closest to to user serves to provide the description of the

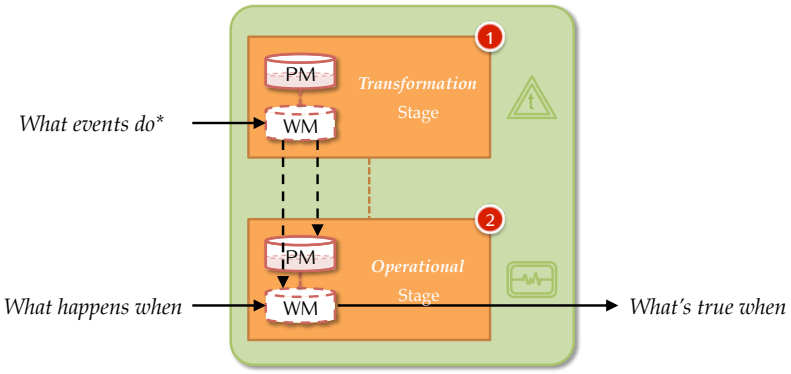


Figure 2.8: Architectural pattern adopted for the tool which shows what subsystem handles the knowledge pertaining to each layer of the stratification.

problem to solve, instead the two innermost layers together implement the EC machinery. The outermost of these two layers contains the definitions of the terms that can be used to define problems: they are, in practice, the interface of the layer. It also contains some rules that translate these terms in a form that is suitable for the innermost layer: they embody the concept of interaction with the following level. The innermost layer, instead, contains the definitions and rules that are necessary to compute the current state of any instance of the problem. The layers that are coloured in red are not directly accessible to the user, while those in green are. In particular, the user is granted read-only access to the innermost green layer, while he has both read and write rights on the outermost since it represents the specification of the problem. Finally, notice that the representation of each layer contains the names of its terms and rules or a generic description of the tasks that they aim to accomplish. The Figure 2.7 on the facing page does not intend to provide an exhaustive discussion on these entities, as all the details about them are available in the following sections.

2.3.2 General Architecture

General diagram and mode of operation

Figure 2.8 summarises the architectural outline of our implementation. As the reader may see, it is organised in two cascading stages. The first stage is called "*transformation stage*" because it takes a representation of a problem in terms of *Java* objects and translates it into a set of corollary declarations and rules. These statements are passed to the second stage named "*operational stage*" as it provides the mechanism that powers the core of the EC machinery. This is actually the stage that performs the monitoring.

Both stages are realised with a PRS so each has a Working Memory (WM) for facts and a Production Memory (PM) for rules. The WM of the transformation stage is initially empty while the PM contains

the set of rules that is discussed in Chapter 2.3.3. As the user asserts problem descriptors into the WM of the first stage, declarations and rules for the second stage are generated. The declarations are notified to the WM of the operational stage and the rules are included in its PM where they join the rules that implements the EC that will be introduced in Chapter 2.3.4 on page 44. As the occurrence of the events is notified to the second stage, the core rules compute the current state of the domain and make it available as output. Notice that all the axioms pertaining to WHAT HAPPENS WHEN and the *initially* axioms of WHAT EVENTS DO are passed as objects to the transformation stage. The *happens* axioms of WHAT EVENTS DO, instead, are directly notified to the operational stage as temporal events. The second stage returns of course the desired information about WHAT'S TRUE WHEN.

Transformation as a pattern

Notice that approaches like this that are organised in stages are quite common and they can be abstracted into a structural software pattern. There may even be more complicated cases in which the number of cascading stages is more than two, or in which the interconnections between stages are more dense. The invariant is that each stage transforms available information to make it available for subsequent stages. Some advanced PRSs like the one that we adopted allow to implement all these logical stages within the same physical instance. In the case of our reference platform, the feature to exploit to achieve such result is called *entry-point*. The entry-points grant the logical partitioning of the WM and therefore they can help in keeping the stages separated.

2.3.3 Transformation Stage

As we have explained, this stage is responsible for the definition of the domain and for its translation in a form that is suitable for feeding the EC machinery. This step is necessary both to hide the more sensible entities that power the EC machinery from the user and to provide a convenient way to deal with the domain definition on the PRS platform that we are using. This tool, in fact, is used to assert and retract *Java* objects from its WM to define a domain, while the EC machinery needs declarations and rules to properly operate it.

Concepts to transform...

As a result of this reification process, we introduce five classes whose meaning is to present the specific events, fluents and fluents' initial states that characterise the domain as well as the contexts in which an event starts or ends a validity interval of some fluent. Listing 2.1 on the facing page contains these definitions. There are a `Fluent` object (line 1) and an `Event` object (line 5), both with a single `name` field. Any instance of these classes means that there is a fluent (or event) in the current domain with that exact `name`. We have an `Initially` object (line 9) holding a reference to a `fluent`, a fluent that is supposed to be initially TRUE. There are also two more defi-

```
1  declare Fluent
2    name : String
3  end
4
5  declare Event
6    name : String
7  end
8
9  declare Initially
10   fluent : String
11 end
12
13 declare Initiates
14   event : String
15   fluent : String
16   condition : String
17 end
18
19 declare Terminates
20   event : String
21   fluent : String
22   condition : String
23 end
```

Listing 2.1: Basic concepts that are objects of the transformation process.

nitions that are very similar to each other introducing respectively the `Initiates` (line 13) and `Terminates` (line 19) objects. They both contain an `event` field, a `fluent` field and a `condition` field. Their meaning is that if the notification of the given `event` occurs, the value of the given `fluent` changes to TRUE or FALSE (depending on whether it is an instance of `Initiates` or `Terminates`) if need be, provided that the given `condition` is verified.

. . . and their meaning

Typical conditions are, for example, to verify whether a fluent holds in a given time instant or interval. Notice that all the fields that we have discussed above are strings, so we assume that they contain valid expressions. Genuine expressions for `event`, `fluent` and `name` fields are alphanumerical sequences of characters, possibly starting with a capital letter. It is also a good practice to append the "`Event`" or "`Fluent`" suffix to the sequence of characters to immediately recognise its qualifying class. A `condition`, instead, is any sequence of "`holdsAt`" and "`holdsFor`" [11] expressions. Their syntax is respectively "`holdsAt( <fluent>, <time> )`" and "`holdsFor( <fluent>, <time>`

11 Notice that, unlike what happens in [**?**], it is trivial to implement the `hodsFor` predicate in a PRS with extended COMPLEX EVENT PROCESSING (CEP) support like *Drools*

`, <time> )`", where `<fluent>` is any valid fluent identifier and `<time>` an absolute time value or `$t`, a symbol that corresponds to the current time instant.

Problem and instance of a problem

We call "*problem*" any set of instances of such reified concepts that provides a general representation of a domain. Problems are translated into sets of declarations and rules that complement the reasoning core of the EC machinery. We call the resulting KB "*instance of the problem*" as it specialises the general problem to a specific case. These cases are characterised by different narratives of events notifications. These complementary sets of **`declare`** and **`rule`** make up a so-called DROOLS RULE LANGUAGE (DRL) resource – a convenient logical unit in which to store a KB that is addressed by specific APPLICATION PROGRAMMING INTERFACES (APIs) of the underlying PRS that ease its loading into any instance of the reasoner.

Obtaining the results

We decided to use a `StringBuilder` object to accumulate the results of the translation process. Notice that the presence of a `StringBuilder` into the WM is also used as the triggering condition for the set of transformation rules. Once the transformation process is completed, the final DRL resource is embedded into the `StringBuilder` that is returned by means of the following query [12]:

```
1 import java.lang.StringBuilder;
2
3 query instances ()
4   StringBuilder()
5 end
```

Listing 2.2: Query of convenience to retrieve the `StringBuilder` including all the embedded resources.

Translating declarations

For example, any `Event` or `Fluent` object is translated into a **`declare`** statement which makes the system aware of the domain-specific `Event` or `Fluent` with the appropriate `name`. In practice, they extend the generic declaration of `Event` or `Fluent` that is provided by the core EC reasoner (not to be confused with the declaration of `Event` and `Fluent` introduced above). As we can see in the Listing 2.3 on the next page, any time a `StringBuilder` and a `Fluent` (line 1) or an `Event` (line 12) are found, a **`declare`** statement is appended to the `StringBuilder`.

Translating initial states

The expressions of initial states can be translated in a similar fashion: `Initially` objects activate a rule which inject the definition of another rule with an empty premise. This rule triggers as soon as the operational stage starts and inserts an instance of the `Declip` object with a reference to the initial time [13] into the WM. As we will see,

because all the Allen's temporal operators are backed and it suffice to apply the *includes* operator between a fluent's MVI and the desired target interval.

12 Multiple instances of the problem may be generated in a single pass by inserting several `StringBuilder` objects into the WM at the same time.

13 The initial time is always assumed as `0` by convention.

```
rule "Include Fluent"
salience 5
no-loop
when
  $i: StringBuilder()
  Fluent( $s: name )
then
  $i.append(String.format(
    "declare %s extends Fluent\n end\n\n", $s));
end

rule "Include Event"
salience 4
no-loop
when
  $i: StringBuilder()
  Event( $s: name )
then
  $i.append(String.format(
    "declare %s extends Event\n end\n\n", $s));
end
```

Listing 2.3: Transformation rules for `Event` and `Fluent` statements.

```
rule "Include Initially"
salience 3
no-loop
when
  $i: StringBuilder()
  Initially( $f: fluent )
then
  $i.append(String.format(
    "rule \"initially %s\"\n", $f));
  $i.append("salience 1\n");
  $i.append("when\n");
  $i.append("then\n");
  $i.append("\tinsert( new Declip($f, 0) );\n");
  $i.append("end\n\n");
end
```

Listing 2.4: Transformation rules for `Initially` statements.

this piece of information is equivalent to the given initial state (see Listing 2.4).

Translating rules

The same process applies to `Initiates` and `Terminates` concepts. Any instance of these classes is converted into a rule whose premise

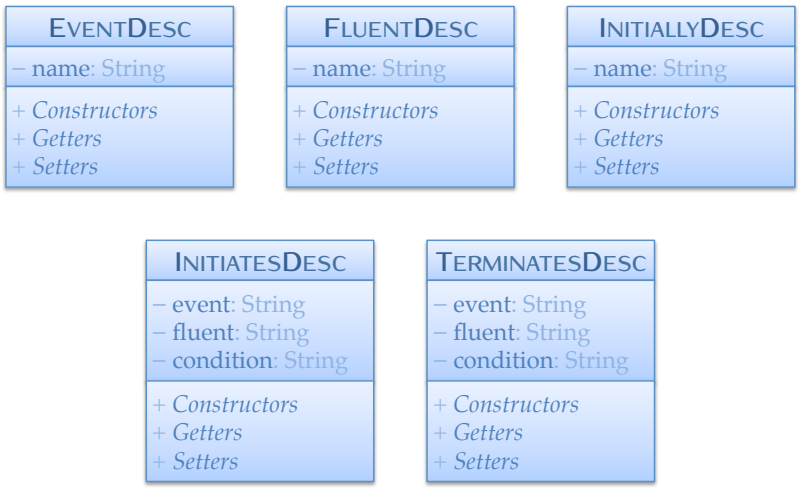


Figure 2.9: *Java* classes at user's disposal as descriptors to connote problems.

identifies a notification of the given `Event`, the given `Fluent` and the given `Condition` – if it is not empty – and the consequence consists in asserting a `Declip` or `Clip` object and retracting the same notification. The `Event` and `Fluent` cited in the translated rule refer to the definitions inside the core stage. Therefore any time a `StringBuilder` and an `Initiates` (line 1) or a `Terminates` (line 21) are found, a **`rule`** statement is appended to the `StringBuilder` (see Listing 2.5 on the next page).

Assumption of uniqueness

Notice that this way of representing the domain is based on the assumption that every concept is unique while the underlying PRS allows for every reified concept to have any number of instances whose fields have the same values. The presence of multiple instances for the same concept does not spoil the correctness of the reasoning but it can lead to issues. The PRS platform does not allow to have several declarations or rules with the same name. Even if we manage to implement a mechanism that makes their identifiers unique – by adding an incremental numeric suffix to their names, for example – we incur in inefficiencies. Multiple copies of the same concept, in fact, introduce redundant declarations and rules which trigger the core stage rules several times, leading to a waste of both memory and computational resources.

Singleton concepts

We addressed this issue by introducing the five *Java* classes that are sketched as simplified UML's class diagrams in Figure 2.9 and an additional set of rules that works on them. The user is free to instantiate any number of these problem descriptors to define the domain. When they are passed to the transformation stage, however, they trigger a set of rules whose purpose is to verify if an equivalent object is already present in the WM. If the check fails, such equivalent object is logically asserted into the WM. Notice that the underlying PRS automatically retracts a logical assertion when the user retracts

```
rule "Include Initiates"
salience 2
no-loop
when
  $i: StringBuilder()
  Initiates( $e: event, $f: fluent, $c: context )
then
  $i.append(String.format(
    "rule \"%s initiates %s (if %s)\"\n", $e, $f, $c));
  $i.append("when\n");
  $i.append(String.format("\t$e: %s( $t: time )\n", $e));

  $i.append(String.format("\t$f: %s()\n", $f));
  if (!$c.isEmpty())
    $i.append(String.format("\t%s\n", $c));
  $i.append("then\n");
  $i.append("\tinsert( new Declip($f, $t) );\n");
  $i.append("\tretract( $e );\n");
  $i.append("end\n\n");
end

rule "Include Terminates"
salience 1
no-loop
when
  $i: StringBuilder()
  Terminates( $e: event, $f: fluent, $c: context )
then
  $i.append(String.format(
    "rule \"%s terminates %s (if %s)\"\n", $e, $f, $c));
  $i.append("when\n");
  $i.append(String.format("\t$e: %s( $t: time )\n", $e));

  $i.append(String.format("\t$f: %s()\n", $f));
  if (!$c.isEmpty())
    $i.append(String.format("\t%s\n", $c));
  $i.append("then\n");
  $i.append("\tinsert( new Clip($f, $t) );\n");
  $i.append("\tretract( $e );\n");
  $i.append("end\n\n");
end
```

Listing 2.5: Transformation rules for `Initiates` and `Terminates` statements.

the object that triggered its insertion into the WM. Therefore the rules in the Listing 2.6 on the next page control the assertion of `Fluents` (line 1), `Events` (line 9), `Initiallys` (line 17), `Initiates`'s (line 25) and `Terminates`'s (line 33), guaranteeing that they are *singletons*. With this addition, the rest of the translation process that we have described above works seamlessly.

2.3.4 Operational Stage

This section is devoted to the presentation of all the rule-bases that we have developed to address the EC. In particular we will see how to implement the versions that adopts only Boolean variables or uses any kind of variables. In both cases, we will show how to deal with domains with negligible or substantial delay in the notification of events.

Boolean Cached Event Calculus

As seen in Chapter 2.3 on page 35, the second stage of our system implements the core EC machinery. In addition to the declarations and rules that are inherited by the translation process, this stage requires a few other statements to work properly.

In this section we discuss the set of statements that realises the version of EC that works on Boolean fluents which can only assume the values TRUE or FALSE.

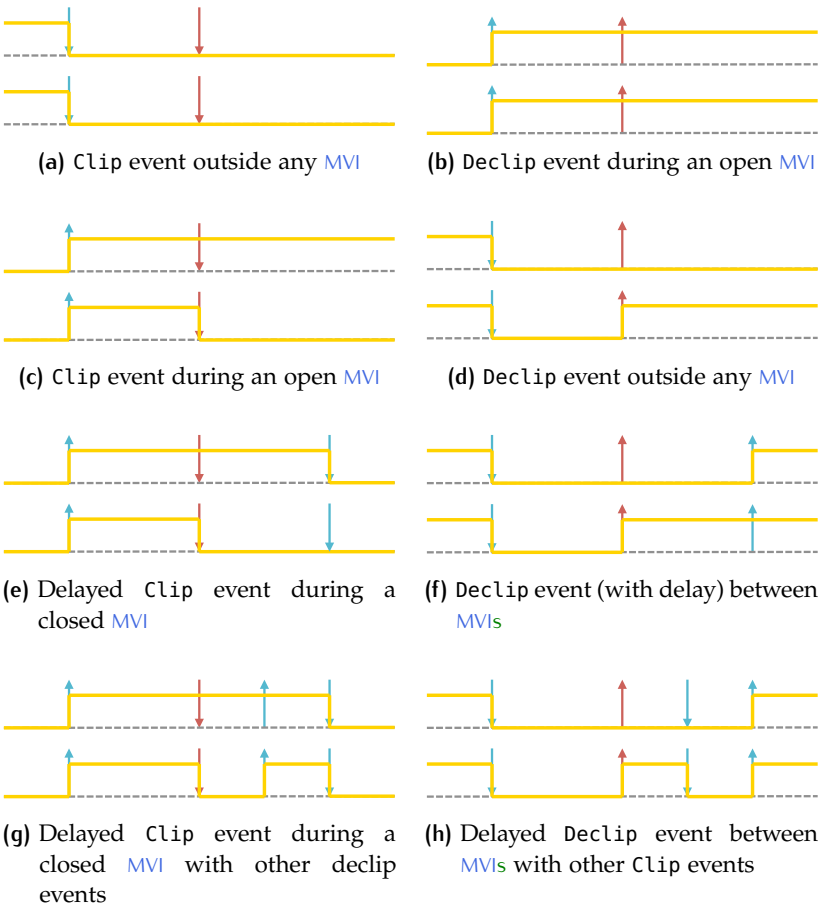
Graphical conventions

In the following paragraphs, we will analyse the configurations of events and fluents that are typically faced in this context. These configurations are summarised in Figure 2.10 on page 46. The Figure includes several cases; for each case there are two diagrams: the top diagram represents the domain state upon the notification of the event to be processed, and the bottom one the new state of the domain that is reached after that the effects of the event under processing are applied. To this aim, we introduce a set of graphical conventions that will be used to describe these configurations. These conventions extend those already adopted in Figure 2.3 on page 29 and in Figure 2.6 on page 34. We use a horizontal dashed gray line to represents the time axis: past instant are on the left and more recent ones on the right. As before, events are drawn as cyan arrows: downward arrows indicate clipping events and upward arrows declipping events. Also as before, the event being coloured in dark red is the event that is being processed. Finally, fluents are sketched by means of a segmented yellow line that toggles between two values – TRUE and FALSE – in reply to events notification.

IRRELEVANT EVENTS It may happen that the notification of an event does not imply any change to the domain's state. This is the simplest case to consider as there is no need to update the fluents' history and

```
rule "Fluent (Singleton)"
when
  FluentDesc( $n: name )
  not Fluent( name == $n )
then
  insertLogical( new Fluent($n) );
end

rule "Event (Singleton)"
when
  EventDesc( $n: name )
  not Event( name == $n )
then
  insertLogical( new Event($n) );
end

rule "Initially (Singleton)"
when
  InitiallyDesc( $f: fluent )
  not Initially( fluent == $f )
then
  insertLogical( new Initially($f) );
end

rule "Initiates (Singleton)"
when
  InitiatesDesc( $e: event, $f: fluent, $c: context )
  not Initiates( event == $e, fluent == $f, context ==
      $c )
then
  insertLogical( new Initiates($e, $f, $c) );
end

rule "Terminates (Singleton)"
when
  TerminatesDesc( $e: event, $f: fluent, $c: context )
  not Terminates( event == $e, fluent == $f, context ==
      $c )
then
  insertLogical( new Terminates($e, $f, $c) );
end
```

Listing 2.6: Rule base supervising the assertion of concepts to ensure that they are unique.



(a) `Clip` event outside any MVI

(b) `Declip` event during an open MVI

(c) `Clip` event during an open MVI

(d) `Declip` event outside any MVI

(e) Delayed `Clip` event during a closed MVI

(f) `Declip` event (with delay) between MVIs

(g) Delayed `Clip` event during a closed MVI with other declip events

(h) Delayed `Declip` event between MVIs with other `Clip` events

Figure 2.10: Initial configuration and outcome of the possible cases of updating the history of Boolean fluents due to the notification of `Clip` and `Declip` events.

consequently we do not need to introduce any rule. This category includes the configurations depicted in Figure 2.10a and Figure 2.10b. They refer to the notification of a `Clip` event outside any MVI and of a `Declip` event inside an MVI respectively. In either case, no action is required because each event tries to impose a value that is already set in the target fluent.

IN-TIME MEANINGFUL EVENTS The second simplest case to consider is presented in Figure 2.10c and Figure 2.10d. This time, the events occur in a context with some boundary conditions that require some changes to the fluent history. In particular, we have cases where a `Clip` event occurs during an MVI and a `Declip` event that instead is not happening during any MVI.

Property of haste

Notice that if we assume that the events are notified with at most a short delay, then all events will be notified by following the proper temporal order. In fact, if the delay of an event is long enough to make the notification slide before some other events, then the chronological

```
1  rule "Clip event during an open MVI"
2  when
3    $e: Clip( $f: fluent, $te: time )
4    $m: MVI( fluent == $f, this includes $e, $im: init )
5  then
6    modify( $m ) {
7      setLength($te - $im);
8    }
9    retract( $e );
10 end
```

Listing 2.7: Handling in-time `Clip` events during open `MVI`s with Boolean fluents.

order of the events is not respected by the notification. In other words, if we know for sure that the delay is negligible than we can conclude that the event that is being processed is always the most recent event appearing on the diagram. Since we are going to refer to this conclusion several times, we have named it *property of haste*. Notice that the validity of this property has a great influence on the way in which the system operates: when this hypothesis is valid, in fact, we can get rid of any notified event just after having processed it, because the observation of the current fluent's state is enough to decide how to update the history.

Therefore, when this property applies and a `Clip` event occurs within an MVI, then we can conclude that this MVI is open since no event other than the one that we are processing has occurred after its introduction. Consequently, the rule that implements this case simply cuts the tail of the open MVI at the time instant in which the event happens. The rule that handles this situation is reported in the Listing 2.7. The premise of the rule identifies the `Clip` event `$e` to be processed, assigning its `fluent` and `time` fields to the variables `$f` and `$te` respectively (line 3). Then it looks for a `MVI` object `$m` for the fluent `$f` that includes the event `$e` and exposes its initial time as `$im` (line 4). The consequence of the rule plans to modify the `MVI $m` by adjusting its duration to `$te - $im` (lines 6–8) and to retract the event `$e` from the WM (line 9).

In a similar fashion, the dual case refers to a `Declip` event that occurs after any MVI that may already be present on the domain. If the *property of haste* is valid, in fact, no event or MVI is more recent than the event under processing. Therefore the expected outcome of such notification is the introduction of a new open MVI that starts at the same time in which the event occurs. The rule that embodies this case is presented in the Listing 2.8 on the next page. The premise of the rule is quite similar to the premise of the dual one. We now consider a `Declip` event `$e` rather than a `Clip` event (line 3) and we

```
1 rule "Declip event (after any MVI)"
2 when
3   $e: Declip( $f: fluent, $te: time )
4   not MVI( fluent == $f, this includes $e )
5 then
6   insert( new MVI($f, $te, Long.MAX_VALUE) );
7   retract( $e );
8 end
```

Listing 2.8: Handling in-time `Declip` events after any `MVI` with Boolean fluents.

verify that there is no MVI that includes `$e` (line 4). The consequent consists of two actions: the introduction of a new (open) `MVI` object with infinite duration (`Long.MAX_VALUE`) at time `$te` (line 6) and, of course, the retraction of `$e` from memory (line 7).

DELAYED MEANINGFUL EVENTS WITH NO INTERLEAVING The configurations that are addressed here are very similar to those that we have introduced in the previous paragraph. We still have a `Clip` event and a `Declip` that are respectively occurring during an MVI and outside MVI, however in this case the *property of haste* does not apply.

Implications of delayed events

This simple difference has two serious implications. The first implication is that it is always possible to find at least another more recent event than the one that we are processing. The second is that it is no longer possible to discard events upon processing because we might need them later to decide how to update the fluent status in case of delayed events. The reasons behind this second motivations will become clear in the following paragraph. It suffices to know for now that the presence of the more recent event involves different actions from those performed in the previous case.

If we consider the configuration of a delayed `Clip` event occurring during a closed MVI as in Figure 2.10e on page 46, the difference is not really evident. We still cut the tail of the MVI and the time of occurrence of the event, but then we do not retract it. The rule that captures this behaviour is available in the Listing 2.9 on the next page. As it is very similar to the one that we have seen before (we just removed the action retracting `$e`) there is no need to further comment on it.

When we look at the dual case described in Figure 2.10f on page 46, the effect of the late notification of a `Declip` event before an MVI is quite different. We have to identify the MVI that is immediately following the event under processing and stretch its duration so that it starts at the same time in which the delayed event actually occurred. The Listing 2.10 on the next page includes the code for the rule that handles this situation. The first part of the premise is similar to the

```
1  rule "Delayed Clip event during a closed MVI"
2  when
3    $e: Clip( $f: fluent, $te: time )
4    $m: MVI( fluent == $f, this includes $e, $im: init )
5  then
6    modify( $m ) {
7      setLength($te - $im);
8    }
9  end
```

Listing 2.9: Handling delayed `Clip` events during closed `MVIs` with Boolean fluents.

```
1  rule "Declip event before an MVI"
2  when
3    $e: Declip( $f: fluent, $te: time )
4    not MVI( fluent == $f, this includes $e )
5    exists Sample( fluent == $f, this after $e )
6    accumulate(
7      Sample( fluent == $f, this after $e, $tt: time )
8      $t: min($tt)
9    )
10   $m: MVI( fluent == $f, init == $t.longValue(), $lm:
         length )
11 then
12   modify( $m ) {
13     setInit($te);
14     if ($lm < Long.MAX_VALUE)
15       setLength($lm + $t.longValue() - $te);
16   }
17 end
```

Listing 2.10: Handling delayed `Declip` events before any `MVI` with Boolean fluents.

premise of the equivalent rule of the former paragraph. In addition, we have to establish that there is at least another event after `$e` (line 5). Notice that we define both the `Clip` and `Declip` events in terms of a common ancestor `Sample`. If such event exists, we iterate (line 6) over all of them (line 7) to determine the one that is closest to the `$e` (line 8). In our hypothesis, this event is a `Declip` so there should be an `MVI` objects `$m` for `$f` that starts there (line 10). The consequence modifies `$m` (line 12) by both setting its initial state to `$te` (line 13) and adjusting its length to `$lm + $t.longValue() - $te` (line 15) if it is not supposed to be infinite (line 14).

DELAYED MEANINGFUL EVENT WITH INTERLEAVING Now, we consider an even more complicated case. As we pointed out while introducing the *property of haste*, we can not retract event notifications after their processing in all those domains where it is reasonable to assume that the events can be notified with significant delay. As we have seen, some events have no influence on fluents history, but anyway they remain latent until the notification of a delayed event does not draw them back in the game. In Figure 2.10g on page 46, for example, we consider the case of a `Clip` event that occurs during a closed MVI, however there is (at least) another event. This event is necessarily a `Declip` event because a `Clip` would have been already processed by one of the rules that we have already discussed. In Figure 2.10h on page 46 is described the dual case where there is at least a `Clip` event between the `Declip` event being notified and the MVI that follows it.

Interleaving

If we generalise this fact, we can conclude that there can be any number of events between of a given type between two other events of the opposite type. Notice that each pair of contiguous events of the same type has no effect on the fluent history, while a pair of contiguous events of different type always produces a change in the fluent state. We call *interleave*, or *interleaving*, the alternation of events of the same type and of different types. Notice that keeping track of interleaving is mandatory because sequences of events of the same type usually do not produce effects, but the delayed notification of an different event can always fall within the sequence and break it into subsections. The events that pertain to each subsection share the same kind, however passing from sequences of one type to another always causes some changes to the fluents.

In addition, it is important to identify the situations where the *interleaving* applies because the actions needed to update the history are different. If we consider again the configuration depicted in Figure 2.10g, for example, we notice that we still need to cut the tail of the closed MVI that includes the event that is being notified, but we also need to add another closed MVI where the *interleaving* occurs. The Listing 2.11 on the facing page shows the code of the rule that addresses this situation. The premise of the rule gains a few additional constraints with respect to the previous case to determine whether an interleaving event exists (line 5).

If so, the rule iterates over the events of the sequence (line 7) to identify the exact time instant in which the *interleaving* occurs (line 8). The consequence of the rule includes an action to modify the `MVI` object `$m` (line 11) by adjusting both its initial time to `$t` (line 12) and duration to `$lm + $te - $t.longValue()` (line 14) if it is not infinite (line 13) and another to insert a close `MVI` object with initial time `$im` and duration `$te - $im`.

Similarly, the dual case portrayed in Figure 2.10h is slightly simpler because it only requires to introduce a closed MVI when a de-

```
1  rule "Delayed clip event during an MVI with interleaving
       "
2  when
3    $e: Clip( $f: fluent, $te: time )
4    $m: MVI( fluent == $f, this includes $e, $im: init,
         $lm: length )
5    exists Declip( fluent == $f, this during $m, this
         after $e )
6    accumulate(
7      Declip( fluent == $f, this during $m, this after $e,
           $tt: time )
8      $t: min($tt)
9    )
10 then
11   modify( $m ) {
12     setInit($t);
13     if ($lm < Long.MAX_VALUE)
14       setLength($lm + $te - $t.longValue());
15   }
16   insert( new MVI($f, $im, $te - $im) );
17 end
```

Listing 2.11: Handling delayed `Clip` events during `MVIs` with interleaving and Boolean fluents.

layed `Declip` event occurs before an MVI where other `Clip` events are also present. The rule that solves this case is very similar to the previous one and it is included in the Listing 2.12 on the next page. The premise of the rule addresses `Declip` events rather than `Clip` events (line 3) and `Clip` interleaving events instead of `Declip` events (lines 5–10), but the overall structure is maintained. The consequent of the rule only assert a new closed `MVI` object to include the effects of the interleaving of events (line 12).

Boolean lite and full mode

The rules that we have presented in the previous paragraphs define two stand-alone complete components implementing the EC with Boolean variables. The two variants address the cases in which the events that are being notified have neglectable delay or with a delay so significant that the notification of events can no longer follow the proper temporal order. Both these components require a common set of declarations that we report in the following Listing only once to reduce the length of the presentation: These declarations include the statements that define the original `Fluent` and `Event` objects (lines 1–2 and 4–8 respectively) that the user extends to identify the domain when the transformation stage takes place. Notice that these objects bear the same name of the concepts introduced in that stage but, as we explained in Chapter 2.3.3 on page 38, they have not to be

```
1  rule "Delayed declip event between MVIs with
       interleaving"
2  when
3    $e: Declip( $f: fluent, $te: time )
4    not MVI( fluent == $f, this includes $e )
5    exists Sample( fluent == $f, this after $e )
6    accumulate(
7      Sample( fluent == $f, this after $e, $tt: time )
8      $t: min($tt)
9    )
10   exists Clip( fluent == $f, time == $t.longValue() )
11 then
12   insert( new MVI($f, $te, $t.longValue() - $te) );
13 end
```

Listing 2.12: Handling delayed interleaving `Declip` events between `MVI`s with interleaving and Boolean fluents.

confused with each other. It also contains the declarations for `Clip` (line 17) and `Declip` (line 20) as an extension of a common generic `Sample` (line 10–15), as well as for `MVI` objects (lines 23–30). Finally two queries complete this set of statements: they identify whether a fluent is TRUE in a given instant (lines 32–35) or interval (lines 37–41) of time.

If we assume that the *property of haste* applies, we can collect the rules that we have introduced while discussing the notification of in--time meaningful events to convey a rule base for the EC on events that is reasonable to suppose that are notified in proper temporal order. We refer to this simpler and more specific EC context with the name of *lite mode*. These rules are exactly the same as before, but nevertheless we report them in Listing 2.14 on page 54 for the reader's convenience.

Now we discuss the more general case that covers events that may be notified with a non negligible delay with respect to their occurrence time and interleaving. This mode of operation which we named *full mode* combines aspects coming from all the situations that we have analysed above. Unfortunately, merging together all the rules that we have discussed so far in a single rule base is not enough. We will discuss below the few changes that were needed to properly address the domain. The resulting rule base for dealing with EC in *full mode* is reported in Listing 2.15 on page 55 and Listing 2.16 on page 56.

The first two rules are equivalent to those in the *lite mode*, as they handle the events that occur at the end of the narrative (lines 1–11 and 13–20). With respect to those rules, we have included a constraint that verifies that there are no events that are more recent that the one being processed (lines 6 and 17). Moreover we have removed

```
declare Fluent
end

declare Event
 @role(event)
 @timestamp(time)
 time : long
end

declare Sample
 @role(event)
 @timestamp(time)
 fluent : Fluent
 time : long
end

declare Clip extends Sample
end

declare Declip extends Sample
end

declare MVI
 @role(event)
 @timestamp(init)
 @duration(length)
 fluent : Fluent
 init : long
 length : long
end

query holdsAt( Fluent $f, long $t )
 exists MVI( fluent == $f,
  $i: init <= $t, $t - $i < length )
end

query holdsFor( Fluent $f, long $ti, long $tt )
 exists MVI( fluent == $f, $ti <= $tt,
  $i: init <= $ti, $ti - $i < length,
  $i: init <= $tt, $tt - $i < length )
end
```

Listing 2.13: Declarations for Boolean EVENT CALCULUS shared by *lite* and *full mode*.

```
1  rule "Clip event during an (open) MVI"
2  when
3    $e: Clip( $f: fluent, $te: time )
4    $m: MVI( fluent == $f, this includes $e, $im: init )
5  then
6    modify( $m ) {
7      setLength($te - $im);
8    }
9    retract( $e );
10 end
11
12 rule "Declip event (after any MVI)"
13 when
14   $e: Declip( $f: fluent, $te: time )
15   not MVI( fluent == $f, this includes $e )
16 then
17   insert( new MVI($f, $te, Long.MAX_VALUE) );
18   retract( $e );
19 end
```

Listing 2.14: *Lite mode* for Boolean EVENT CALCULUS.

the actions that were retracting the events upon processing from the consequents of the rules. Then we have addressed the case of events whose notification is suffering from substantial delay but at least not from interleaving.

Since the rule that handles delayed `Clip` events without interleaving has the same consequent of the first rule above, we have decided to merge them in a single rule (lines 1–11). In this regard, we have relaxed the constraint that verifies that there are no events that are more recent than the event being processes to verify that there is no such `Declip` event (line 6). The rule that handles the dual case of delayed `Declip events` without interleaving (lines 22–38) is exactly the same as before.

The rules that address the cases of delayed events with interleaving (lines 40–56 and 58–70) close the rule base. Notice that they also are exactly the same as those introduced earlier.

Fuzzy CACHED EVENT CALCULUS

Beyond Boolean EC

The two modes of operation presented above are extremely powerful, but possibly not sufficiently versatile. Many EC implementations are limited to Boolean values to describe the state of the domain. This practice is justified by the fact that, in general, it is always possible to decompose the compound state of a domain in simpler terms, until atomic terms (that are actually Booleans) are reached. Although pos-

```
rule "Clip event during an (open) MVI" +
  "Delayed Clip event during a (closed) MVI without
     interleaving"
when
  $e: Clip( $f: fluent, $te: time )
  $m: MVI( fluent == $f, this includes $e, $im: init )
  not Declip( fluent == $f, this during $m, this after
     $e )
then
  modify( $m ) {
    setLength($te - $im);
  }
end

rule "Declip event after any MVI"
when
  $e: Declip( $f: fluent, $te: time )
  not MVI( fluent == $f, this includes $e )
  not Sample( fluent == $f, this after $e )
then
  insert( new MVI($f, $te, Long.MAX_VALUE) );
end

rule "Delayed Declip event before an MVI without
    interleaving"
when
  $e: Declip( $f: fluent, $te: time )
  not MVI( fluent == $f, this includes $e )
  exists Sample( fluent == $f, this after $e )
  accumulate(
    Sample( fluent == $f, this after $e, $tt: time )
    $t: min($tt)
  )
  $m: MVI( fluent == $f, init == $t.longValue(), $lm:
     length )
then
  modify( $m ) {
    setInit($te);
    if ($lm < Long.MAX_VALUE)
      setLength($lm + $t.longValue() - $te);
  }
end
```

Listing 2.15: *Full mode* for Boolean Event Calculus (first part).

```
39 rule "Delayed Clip event during a (closed) MVI with
       interleaving"
40 when
41   $e: Clip( $f: fluent, $te: time )
42   $m: MVI( fluent == $f, this includes $e, $im: init,
         $lm: length )
43   exists Declip( fluent == $f, this during $m, this
         after $e )
44   accumulate(
45     Declip( fluent == $f, this during $m, this after $e,
           $tt: time )
46     $t: min($tt)
47   )
48 then
49   modify( $m ) {
50     setInit($t);
51     if ($lm < Long.MAX_VALUE)
52       setLength($lm + $te - $t);
53   }
54   insert( new MVI($f, $im, $te - $im) );
55 end
56
57 rule "Delayed Declip event after any MVI with
       interleaving"
58 when
59   $e: Declip( $f: fluent, $te: time )
60   not MVI( fluent == $f, this includes $e )
61   exists Sample( fluent == $f, this after $e )
62   accumulate(
63     Sample( fluent == $f, this after $e, $tt: time )
64     $t: min($tt)
65   )
66   exists Clip( fluent == $f, time == $t.longValue() )
67 then
68   insert( new MVI($f, $te, $t.longValue() - $te) );
69 end
```

Listing 2.16: *Full mode* for Boolean EVENT CALCULUS (second part).

sible, it is not always appropriate to reason in this way. Sometimes it is simply more convenient to choose among a multi-valued array of options, be they (sets of) integers, reals, enumerations or even more like strings and ABSTRACT DATA TYPES (ADTs) possible states. All these options, in fact, provide an abstraction for dealing with ranges of values much larger than Boolean. Integers, $\mathbb{Z}$, and reals, $\mathbb{R}$ – for instance – have their own domains. Enumerations are sets of options by defi-

nition. Strings may be such a thing, if they are considered as ordered sequences of characters. And any full assignment of the fields of an ADT identifies a specific configuration among all the possible combinations that are given by computing the power set of the domain of each field.

Many-valued EC and Fuzzy Logic

In addition, if we consider that these richer ranges can be used as domains of *fuzzy sets*, we can imagine to use fluents like *linguistic variables*. This is made possible by the adoption of this richer formulation of the EC on one side and of *Drools Chance* – the Drools variants that supports imprecise reasoning and degrees of truth in rules – on the other. A *fuzzy set* is a pair (U, m) where U is virtually any set and $m : U \to [0, 1]$ a matching function that returns a degree. For each $x \in U$, in fact, the value $m(x)$ is called the *grade of membership* of x in (U, m). Linguistic variables are non-numeric variables that are usually adopted in fuzzy applications. They have the usuals domains, however they define linguistic terms to identify specific distribution of admitted values. A typical example is the linguistic variable *age* which ranges over the possible interval $[0, 125]$ but uses more convenient terms like *young*, *adult* or *old* to identify specific "segments" of age. If we define appropriate matching functions for *young*, *adult* or *old* on the same interval $[0, 125]$, we can determine the degree of membership of each value of age to each linguistic labels and it becomes possible to go from crisp values to linguistic labels (*fuzzification*) and vice versa (*defuzzification*). This very short and simple introduction to *fuzzy logic* does not pretend to be exhaustive, but it is intended to suggest to the reader the great advantages in expressiveness that this technology brings into play. The interested reader may find more details on *fuzzy logic* and PRS in Appendix A on page 153.

Main practical differences

In the following paragraphs we will focus on the differences between this new formulation and we will present the code to process it. A first important shift from the previous version is the type of fluents' values. We have defined them as *Java* `Objects` so that virtually anything – any instance of any class – can be assigned as a fluent's value. This small but profound change also affects the way in which MVIs are handled. In Boolean formalisations, fluents could get only two values so it was very convenient to connote one as *default* value (i. e. FALSE). In this way, it suffices to keep track of only the intervals in which the fluent assumes the other value (TRUE) because the other intervals are obtained by difference [14]. With many values, however, choosing a default value for each possible data type is completely arbitrary since different problems – or even different models for the same problem – that use the same data type, may require different default values.

14 This practice applies only to the positive half of the time axis, where the point of separation coincides with the starting time of the operations on the domain that is usually considered as 0.

Property of membership

In this regard, we have decided to initialise all the fluents' values to **`null`**. A fluent is **`null`** when it is in an undefined state that persists until the happening of an event imposes a specific value to it. This choice leads to an important implication: the history of any fluent is always fully described in terms of MVIs from the initial time on forward. When an event happens, in fact, it may partition an MVI into two MVIs with different values. Such operation, however, does not introduce any temporal discontinuity between MVIs. This leads to the definition of an important property for this EC formalisation. This property, named *property of membership*, states that *the notification of any event is always included in exactly one* MVI. Such property will be exploited to simplify the rules that implement the calculus.

Other practical implications

With respect to the encoding approach that applies to the Boolean case, notice that it is still applicable but it would be an unnecessary complication. If we were embracing this philosophy, in most cases (precisely $n-1$, where n is the number of possible values) we would need a reference to the value itself within the MVI to properly track it. In the remaining case, instead, we would interpret the absence of any MVI as a suggestion that the fluent assumes the default value in the corresponding time interval. In this case, a discontinuity in the temporal sequence of the MVIs means the implicit presence of an MVI referencing the default value. On one side, this approach saves a little memory but, on the other side, it requires more complicated sets of rules to take care of the two different ways of denoting MVIs. Such a memory gain, however, is quite marginal, certainly not as dramatic as in the Boolean case. Roughly, the gain ratio due to this encoding technique is 1 to n therefore it quickly drops from the 50% of the Boolean case towards 0%, as n increases. Another aspect to consider is that more complicated and populated sets of rules, require more computational resources to activate and are more difficult to maintain. The adoption of a PRS mitigates the overhead introduced by larger rule bases, but it does not eliminate it. For all these reasons, we have opted for a single, more compact and efficient ruleset at the expense of a slightly higher memory consumption.

Additional features

Another consequence of this choice is that all the transitions from one value to another are treated in the same way. In other words, all the kinds of notification events that we were used to have collapse in a single type of event. This fact further simplifies the rule-set. In the Boolean case, in fact, we were used to deal with `Clip` and `Declip` events, but now we just have `Samples`. With respect to the events notification of the Boolean case, a `Sample` also contains a reference to the specific value that is going to be set on the fluent. Notice that `Sample` is a private class of this stage, just as `Clip` and `Declip` are for the Boolean case. Therefore the user can not directly interact with `Samples`, but just indirectly through the more external layers of the

tool. In effect, the presence in the `Sample` of a reference to the value to be set in fluent opens up new possibilities, as it is shown below.

Absolute and relative values

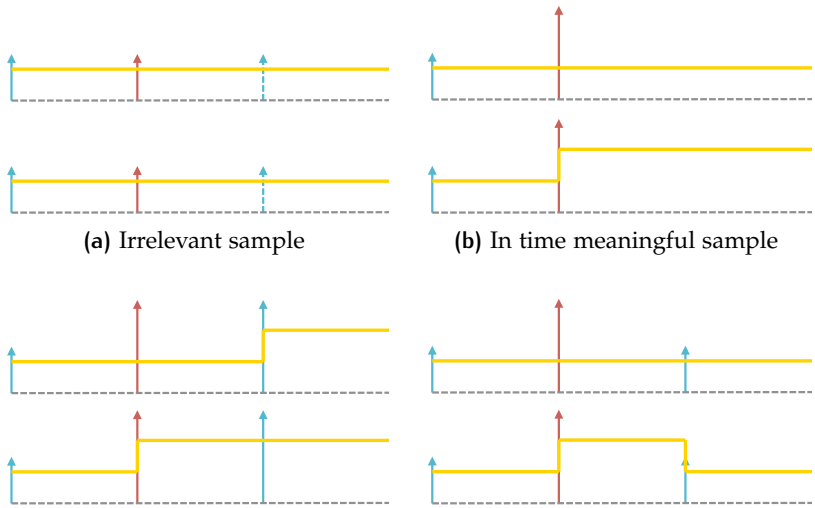
This value can be interpreted as either an *absolute value* or as a *relative value*. `Samples` with absolute values work exactly like the `Clip` and `Declip` that we have seen before: they impose the value that they contain to the fluent. The only difference is that the value is now explicitly maintained in a field rather than being implicit in the type. Relative values, instead, require to be processed before being imposed to fluents. Processing means that the value that they carry must be added to the current value of the fluent to determine the value to set. On one hand, the adoption of relative `Samples` poses some threats: MVIs are always initially set to **`null`** so adding a relative value to this value does not make sense. This problem can be easily solved by checking whether the current value of the fluent is **`null`** before doing the addition: if so, we consider the relative value as an absolute value, otherwise we proceed with the sum. Notice that, despite a further small complication, the adoption of `Sample` with relative values can help to introduce the idea of *trajectories* suggested by the EEC.

We have preferred the `Samples` with absolute values to stick with the simplicity of the solution. Notice that we are not loosing in generality because relative values are treated accordingly, they just require an additional query to retrieve the current value: the provided delta is combined with the retrieved value and an absolute value is issued. The only tricky case would be dealing with relative statements on fluents whose state is still undefined. Possible solutions to this problem are the following. We could constrain the end user to provide an absolute statement first in order to set the fluent's state, but this is an unrealistic assumption. Conversely, we could establish a default initial value for each fluent, but this is equally unreasonable. Ultimately we could simply decide to leave the fluent's state undefined. This last option is more practicable even if it allows longer initial transients. The rules to govern these cases are trivial to implement for a developer which is proficient with Drools syntax. We have not simply because absolute values were already covering all our needs.

Notice that it is possible to have systems where `Samples` with absolute values and with relative values coexist. In both cases, passaging the `Samples` is an external operation that does not interfere with the ruleset for the EC that are presented below.

Graphic conventions

Now we describe the possible configurations that can occur when a new `Sample` event is notified and then we introduce the rules that handle this peculiar formulation of the EC. Figure 2.11 on the next page summarises all these cases. It adopts graphical conventions that are similar to those used before. The time axis is still represented as a horizontal dashed gray line (with increasing values towards right). Cyan upward arrows are still used to indicate when `Sample` events



(a) Irrelevant sample

(b) In time meaningful sample

(c) Delayed meaningful sample with no interleaving

(d) Delayed meaningful sample with interleaving

Figure 2.11: Initial configuration and outcome of the possible cases of updating the history of many-valued fluents due to the notification of sample events.

take place. In this context, however, their length has a different meaning: it gives a rough measure of the specific value that is going to be set on the fluent: the longer the arrow, the higher the value. Since this metaphor is not valid for some value types, we say that the length of the arrow helps to distinguish between values when no proportion between length and value holds. For those types, in fact, it is always possible to introduce a mapping function that linearises the values in the range. The `Sample` that is being notified is still coloured in red. The history of a fluent is represented again by a collection of contiguous MVIs. Their values are used to determine the height of the steps of a staircase function that describe the history of the fluent in a graphical way. The resulting staircase function is rendered as a segmented yellow line as before.

Possible configurations

The Figure 2.11 uses these graphical conventions to convey the description of a few meaningful configurations of `Sample` and `MVI` objects. Each part of this Figure contains two diagrams: the top one presents the fluent history upon notification of the `Sample` in dark red, and the bottom one the history after that the effects of the `Sample` are applied.

IRRELEVANT SAMPLE A `Sample` event that does not affect any fluent is still the simplest case to consider, as no rule is needed to take care of that. A possible example is shown in Figure 2.11a where a `Sample` event (whose values is exactly the same as the valued of the including MVI) is notified. Please notice that, according to the *property of membership*, an MVI that includes the event always exists.

```
1  rule "In-time meaningful sample"
2  when
3    $s: Sample( $f: fluent, $v: value )
4    $m: MVI( fluent == $f, this includes $s, value != $v )
5  then
6    update( $m ) {
7      setLength($s.getTime() - $m.getInit());
8    }
9    insert( new MVI($f, $v, $s.getTime(), Long.MAX_VALUE) )
         ;
10   retract( $s );
11 end
```

Listing 2.17: Handling in-time `Samples`.

IN-TIME MEANINGFUL SAMPLE The second configuration is presented in Figure 2.11b on the facing page. In this example, the value of the `Sample` event is different from the value of the MVI during which the event is received. We can make two assumptions here: the first is again that such MVI always exists (*property of membership*) and the second that the event that is being processed is always the most recent (*property of haste*). These conditions are translated into two statements forming the premise of the rule that handles this kind of configurations. You can find the code that handles this case in the Listing (lines 3 and 4) at the end of this paragraph.

Before that, consider that the *property of haste* applies also to the event that created the MVI that we have just identified, so we can conclude that it must be necessarily an open MVI. Moreover, it allows us to say that the MVI that is going to be added, will be an open MVI too. It follows that the consequence of the rule will cut the tail of the former open MVI at the time instant of the `Sample` event just received and then it will append a new open MVI to the end of the history whose value is the same carried by the event. The consequence of this rule is presented in Listing 2.17 (lines 11–17):

DELAYED MEANINGFUL SAMPLE WITH NO INTERLEAVING The case discussed in this paragraph is sketched in Figure 2.11c on the facing page. The `Sample` event under scrutiny is notified with a substantial delay that allows some subsequent events to be processed before its notification. With this premise, we can only consider the *property of membership* to be valid, but not the *property of haste*. Because of this delay, we can conclude that there is always at least a `Sample` event (and consequently an MVI) following the event that is being examined, or otherwise we would fall back into the previous case.

In addition to those considerations (which are also valid for the paragraph below), we assume that there is no "interleaving" between

```
1  rule "Delayed meaningful sample with no interleaving"
2  when
3    $s: Sample( $f: fluent, $v: value )
4    $m: MVI( fluent == $f, this includes $s, value != $v )
5    accumulate(
6      Sample( fluent == $f, this after $s, $tt: time )
7      $t: min($tt)
8    )
9    $e: MVI( fluent == $f, init == $t.longValue(), value
         == $v )
10 then
11   update( $m ) {
12     setLength($s.getTime() - $m.getInit());
13   }
14   update( $e ) {
15     setInit($s.getTime());
16     setLength($e.getLength() + $t.longValue() - $s.
           getTime());
17   }
18 end
```

Listing 2.18: Handling delayed `Samples` with no interleaving.

the values of the notified event and the subsequent one or, in other words, that they bring the same value. The Listing at the end of this paragraph shows the premise that identifies this configuration: the `Sample` event is retrieved and chained to the MVI *$m* that includes it as before (lines 3 and 4), then the rule iterates over the `Samples` of the same target fluent that follow *$s* to identify the least recent (lines 5–8) and finally its corresponding MVI *$e* is returned (line 9).

With respect to the expected outcome, the consequence of the rule is responsible for cutting the tail of the MVI *$m* at the time in which the delayed event was expected (lines 11–13) and the MVI *$e* is extended towards left to fill the gap between them (lines 12–15). The full content of the rule is available in Listing 2.18:

DELAYED MEANINGFUL SAMPLE WITH INTERLEAVING The last possible case is shown in Figure 2.11d on page 60. Most of the considerations included in the former paragraph apply here too. The only difference is that the value of the `Sample` event is different from the value of the subsequent MVI (the values are interleaving).

Accordingly, the premise of the rule that tackles this case is also very similar to that of the previous case. The difference relies in the statement (line 9) that aims to verify that the values of the `Sample` *$s* and of the subsequent MVI are not the same.

```
1  rule "Delayed meaningful sample with interleaving"
2  when
3    $s: Sample( $f: fluent, $v: value )
4    $m: MVI( fluent == $f, this includes $s, value != $v )
5    accumulate(
6      Sample( fluent == $f, this after $s, $tt: time )
7      $t: min($tt)
8    )
9    exists MVI( fluent == $f, init == $t.longValue(),
         value != $v )
10 then
11   update( $m ) {
12     setLength($s.getTime() - $m.getInit());
13   }
14   insert( new MVI($f, $v, $s.getTime(), $t.longValue() -
         $m.getInit()$) );
15 end
```

Listing 2.19: Handling delayed `Samples` with interleaving.

The consequence of the rule still cares to cut the tail of MVI *$m* (lines 11-13) and, in this case, to assert a new MVI of value *$v* to fill the temporal discontinuity that the first action introduced (lines 14 and 15). The full code is included in Listing 2.19.

Fuzzy lite and full mode

Once again, the above rules may be combined together to provide two organic stand-alone reasoning modules. The first addresses the simpler case in which both the *properties of haste* and the *property of membership* hold and, in analogy to the choices made earlier, we named it *lite mode*. The second addresses the more general case in which only the property of membership holds and attention must be paid to the interleaving of events. This mode of operation is called instead *full mode*.

Both modes requires a few declarations and queries to interact with the module. They are provided only once in the following Listing to keep the presentation short. It contains a definition of `Fluent` (line 1 and 2) and `Event` (lines 4–9) to provide the parent class types for the translation that takes place in the transformation layer. Then it contains a declaration of both a `Sample` and `MVI` object that are needed by the rules. The `Sample` object (lines 11–17) has a reference to the fluent to which it refers, to a value and to the time instant in which it occurs. The `MVI` object (lines 19–27) holds a reference to the fluent to which it refers, to a value and to its initial instant and duration. The two queries that follows, `holdsAt` (lines 29–32) and `holdsFor` (lines 34–38), are used to determine wether a fluent *$f* has a value *$v* at a time instant *$t* or interval [*$ti*,*$tt*).

```
declare Fluent
end

declare Event
  @role(event)
  @timestamp(time)
  value : Object
  time : long
end

declare Sample
  @role(event)
  @timestamp(time)
  fluent : Fluent
  value : Object
  time : long
end

declare MVI
  @role(event)
  @timestamp(init)
  @duration(length)
  fluent : Fluent
  value : Object
  init : long
  length : long
end

query holdsAt( Fluent $f, Object $v, long $t )
  exists MVI( fluent == $f, value == $v,
    $i: init <= $t, $t - $i < length )
end

query holdsFor( Fluent $f, Object $v, long $ti, long $tt
    )
  exists MVI( fluent == $f, value == $v, $ti <= $tt,
    $i: init <= $ti, $ti - $i < length,
    $i: init <= $tt, $tt - $i < length )
end
```

Listing 2.20: Declarations for fuzzy EVENT CALCULUS shared by *lite* and *full mode*.

```
1  rule "In time meaningful sample"
2  when
3    $s: Sample( $f: fluent, $v: value )
4    $m: MVI( fluent == $f, this includes $s, value != $v )
5  then
6    update( $m ) {
7      setLength($s.getTime() - $m.getInit());
8    }
9    insert( new MVI($f, $v, $s.getTime(), Long.MAX_VALUE) )
       ;
10   retract( $s );
11 end
```

Listing 2.21: *Lite mode* for fuzzy EVENT CALCULUS.

Listing 2.21 presents instead the *lite mode*. Although this context is much more complicated than its Boolean equivalent, a single rule is enough to fully manage the case. We have presented this rule in the second paragraph of this section that is about in-time meaningful samples (see Listing 2.17 on page 61), but we repeat it in Listing 2.21 for the reader's convenience.

Finally, Listing 2.22 on the following page and Listing 2.23 on page 67 introduce the set of rules that constitute the *full mode*. The whole mode consists of 3 rules only. The first one is the same rule that we have introduced above for the *lite mode* (lines 1–11); we just modified it to conform to the case of events with delay. In practice we verify that the notified `Sample $s` is the most recent event (line 5) and we do not retract it as it could serve later to assess the interleaving during the notification of other events. The last two rules are exactly the same rules that we have introduced in the last two paragraphs that are about delayed meaningful samples with or without interleaving (see Listing 2.18 on page 62 on lines 13–30, and Listing 2.19 on page 63 on lines 32–46) to handle the cases of events notified with non neglectable delay.

2.4 EXPERIMENTAL EVIDENCES

This section presents the experiments that we have conducted on our implementations of EC. In the first part of the section we introduce the test suite that we have built to empirically asses the correctness of our work. The second part contains instead the result of the comparison between our implementations and their PROLOG counterparts as well as some evidences about memory consumption.

```
rule "In time meaningful sample (interleaving-safe)"
when
  $s: Sample( $f: fluent, $v: value )
  $m: MVI( fluent == $f, this includes $s, value != $v )
  not Sample( fluent == $f, this after $s )
then
  update( $m ) {
    setLength($s.getTime() - $m.getInit());
  }
  insert( new MVI($f, $v, $s.getTime(), Long.MAX_VALUE) )
      ;
end

rule "Delayed meaningful sample with no interleaving"
when
  $s: Sample( $f: fluent, $v: value )
  $m: MVI( fluent == $f, this includes $s, value != $v )
  accumulate(
    Sample( fluent == $f, this after $s, $tt: time )
    $t: min($tt)
  )
  $e: MVI( fluent == $f, init == $t.longValue(), value
      == $v )
then
  update( $m ) {
    setLength($s.getTime() - $m.getInit());
  }
  update( $e ) {
    setInit($s.getTime());
    setLength($e.getLength() + $t.longValue() - $s.
        getTime());
  }
end
```

Listing 2.22: *Full mode* for fuzzy EVENT CALCULUS (first part).

2.4.1 Assessing the Correctness of the Calculus

Before testing the efficiency of our implementation we need to verify its correctness.

Test suite and test cases

In this regard, we have created a very large test suite that contains more than 4,000 individual test cases. Each test case consists of an *initialisation*, an *execution* and a *comparison* phase. During the initialisation phase, a new empty instance of the EC component is created. We have introduced here an additional, special test case which aims to verify that this operation is successful. During the execution phase

```
31 rule "Delayed meaningful sample with interleaving"
32 when
33   $s: Sample( $f: fluent, $v: value )
34   $m: MVI( fluent == $f, this includes $s, value != $v )
35   accumulate(
36     Sample( fluent == $f, this after $s, $tt: time )
37     $t: min($tt)
38   )
39   exists MVI( fluent == $f, init == $t.longValue(),
         value != $v )
40 then
41   update( $m ) {
42     setLength($s.getTime() - $m.getInit());
43   }
44   insert( new MVI($f, $v, $s.getTime(), $t.longValue() -
         $m.getInit()$) );
45 end
```

Listing 2.23: *Full mode* for fuzzy EVENT CALCULUS (second part).

we take a fictional narrative of events and we simulate the notification of one event at a time. Depending on the cases, the events may be notified in the right temporal order or not, as delay may interfere. Finally, during the comparison phase the memory content of the component is processed to make sure that it only contains instances of the objects that correspond to the events that have been notified earlier and to the objects that describe the resulting expected trace. If so, the test case is successful, otherwise it fails.

Testing narratives

We decided to take into account all the possible narratives that can be achieved by building sequences of `Clip` and `Declip` events that are no longer than 5 elements. The narratives obtained in this way are permutations with repetitions of the 2 events above. Each choice between a `Clip` or a `Declip` is independent so, if `l` is the length of a narrative, the number of permutations is 2^l. The events of each of these narratives are associated with a progressive timestamp so they can be considered as unique even though we allowed repetitions during their construction. Notice that since we notify the events of a narrative by following the order in which they appear, we can conclude that they all correspond to narratives in where the delay on the notification of events is negligible. In order to consider all those narratives in which events can be received with substantial delay, for each narrative we generate all the possible permutations of events. In this way, the timestamp associated with each event is still the same and only the order with whom events are passed to the component changes, thus simulating the delay. Since the events are now unique within each narrative, these permutations are permutations with no

LENGTH l	NARRATIVES 2^l	PERMUTATIONS $l!$	TOTAL AMOUNT
0	1	1	1
1	2	1	2
2	4	2	8
3	8	6	48
4	16	24	384
5	32	60	3840
	63	94	4283

Table 2.4: Figures about the amount of test cases in the test suite.

repetitions. If we still denote the length of a narrative with l, in this case the number of permutations is $l!$.

Some figures about the test suite

These figures are summarised in Table 2.4 where the terms *narratives* and *permutations* respectively refer to the permutations with repetitions and without repetitions described above. Since we plan to consider narratives no longer than 5 events, the total amount of test cases that we obtain with the above criteria is given by the following formula:

$$\sum_{l=0}^{5} (2^l \cdot l!) = (1 + 2 + 8 + 48 + 384 + 3,840) = 4,283$$

If we also consider the special test case for the instantiation of the component, the total number of single tests becomes 4,284. Each narrative is processed by a different instance of the EC machinery during the execution phase. Then the content of each WM is checked during the comparison phase. The purpose of this step is to verify that the memory contains only those instances that describe the expected final state for each narrative. Notice that we have prepared a similar set of narratives for the many-valued case by using `Samples` and two integer values (`0` and `1`).

We have implemented these test cases using *JUnit* [15] for all the *modes of operation* introduced in Chapter 2.3 on page 35. The execution of this test suite results in a complete success as no errors are reported.

15 http://junit.sourceforge.net

2.4.2 Considerations on Efficiency

After the empirical verification of the correctness of our implementation, we want to assess its efficiency compared to other solutions as well as its memory consumption.

Prolog interpreters

Our terms of comparison are of course the efficient implementations of CEC and REC which inspired our work. Since CEC and REC only cover the Boolean case, we will compare only the *full* and *lite mode* of our Boolean implementation. As CEC and REC are built on a PROLOG platform, first we have conducted a survey to identify the best open source or freely usable systems that are currently available. Among our findings, the most promising openly avaialable interpreters were: *B-Prolog* [16], *SWI-Prolog* [17], *tuProlog* [18] and *YAP Prolog* [19]. In order to determine which interpreter guarantees the best overall performances, we executed all the PROLOG EC implementations using the same narratives that we prepared for the comparison that are described in the following paragraph labelled "*Test narrative*".

With the first narrative, REC clearly outperformed CEC since it is the most favourable case for the former variant. The second narrative sees CEC to win because it is the worst case scenario for REC. Moreover, the difference in performance is not as great as we supposed before performing the test. In the third narrative, the comparison is tighter and it appears to be the most unbiased case. For this reason, we decided to elect the fastest prolog system among those that were previously identified by only using this narrative. With all the interpreters, REC generally appears to be quite faster than CEC. Among these, *B-Prolog* and *YAP Prolog* are the fastest. More specifically, *B-Prolog* is actually faster than *YAP Prolog*, but due to its aggressive strategy on memory allocation, more than a few executions were not able to complete for lack of free memory. For this reason, we have preferred *YAP Prolog* to *B-Prolog* to perform the rest of the experiments.

Test narrative

So we have prepared a narrative of events to feed the EC implementations. This narrative has to fulfil some requirements: it must be meaningful but not trivial, so that we can easily determine the correct status of the domain in reply to the occurrence of the narrative's events, as well as simple but long enough to appraise the efficiency of each implementation. With respect to these requirements, narrative that we have introduced as an example in Chapter 2.2 on page 31 and shown in Figure 2.6 on page 34 is a good candidate. However, being composed of only 6 events, we had to repeat it a hundred times to get a narrative with a decent length of 600 events.

Starting from this narrative – which is clearly totally ordered as no event is delayed – we have derived two other narratives. In the

16 http://www.probp.com/
17 http://www.swi-prolog.org/
18 http://tuprolog.alice.unibo.it
19 http://www.dcc.fc.up.pt/~vsc/Yap/

first one, the events are rearranged in reversed chronological order to simulate a delay for each event that is inversely proportional to its position in the sequence. In the last one, instead, the events are scrambled in a way that simulates the occasional delay of a few events. In particular, all events in the narrative have equal probability of being delayed to slip after 0, 1, 2, 3, 4 or 5 other events respectively.

Single execution times

The next step of the experimentation is to feed the *lite* and *full mode* with the same scrambled narrative and to compare the results with the figures that we have obtained above. These results are presented in Figure 2.12 on the facing page and Figure 2.13 on page 72. The numbers presented in Figure 2.12, in particular, show the times (in milliseconds) that are required to process each single event. CEC is the slowest system. The *full mode* is sensibly faster but nevertheless it ranks as the third. The second fastest system is the *lite mode* whose events are almost always processed in 1 millisecond. The fastest solution is REC for which the Figure 2.12 reports processing times of 4 milliseconds or, generally, 0 milliseconds.

Clearly, a measure of 0 milliseconds is not realistic. Such value is due to the different scale with whom elapsed time is measured in *Yap Prolog* and *Drools*: *Yap Prolog* uses nanoseconds while *Drools* milliseconds. Therefore the times measured in *Yap Prolog* have been truncated up to milliseconds to allow the comparison with those read in *Drools*, thus introducing the above misunderstanding. If we imagine to round up all the results computed by *Yap Prolog* during the experiments (as it internally happens in *Drools*), all the readings of 0 milliseconds would instead be of 1 millisecond. Since the performance of REC is also characterised by a non-negligible set of readings of 4 milliseconds, we can even conclude that the *lite mode* is faster. However, due to the narrow difference between the two fastest solutions, it is safer to consider the comparison a tie.

Cumulative execution times

The times presented in Figure 2.13 on page 72 are conversely cumulative. Each value, in practice, is obtained by accumulating the time to process all the previous events plus the time required to handle the current one. In this scale, the CEC and the *full mode* are still the slowest. In particular, CEC is faster for problems up to approximately 300 events and then the *full mode* starts to perform better. As the reader can see, the CEC curve is steeper so the *full mode* should be increasingly faster for longer narrative. REC and *lite mode* are still the fastest, of course. In particular, REC seems to outperform the *lite mode* but, again, we must consider that the times coming from the PROLOG interpreter have been truncated up to milliseconds, thus distorting the comparison.

As we have explained for Figure 2.12, each reading of 0 milliseconds should be considered instead as of 1 millisecond. If we keep into account this higher time in all the readings done in *Yap Prolog*, we accumulate up to an additional 600 milliseconds for applying the effects

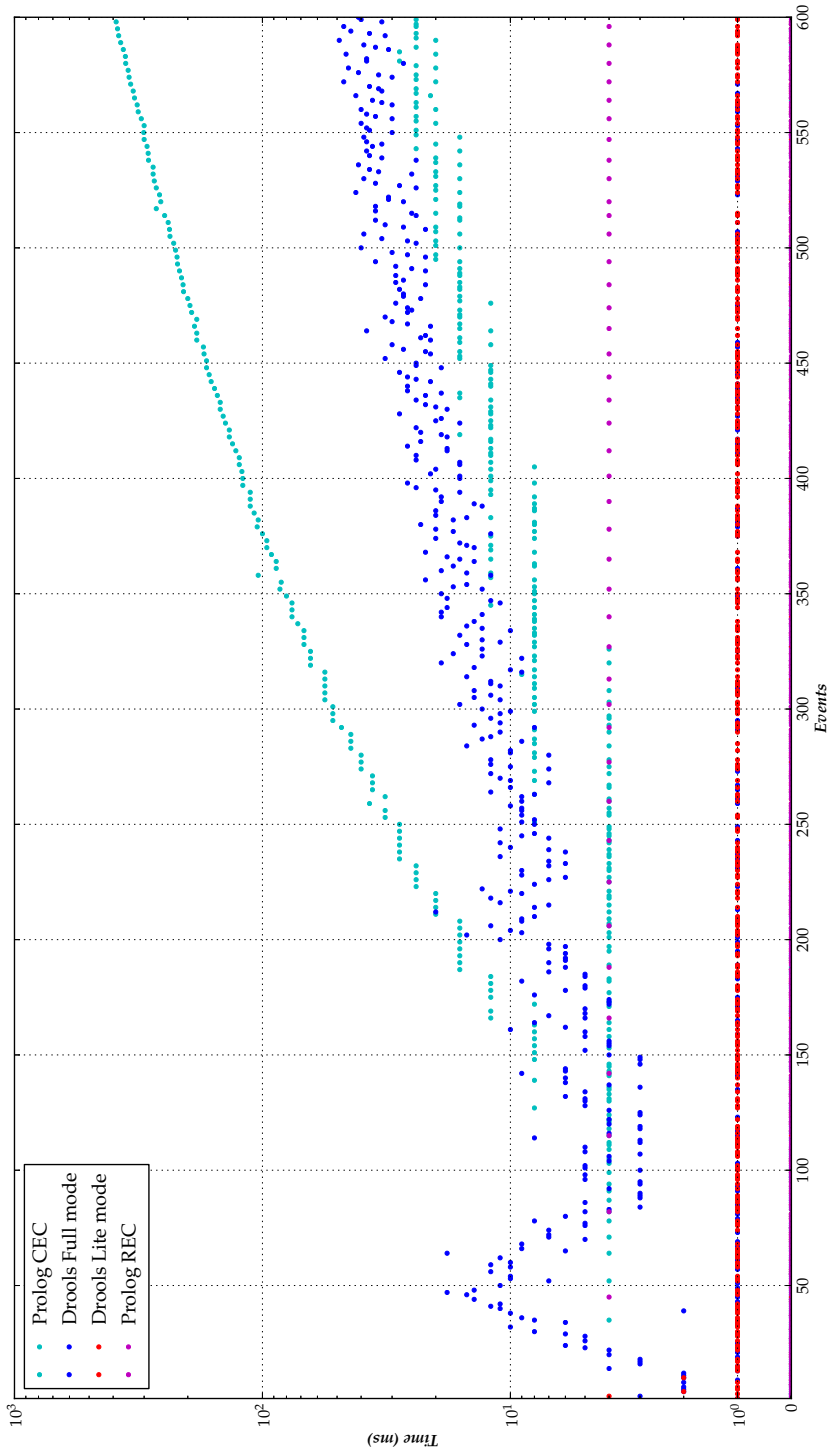


Figure 2.12: Comparison between single execution times of *CEC*, *REC*, *Full* and *Lite mode*.

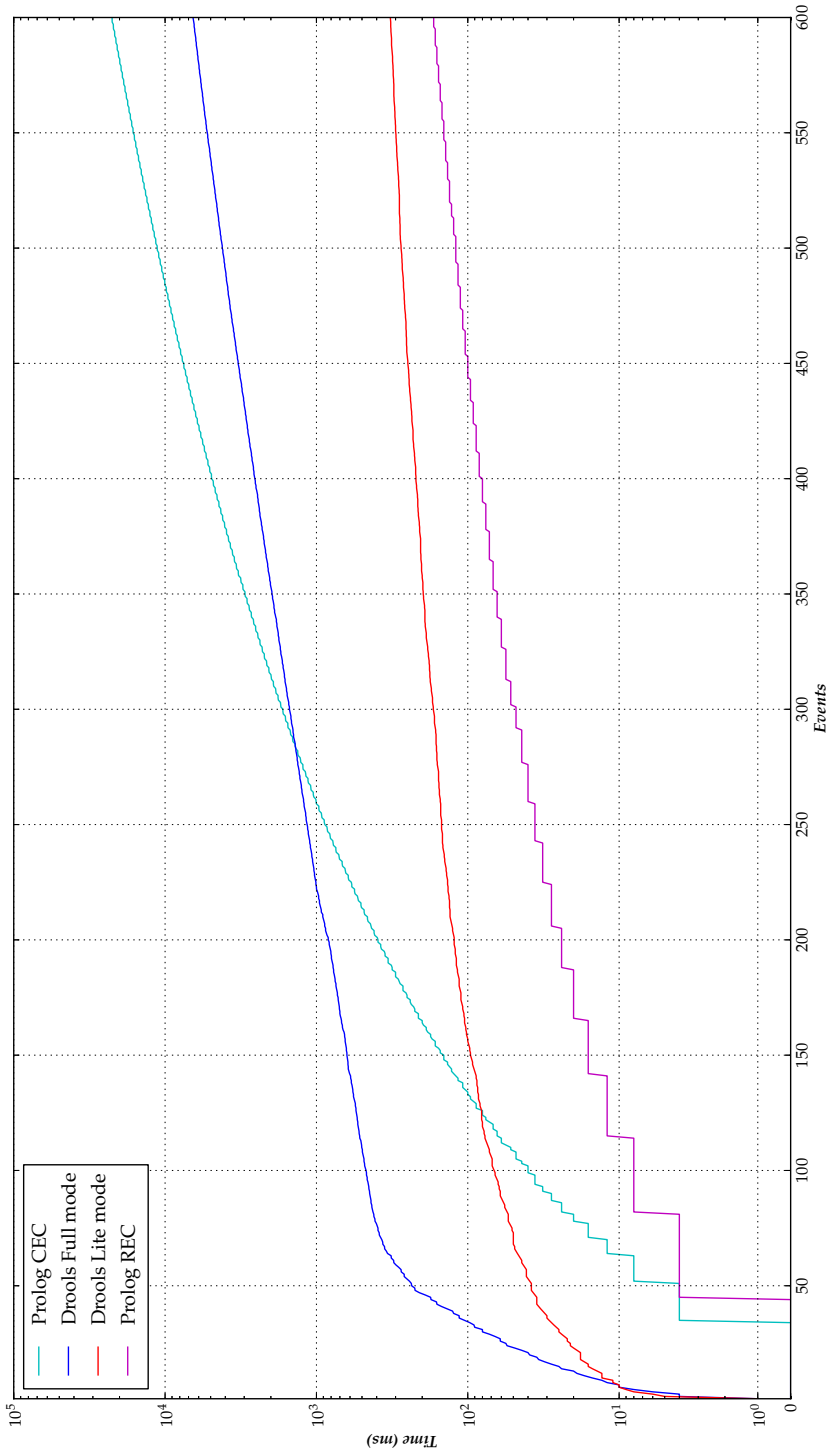


Figure 2.13: Comparison between cumulative execution times of *CEC*, *REC*, *Full* and *Lite mode*.

of all the 600 events of the narrative. However, since the rounding up introduces an average error up to half a millisecond, it is more reasonable to suppose that this execution time needs to be increased of 300 milliseconds. Now, if we imagine to add this figure to the value that we can read in Figure 2.13 on the preceding page for handling all the 600 events of the current trace, we can finally make a fairer comparison. The distance between the running times for the whole narrative of REC and the *lite mode* shown in Figure 2.13 is slightly less than 200 milliseconds: with the above additional 300 milliseconds, REC appears to be slower than the *lite mode* for about 100 milliseconds.

Notice that these considerations may be extended to each event of the narrative. This result corroborates our findings about Figure 2.12. In addition, if we consider the steepness of the two curves (without even correcting the running times for REC), we notice that the *lite mode*'s curve is milder, suggesting that this tool is faster, especially when dealing with long narratives.

Memory footprint

Finally, Figure 2.14 on the following page compares the memory footprint of *lite* and *full mode* only. As the reader can see, the *full mode* has a rather erratic memory consumption while the *lite mode* shows a linear consumption. These different behaviours are due to the retention strategy of events in memory: the *full mode* always keeps data in memory, while the *lite mode* dispose objects from memory when it is safe to believe that they are no more needed (see Chapter 2.3.4 on page 44). The peaks on the *full mode* curve in the Figure correspond to the activation of the *garbage collection* mechanism that frees memory to make room for new events. Similar peaks should also appear on the *lite mode* curve, but later in time because its more parsimonious use of the memory. Since garbage collection introduces overhead, we can conclude that the *full mode* trades its robustness towards delayed events with this overhead.

2.5 SUMMARY

In this Chapter we have summarised the most important aspects of Event Calculus (EC) and we have compared a few variants to identify the most efficient one. Then we have outlined the general architecture of our proposal and we described the way it works. We also showed how the knowledge provided by the user is transformed into a form suitable for the EC machinery. We have also introduced two implementations of this machinery: one uses only Boolean variables, and the other any kind of variables. We also explained how this second implementation implicitly supports Fuzzy Logic (FL). In both cases, we showed how to deal with plain ordered narratives or narratives afflicted by delay in the notification of events. As a concluding step,

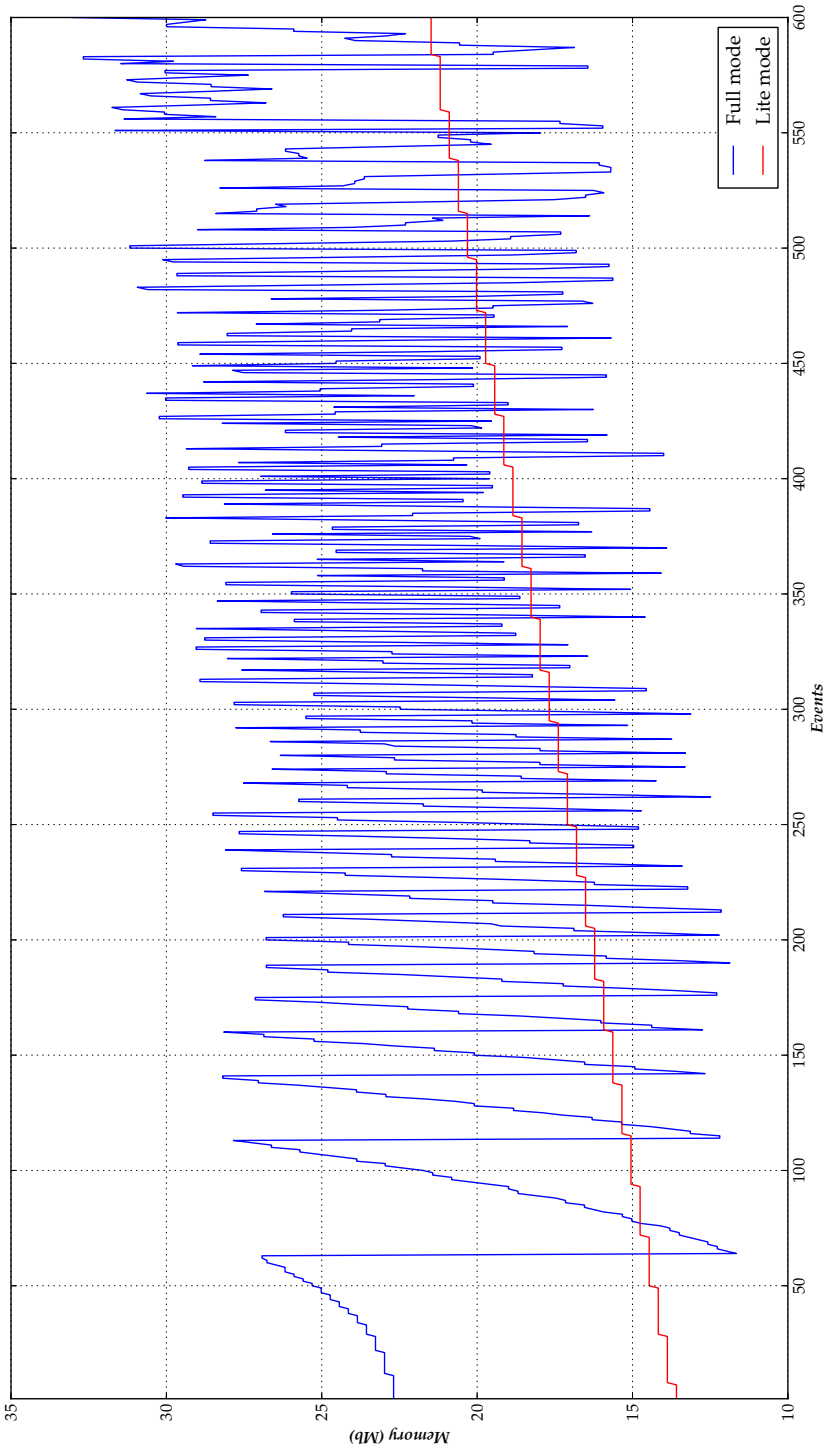


Figure 2.14: Comparison between the memory footprint of *Full* and *Lite mode*, carried out with *Drools 5.3* on an Intel i5 @ 2,4 GHz with 4GB.

we have described the tests that we conducted on these implementations to asses their correctness and to evaluate their efficiency.

In the following Chapters we will introduce other complementary tools that can improve the expressiveness of our system and to check the conformance of the domain.

3 HYBRID REASONING

«For everything you have missed, you have gained something else, and for everything you gain, you lose something else.»

— Ralph Waldo Emerson
American Poet, Lecturer and Essayist, 1803–1882

This Chapter introduces our advanced hybrid reasoner that can perform rule-based reasoning, semantic reasoning and fuzzy reasoning. We believe the novelty of this tool to be undisputed as, to the best of our knowledge, it is the first – and yet the only – system that can perform homogeneously those kinds of reasoning at the same time. It improves the way Production Rule Systems (PRSs) work by introducing operators that can perform subsumption within rules and handle approximation or imprecision within facts. It is not mandatory to use this tool to monitor a system and evaluating its compliance with respect to some desired behaviour, however its adoption introduces interesting new possibilities in this regard. The domain and the desired behaviour, for instance, could be expressed by means of an ontology where the dimensions are expressed in fuzzy terms: this information could be automatically accessed by some other rules to evaluate the conformance of the system.

Notice that it was our desire to also include probabilistic reasoning into the tool: our results, however, were not as conclusive as those presented in this Chapter, so we have opted not to report them here. The interested reader, however, can find a close examination of the work done in this regard in Appendix B on page 183.

3.1 INTRODUCTION AND RELATED WORKS

In recent years, there has been a growing interest in combining ontologies with rules. Many works focusing on the theoretical aspects of such integration have surfaced, sometimes leading to concrete solutions [14, 78, 84, 111, 126, 127]. These solutions, however, usually deal with only "crisp" concepts while real domains typically benefit from fuzzy expressiveness. In a similar way, we have seen the introduction of fuzzy reasoning in the context of both ontologies [173] and PRSs [93, 132].

Integration between rules and ontologies

From a practical viewpoint, despite several rule engines and ontology reasoners are currently available, only a few of them support

semantic reasoning – or, more generally speaking, Description Logic (DL) – as well. *Jena* (and *JenaBean*) [1], for example, includes a generic rule based inference engine that can easily cooperate with the supported reasoners. *Hammurapi Rules* [2] is another Java rule engine that leverages Java language semantics to express relationships between facts as ontologies do. *Algernon* [3] is an efficient and compact *Protégé* [4] extension supporting both forward and backward chaining rules that persists information in ontologies. Another solution is *SweetRules* [5], an integrated toolkit based on *Java* for business rules in the semantic Web, revolving around several standards that are related to both the RuleML and World Wide Web Consortium (W3C).

Integration between rules and fuzziness

This is probably due to the general lack of fuzzy reasoners: despite a few are under development (such as *DeLorean* [6]), *FuzzyDL* [7] seems to be the only mature Java-based solution that we managed to find. Fuzzy Logic (FL), instead, is possibly the only non-Boolean logic which has been integrated in mainstream open source Business Rule Management System (BRMS). Several rule engines currently support FL: most of them treat it "in a broad sense" by supporting various mathematical theories, much fewer implementations consider it as a formal theory that supports the human way of reasoning based a mathematical model [131]. *Clips* [8] and *Jess* [9], two of the first and most famous rule-based systems, have a proper fuzzy extension which is respectively *FuzzyClips* [10] and *FuzzyJess* [11] but, unfortunately, they are no longer maintained. *Drools* [12] natively supports FL as a part of the experimental extension called *Drools Chance* [13] [124] which enhances every formula by annotating it with a degree that models partial truth.

Integration between ontologies and fuzziness

To the best of our knowledge, the only tools that are currently supporting FL statements within ontologies are the already cited *FuzzyDL* [173] and *FuzzyOWL2* [14], a plug-in for *Protégé* by the same authors that provides an intuitive User Interface (UI) to deal with fuzzy ontologies.

Other examples of integration

With respect to the integration of semantic knowledge with other kinds of reasoning, some attempts have been made to exploit the common ground in First Order Logic (FOL) of Logic Programming (LP) and DL by means of LP clauses. The instances of this intersection has been

1 http://jena.apache.org/, http://code.google.com/p/jenabean/
2 http://www.hammurapi.com/
3 http://algernon-j.sourceforge.net/
4 http://http://protege.stanford.edu/
5 http://sweetrules.semwebcentral.org/
6 http://webdiis.unizar.es/~fbobillo/delorean.php
7 http://gaia.isti.cnr.it/straccia/software/fuzzyDL/fuzzyDL.html
8 http://clipsrules.sourceforge.net/
9 http://www.jessrules.com/
10 http://www.nrc-cnrc.gc.ca/eng/projects/iit/fuzzy-reasoning.html
11 http://www.csie.ntu.edu.tw/~sylee/courses/FuzzyJ/FuzzyJess.htm
12 http://www.jboss.org/drools/
13 https://github.com/droolsjbpm/drools-chance
14 http://gaia.isti.cnr.it/straccia/software/FuzzyOWL/index.html

named Description Logic Programs (DLPs) [78] and a specific resolution method has been proposed from them [84]. *dlpconvert* [127], for instance, is a tool the converts the DLP fragment of OWL ontologies to DATALOG clauses. Some techniques to reason on DL within the $\mathcal{SHOIN}$ and $\mathcal{SHIQ}$ families other than the usual TABLEAU algorithms have been proposed as well: some exploit the bottom-up inference method that is typical of DATALOG and Deductive Databases (DDs) [126], others adopt the top-down PROLOG own resolution mechanism to deal with large data sets of individuals [111]. Although not actively maintained, *FRIL* [15] [19] – a PROLOG-like language and interpreter that aims to integrate LP and FL – is indeed worth a mention. Another example of technology that integrates rules with ontology is Web Service Modelling Language (WSML), a formalism of the Web Service Modelling Ontology (WSMO) framework that drops the full compatibility with OWL to support rules [46]. The *MARS* [25] is a framework with a declarative modelling style that focuses on the rule layer of the Semantic Web (SW). Its rather general language has been designed to specify rules for the SW, however, it is not equipped with any specific reasoning technique for SW services.

3.2 DEVELOPING THE TOOL

Loosely- vs. Tightly-coupled approaches

The development of this component has gone through two iterations. Our first idea was to identify a set of flexible and powerful tools, and put them together within a single tool. It was justified by the fact that loosely-coupled architectures usually require a little less research efforts – and a bit more engineering work – than tightly-coupled ones, converging in less time to a usable prototype. Another aspect of no less importance to consider is the expressivity of the resulting tool.

In loosely-coupled solutions, in fact, each tool that is part of the framework can use its own means at full potential, with respect to the relevant domain of application. Conversely, according to whether the domains of application of the single components are completely orthogonal with respect to each other or not, all the joint operations require more efforts to coordinate and complete a single hybrid task. Tightly-coupled solutions, instead, typically compromise on the specific kind of operations to perform on the several domains of application of these reasoning styles. By finding a sort of common ground, it results that the overall expressiveness of the final tool is reduced but its internal architecture is sensibly better.

In the following sections we will introduce both our attempts by describing the loosely and tightly-coupled ones first, and later by justifying such direction.

15 http://fril.sourceforge.net/

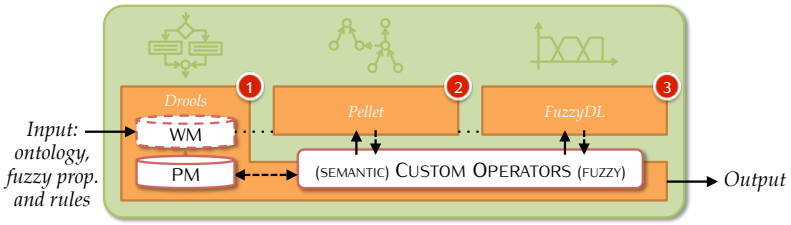


Figure 3.1: System architecture of the tentative version of the hybrid reasoner.

3.2.1 The Loosely-Coupled Hybrid Reasoner

Architecture of the loosely-coupled tool

As you can see in Figure 3.1, our loosely-coupled prototype sports a rules reasoner (Drools [16] marked by the number 1), a semantic reasoner (Pellet [17] marked by the 2) and a fuzzy reasoner (FuzzyDL [18] marked by the 3). It was our intention to give them equal relevance within the final software, however we preferred to use Drools as host component for the others due to its natural operational capabilities.

Dispatching requests to modules

It follows that the RULE-BASED SYSTEM (RBS) takes care of data input and output: the factual knowledge about the domain is loaded in its own WORKING MEMORY (WM) and then propagated to the semantic and fuzzy reasoners. The rules given by the user to manage the domain are loaded as well and included into the PRODUCTION MEMORY (PM) of the PRS. The premises of these rules are allowed to use the subsumptions operators **`isA`** and **`~isA`** [19]. The subsumption is a cognitive process that allows humans to naturally interpret domain entities as instances of more general classes aiding their categorisation. Such process exploits the intrinsic hyponym-hypernym relationship that intervenes between the entities and the categories to whom they pertain or between categories to determine whether a specific concept qualifies as a member of a more general structure. In particular, we use the first operator to deal with crisp knowledge and the latter to cope with imperfect or imprecise knowledge on the reference domain.

Custom operators as points of variability

These non-standard operators that we have managed to include in our prototype. The PRS of choice provides in fact a plug-in mechanism that allows to introduces new operators called CUSTOM OPERATORS (COs). Each CO needs to implement an interface that forces the developer to deal with the operands and to compute the result of the operation on such operands. The operands may be one or two in case of an unary (both prefixed or postfixed) or a binary operator. In our context two operands are required: the first is either any instance or

16 http://ww.jboss.org/drools
17 http://clarkparsia.com/pellet/
18 http://gaia.isti.cnr.it/~straccia/software/fuzzyDL/fuzzyDL.html
19 Notice that the ~ symbol is used by convention to denote fuzzy operators or definitions.

class type present in the WM and the second a class type. Our operation consists in determining which entities or concepts correspond to the given operands and to ask to the proper reasoner (semantic or fuzzy semantic) the answer to return. In other words, any time a constraint involving a CO is evaluated, the proper external reasoner is called synchronously.

Evaluating the solution

This solution, however, proved to be cumbersome and difficult to use and, in the end, very inefficient. With respect to the first issue, the inconvenience in using this component comes from the fact that the user is required to model the domain three times – one for each kind of reasoning supported by the component – and that he must be consistent in his definitions. Notice that this is not a trivial task as the several submodules may adhere to different assumptions.

The semantic reasoners (both in their crisp and fuzzy declinations), for example, work in Open World Assumption (OWA) while PRSs typically work in Closed World Assumption (CWA). CWA, on one hand, is a quite common and a definitely strong assumption that makes the inference to generally derive much information, as anything that is not explicitly stated is presumed to be false. OWA, on the other hand, is an assumption with a more conservative approach where only the information that can be safely derived from data is considered, possibly introducing an undecided group of data for whom it was not possible to determine the truth. This is a limitation that afflicts many hybrid commercial tools as well. Indeed, semantic reasoning is typically performed presuming the OWA, while PRS generally requires rules to be evaluated in CWA: such important difference makes impossible to define a unified semantics for both the reasoning paradigms.

The problem of the consistency among different models of the same domain is partially lessened by the usage of tools that simplify the automatic conversion from one model to the others. This idea has some obvious limitations: semantic and fuzzy semantic models are typically richer than object-oriented models, so richer knowledge bases can always be impoverished but the opposite operation is rarely possible as crucial information would be missing. There are a few tools like the *JenaBean* [20] extension to the *Jena* [21] framework that we have used in this prototype, for example, that allows to pass from a semantic domain to an object-oriented one but nevertheless the conversion is never completely accurate due to the shift of paradigm.

The most obvious lack of this solution, however, is its poor efficiency. Consider that each time a semantic operation is involved, the dedicated tool for these tasks may determine that, due to the changes occurred onto the knowledge base since last execution, some instances or concept are now different than before. In addition to the

20 http://code.google.com/p/jenabean/
21 http://jena.apache.org

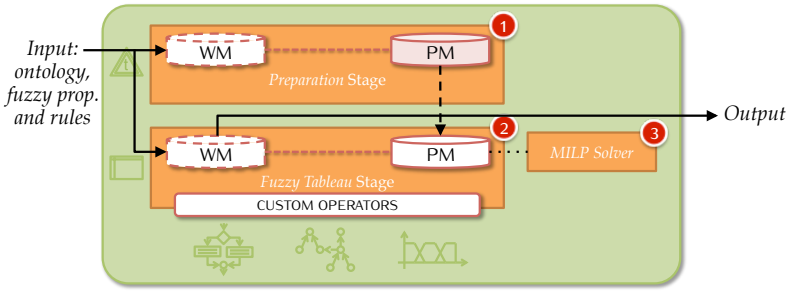


Figure 3.2: System architecture of the definitive version of the hybrid reasoner.

specific answer requested by a CO [22] call, this new information should be propagated as well to the other models. This update, however, may trigger other threads of reasoning that result in additional information that needs to be propagated further. As the reader may understand, simple actions may result in not negligible overhead due to the continuous synchronisation of submodules that is exponentially proportional to the number of them.

All these considerations suggested us to move towards the tightly-coupled solution that is described below.

3.2.2 The Tightly-Coupled Hybrid Reasoner

Architecture of the tightly-coupled tool

Figure 3.2 shows the architectural diagram of our tightly-coupled implementation. In order to reduce the problems that we had identified with the former contribution, we decided to merge the individual models in a single knowledge base to get rid of the representation issues and the overhead due to the dispatching of knowledge across models.

The resulting component has the shape of any common instance of our reference PRS, but includes a subsystem that is dedicated to fuzzy semantic reasoning. The whole component can understand ontologies expressed in a slightly modified *Manchester syntax* [23] to cope with fuzzy declarations, as well as models defined in the typical *Drools Chance* syntax adapted to the fuzzy semantic case. The resulting component has an expressiveness that is comparable with $\mathcal{ALC}^{+}$, the *Attributive Concept Language with Complements* identified by Baader, Horrocks, and Sattler that is the basis of many other DLs [18].

Principles of operations of the tool

If we look inside the component, we can recognise the same lay-

22 Please notice that Drools provides some *Java* interfaces and abstract classes that need to be extended to create a CO. Drools also offer an Application Programming Interface (API) to plug-in any COs and made them available to the user. More details on how to define and use COs is available in the official documentation of Drools (`http://www.jboss.org/drools/documentation`).

23 The Manchester syntax for OWL! (OWL!) was chosen because of its much greater readability by human beings.

ered pattern that we have introduced in Chapter 2.3.2 on page 37. In this case, the first stage is again a transformation stage (marked by the number 1 in Figure 3.2) while the second stage (marked by the number 2) implements a *Fuzzy Tableau* for the given domain. As the reader can see in Figure 3.2), the domain knowledge is loaded into the WM of both stages. The PM of the *preparation* stage is already filled with a set of rules that are equivalent to the steps of a *Fuzzy Tableau algorithm*. These rules are triggered by the domain knowledge and produce a second set of rules that is stored into the PM of the second stage whose purpose is to perform the typical tasks of any (fuzzy) semantic reasoner: to determine whether the model is consistent, to make explicit all the implicit knowledge on the domain that can be safely derived from the model and, lastly, to properly classify the residual facts or domain entities that are present in the WM. Our declarative implementation of the *Fuzzy Tableau algorithm* is freely inspired to the reference implementation by Straccia and subsequent work [28, 112, 172, 173].

Fuzzy logics and Tableaux algorithm: backtracking as mixed-integer linear programming

Readers who have a good knowledge of DL and know how a *Tableau* algorithm works, may wonder how it is possible to implement an algorithm that relies on *backtracking* so much in a system that does not support this feature at all. In a classic *Tableau* algorithm, in fact, each time that an expression is tested there is a sequence of actions that need to be taken to determine the result that best match with the expression. When an instance is going to be classified, for example, the algorithm tries to build the best concept that contains the instance by considering the rest of the knowledge that is available. This process often ends up in impasse , so the last few inconclusive steps have to be reverted and other explanations considered. The semantics of FL, however, relies on continuous ranges rather than plain alternative discrete values.

In a context of Boolean values, for instance, a variable may either be TRUE or FALSE, but not both at the same time.

A fuzzy variable ranges from 0.0 to 1.0, where 0.0 corresponds to FALSE and 1.0 to TRUE. It may assume any value in between (0.35 for example) and be either FALSE with a degree of 0.35 or TRUE with a degree of 0.65 at the same time. In other words, FL always considers each option and its alternatives at the same time, thus not requiring any backtracking. Then, testing a single expression means to test all the possible alternative results at once, or determine the truth degree of each of them.

Notice that at the beginning of this procedure the whole range of values is eligible for each result, so any additional condition that is considered by any given result becomes a constraint that reduces the domain to some extent. Therefore the original problem turns out to be a MIXED-INTEGER LINEAR PROGRAMMING (MILP) problem that we delegate to a third-party component to be solved. This is the reason why the

architecture of our hybrid reasoner includes a dedicated MILP solver – marked by the number 3 in Figure 3.2. The interpretation that is given to the degrees of truth that are returned by the Fuzzy Tableau will be discussed with more details in the following section.

Object-oriented programming and Description Logics: multiple inheritance as introspection

Now, readers that are fond on OBJECT-ORIENTED PROGRAMMING (OOP) may wonder how our reference PRS manages to promote or demote an instance to a given class that is present in the hierarchy of concepts for the domain. Typical implementations of PRS use triples in the form *(subject, predicate, object)* to store the knowledge about the domain, while Drools uses PLAIN OLD JAVA OBJECT (POJO) objects (see Chapter A on page 153 for additional information). With the former approach, the knowledge is "*ground in pieces*" so small that it suffices to retract the triple that binds the entity to its current concept and assert another one that binds it to the new concept to "*move*" the entity from one to another. Notice that the rest of the information about the entity is left unchanged, meaning that the whole operation is not costly. With the latter approach, this is simply not feasible as the operation would involve the updating of the whole object. Many packages like the aforementioned *JenaBean* framework introduce useful approaches to solve the problem by creatively adopting design patterns [67].

At first we decided to follow this route as well, defining a sub-architecture that was making intensive usage of the *Decorate* pattern to provide a flexible method of moving instances across the hierarchy of classes without allocating large amount of memory. This solution required a pre-emptive phase during which all the base classes were identified as well as their possible combinations (to support the multiple inheritance). For each class – basic or composite – all the support interfaces and classes required by the *Decorate* pattern were generated, also providing methods to add and remove the traits of a class to the instance. It is easy to see that this process has an exponential complexity in the number of the basic concepts and thus it is only feasible for small toy-like domains.

We abandoned this solution and moved forward another approach much similar to *traits* or *mixins* like in *Scala* [24]. The concepts are now defined as traits and the instances are simple objects that manage a *key-value* map (where the keys are the names of the fields and values just their values) and a vector of the traits associated with the objects. We also provide a global object that favours the promotion and demotion of instances by assigning or removing traits to an object, and that intercept any *get/set* operation that is redirected to the proper key-value entry of the target instance. This is possible thanks to *Java*'s *introspection* and *reflection*. In this way, objects are still seen as objects and more importantly can be accessed as objects, even if they have been crumbled in pieces like in the approach with triples. Notice

24 http://www.scala-lang.org

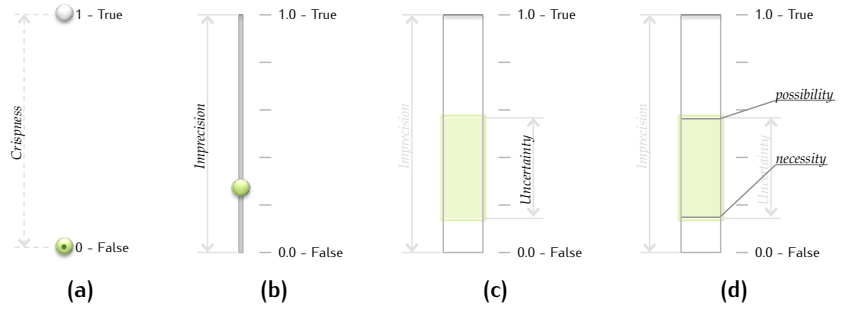


Figure 3.3: Meaning of interpretations: *crisp* values, *imprecise* values, *uncertain* values and "*necessity*" and "*possibility*" of uncertain values.

that when a trait that was exporting some fields is removed from an instance, the underpinned key-value entry is kept for possible later usage. On the contrary, when a trait with new fields is added, the corresponding entries are generated and initialised (if values are provided). As in many Aspect Oriented Programming (AOP) frameworks, fields with the same name but coming from distinct traits are unified only if it is expressly required by the domain.

The Semantics of the Degrees of Truth

Fuzziness in a narrow sense: understanding gradual evaluations

Many works on FL refer the practice to enrich fuzzy predicates of deeper meaning – often indicated as "*fuzzy in a narrow sense*" [131] – that goes beyond their typical usage in control systems [101]. The pictures in Figure 3.3 try to introduce in a simple way a possible usage of the fuzzy degrees of truth to model the vagueness and uncertainty that is required in some domains.

Crisp values

As the reader can see, on the far left (Figure 3.3a) we have a possible assignment of a Boolean value in a "crisp" context. The value may either be TRUE or FALSE (as in this example), so we have a very limited set of alternatives to choose from to model the specific aspect of the domain under consideration.

Imprecise values

If we relax the domain of this value to the range of real values in the interval $[0.0, 1.0]$, we introduce *imprecision* to the model (see Figure 3.3b). If we use 0.24 as a value we mean that the aspect of the domain that we are considering is not exactly FALSE, but something that more or less closely resembles it. This vagueness may be used to model several things such as the confidence of the data, the statistical evidence that we have collected on the data, the sentiment of a community about the data or virtually anything. Notice that, for instance, we say that we consider the domain aspect under examination as false with a confidence of 0.76 or, conversely, that it is true with a confidence of 0.24 with a single assignment. In other words, FL keeps open both the former options by annotating them with the appropriate degree.

Imprecise and uncertain values

Now imagine to determine the value of our target domain aspect according to some other feature of the domain by applying some other operational rule. Initially all the values in the range $[0.0, 1.0]$ are admissible, but then the aforementioned actions may determine that some these values are no more valid and restrict the range. If the process is aggressive enough, the range may be reduced to a single value and we fall back to the previous case. Otherwise, we have determined two values in the interval $[0.0, 1.0]$ that identify the residual range (see Figure 3.3c). Here we use two real values to define the target domain aspect instead of one: this additional degree of freedom allows us to introduce *uncertainty*. It means that we do not know exactly the imprecise value of the aspect that we are modelling, but only where we can presumably find it. This additional option is used for example in Transferable Belief Models (TBMs) to model the fact that we may improve the precision with whom we know the given domain aspect as we revise our knowledge about the model.

Possibility and necessity of a conclusion

The end points of the residual interval may have an additional meaning if we contextualise them to the fuzzy semantic reasoning that we have introduced above. As we have explained, each fuzzy semantic expression in our component is translated into a MILP problem whose solution is delegated to the external MILP solver. Each solution returned by the solver is given in terms of double degrees of truth, the residual ranges that we have just introduced. Each range, in practice, tells how much we can consider an entity to pertain to a given concept – or not. The two endpoints become a measure of this consideration (see Figure 3.3d) [25]: the lower bound (that is closer to the FALSE terminal value) tells us that we can consider the entity as a member of such concept no less than this degree. In other words, it tells us how *necessarily* the entity is related to the given concept. Conversely the upper bound (that is closer to the TRUE terminal value) tells us how easily the instance under examination qualifies as a member of the given concept. In other words it tells us how *possibly* it is related to that concept. In a way, we can say that the **pos** and **nec** operators allows us to respectively retrieve the OWA and CWA degrees of truth of any given statement.

Passing from model to another

Notice that it is always possible to pass from a model with a richer semantics to a poorer representation. Consider the case of the single degree of truth that is used to model vagueness: the continuous range may be flattened to its discrete terminal endpoints and the value that is closest to the given value chosen. This procedure is sometimes assimilated to the so-called *defuzzication* process. We have also introduced two unary CO – **nec** and **pos** – to respectively return the lower and upper bound of any residual range of truth. These operators are used to flatten a two-dimensional model into a single dimensional

25 This formal connection to fuzzy modal logic is still object of study as it is not yet commonly accepted by the scientific community.

representation and pass from a richer model to a simpler one, in a similar fashion of *defuzzification*.

An example of usage is provided in the following section to further clarify the capabilities of our hybrid reasoner.

3.3 EXAMPLE OF USAGE

In the following paragraphs we present some examples with the aim of clarifying the principle of operation of the component that we have introduced in the central part of this Chapter.

An example in eTourism: why OWA is needed

The example below partially revives a use case that we have studied a in the context of an Italian research project focusing on eTourism that also proved to be very useful when trying to explain the difference between OWA and CWA. The full example will be introduced in more details in the Chapter 5.3 on page 119. Tour operators takes advantage of intelligent information systems to propose the right offers to customers. The choice of the right offer comes from a rigorous process of customers and offers classification. We consider now the case of "*advanced accommodation services*" that are basically hotels which provide additional services to the customers. One of this extra services is a shuttle service to bring the customers to some ski resorts and back. An advanced accommodation service is therefore defined as an accommodation service with some employee and a shuttle service operated by an employee to drive customers to ski. Notice that when the hotel owners insert their data into tour operators' databases, they do not know which employee will exactly operate each run of the shuttle service, nevertheless they know that they can offer such a service. This is the typical way of reasoning of OWA. Although many tools allows to read ontologies, they just interpret those definitions as if in CWA with the result that those hotels do not qualify as advanced accommodation services as the driver of the shuttle is not explicitly stated. These systems typically work around the problem by adopting a different model and forcing the user to tick some checkboxes instead.

Semantic definition of concepts

Listing 3.1 on the following page shows an example of how to declare semantic concepts: the concept `AdvancedAccommodation` is derived from the `Accommodation` concept (line 1) and further refined by the requests that a shuttle is needed (line 2, "`== true`" is implicit) and that at least one employee will drive the shuttle (line 5). By using the connective **`and`**, **`or`** and the negation **`not`** it is possible to define any kind of conjunctions or disjunctions of compound concepts, including mutually disjunctive concepts. The concepts `Accomodation` and `Employee` are not included to keep the presentation simple.

Subsumption operators in rules

Because of the above definition, any hotel with a shuttle and at least a person to drive it qualifies as an advanced accommodation ser-

```
declare AdvancedAccommodation as Accommodation
  and HasShuttle
  and HasDriver some Employee
end
```

Listing 3.1: Semantic declarations.

```
1  rule "Advanced Adv. Services to wealthy Customers with
       no family"
2  filter 0.75
3  when
4    $c: Customer(
5          this ~isA Wealthy.class,
6          not this isA Married.class )
7        not exists Customer( age ~seems young,
8                             isChildOf == $c )
9    $o: Offer( this isA
10         AdvancedAccommodation.class and Adventure.class )
11 then
12   EmailSvc.send($.getAddress(), $o);
13 end
```

Listing 3.2: Subsumption operators in action.

vice and the entity is properly promoted to the more specific concept. Now, imagine that we want the information system of the tour operator to automatically send advanced accommodation service that is adventurous offers to wealthy single customers, where adventurous offers are defined as offers involving dangerous activities as extreme snowboarding or kitesurfing, for instance, and posh singles are unmarried customers with no children and a high income. The rule in Listing 3.2 tries to do so. In this case, we want the rule to identify any couple *(customer, offer)* whose elements are of the desired type so that we can automatically send the offer to the customer (line 11). First of all we check that the customer under scrutiny is not married (line 6) by triggering a semantic subsumption reasoning with the CO **isA**. Then we verify that this customer is not the parent of any other young customer (line 7). Notice that the condition on the age of a child is tested with another CO which triggers fuzzy reasoning. It follows that the field `age` has been defined as a fuzzy partition on an integer range. More details on FL within our reference PRS are available in Appendix A on page 153. Last but not least, we require the customer to also qualify as a wealthy customer (line 5): notice that this time we have used the fuzzy semantic subsumption operator **~isA** therefore the evaluation will return a degree of truth rather than a crisp Boolean value. The attentive reader may remember that

Crisp evaluation

Fuzzy evaluation

fuzzy semantic evaluations compute the desired result together with its alternatives, so it happens that any customer qualifies as wealthy possibly with a low degree. In this regard, the **filter** statement in line 2 discards any customer whose degree of wealth is lower than 0.75. With respect to the offers, we do a similar selection by subsumption (line 8) however we explicitly state a composition of concepts as a term of comparison for the **isA** operator. Although the result is similar, in this case we implicitly define a compound concept on the fly and we test the instance *$o* against it.

Taking advantage of gradual evaluation

At last, consider that we could further refine the behaviour of the rule thanks to the use of **pos** and **nec** operators. If we replace line 5 with `this nec ~isA Wealthy.class`, for instance, we mean that we are only interested into customers that are necessarily wealthy at least at a degree of 0.75. We obtain a more evident difference in the behaviour of the rule if we replace line 8 and 9 with the following:

```
9   $o: Offer( this pos ~isA
10        AdvancedAccommodation.class and Adventure.class )
```

In this case, in fact, we allow a wider range of offers to qualify for the advertising as we consider valid all the instances that *potentially* qualify as advanced accommodation services that are adventurous.

This tool is still work in progress and it is currently incubated within *JBoss* repositories as a collateral *Drools* open source project. The intention is to make available for download as soon as it becomes enough stable. Some ideas presented in this Chapter have managed to reach the mainstream development branch of the official *Drools* distribution and will be released with the next upcoming major release of the software.

3.4 SUMMARY

In this Chapter we have presented our state-of-the-art research on hybrid reasoning that combines rule-based, semantic and fuzzy reasoning styles. We first reviewed the research done in this regard, identifying projects and technologies . Then we have described the development process that led to the implementation of our original contribution in the field, emphasising its limits, the way in which we managed to overcome them and its remarkable features. The first version was based on a loosely-coupled approach that was easier to achieve but plagued by poor ease of use and performance. The second final version was oriented towards a tightly-coupled approach that required a substantial research effort and properly addressed the former limitations. Finally we presented the fuzzy semantics of the module and a selection of explanatory examples.

This components represents the results that we have achieved in our effort to make the way of reasoning of a PRS more similar to the human beings one, both in terms of expressiveness (with fuzziness) and modalities (with subsumption). We have taken extra care to make the component as much self-contained and modular as possible so that it can effectively cooperate with the tool that we have introduced in the previous Chapter and the one that we will introduce in the following one.

4 EXPECTED BEHAVIOURS

«Blessed is he who expects nothing, for he shall never be disappointed.»

— Alexander Pope
English Poet, 1688-1744

In the last ten years, we have seen the prospering of models and technologies for developing, deploying, and maintaining Information and Communication Technology (ICT) systems based on heterogeneous and distributed components. Moreover, paradigms such as Service-Oriented Architectures (SOAs), Web Services (WSs), Cloud Computings (CCs), Business Process Management Systems (BPMSs) have been already largely adopted by the ICT industry. When looking at the medical and healthcare context, Computerised Clinical Guidelines (CCG) [44], care plans, and clinical decision support in general aim to ensure that care standards can be implemented reliably and effectively.

All these solutions – and many others as well – allow for increasingly complex systems, while the adoption of standards pushes for the use of heterogeneous, third-party software and hardware components. As a consequence, assuring the correct behaviour of the complex processes that take place in such systems is becoming a more and more difficult task. To this end, approaches based on the notion of *conformance* have been proposed. Roughly speaking, the *expected* behaviour of a process is specified a-priori, by means of some formal language. Then, the complex system is observed at run-time, and the behaviour that is observed externally is compared with those *expectations*. In case they are not met by the observations, some alarms or managing procedures are triggered. This contradicts the opening quote of this Chaptersince, from out point of view, expectations can be used to determine whether a system is behaving correctly and, possibly, to nudge it towards the right direction, if it is deviating from there.

In this Chapter, in fact, we introduce the concept of *expectation* as the concept around which our criteria for the global conformance of the complex processes revolves. The rest of the Chapter is organised as follows: we first better contextualise the problem by discussing some related works and then we introduce expectations as the most iconic concept of Event-Condition-Expectation rules (ECE-rules), an evolution of the well-known Event-Condition-Action rules (ECA-rules)

towards the evaluation of conformant processes. We also discuss the concepts that are needed to properly handle expectations and the changes that are required to enable them within the definition of the standard rules of a Production Rule System (PRS). An example is provided as well to further clarify these concepts. Then we describe the architecture of the tool, the metamodel behind it, the typical life-cycle of an expectation and, last but not least, the set of static and dynamically generated standard rules that are needed to seamlessly handle expectations. In the last part of the Chapter we present our idea of global conformance as a user-definable function to compute a conformance score of a process. We also show how to take advantage of other forms of reasoning to further refine the conformance evaluation. In particular, the self-contained modules that we have presented in this dissertation – or even any other third-party component – could be used, provided that they return their evaluations as degrees of truth (see Chapter 3.2.2 on page 85). In this regard, we also show how fuzzy temporal evaluation is addressed as it is an aspect that was not properly covered by any of the other tools that we have introduced in the previous Chapters. The procedure discussed here could also be used to address similar complex cases that we have not considered in this work. The Chapter is concluded by an advanced example that shows how the same problem considered before could be solved by considering the contributions of several custom evaluations to compute the overall global conformance score.

4.1 INTRODUCTION AND RELATED WORKS

Assuring that a complex process is progressing correctly is a task that is becoming more and more difficult. The traditional debugging techniques alone, in fact, may not be sufficient because of the intrinsic complexity of the overall system or because it is not possible to interact with some of its parts, either because they involve third-party components or human tasks. In these contexts, the Just-In-Time (JIT) monitoring techniques could contribute in some way since they verify whether the system behaves correctly while it is executing.

Beyond traditional approaches

What typically happens is that the developer specifies in advance the correct outcome that the system should exhibit in reply to some input or sequence of inputs. By observing the inputs, a JIT monitor tries to guess what effects are produced on the system in a time that approximates the real-time. Then the output of the system is compared with the excepted outcome to check whether they match. A positive answer means that the system is deemed to be *conformant*. This approach can also be exploited to verify the conformance of complex processes against some high-level constraints or specifications. The Quality of Service (QOS) criteria, for instance, must be continuously

monitored and proper actions must be taken any time the system deviates from the standard behaviour. Legal aspects, medical guidelines and even business constraints could be subject to monitoring as well.

Conformance as monitoring of expectations

A common idea is to use rules in the most general sense, to express the system's desired behaviour. These rules are exploited to codify both the conditions under which a behaviour is manifested – be they a specific set of inputs or peculiar pattern describing the system current state – and the expectations about the system outputs. Our proposal is to extend rules to naturally support conditions and expectations when analysing and handling a system's deviations. This idea, however, is rather consolidated and shared among different fields that are related in some way to the computer science. In all these domains, the (close to) real-time process of systems monitoring is regarded as a possible solution to assess the conformance between them and the expected behaviours.

Approaches in literature

In [6], for instance, the deontic concepts and operators that can be used to represent norms, obligations and similar concepts have been identified and converted into Abductive Logic Programming (ALP) theories. Although supported by a different scope and motivations, a common background is introduced that also fits with the semantics of our proposal. In the field of Multi-Agent Systems (MASs), instead, a few social approaches are been defined to specify which interactions among agents are admissible in terms of expected behaviours. These approaches also define accomplishment or transgression of behaviours in terms of deviation from the expected. The framework SCIFF [5], for example, mainly focuses on a logic-based notion of expectations and their fulfilment or violation. The *commitments*, as throughly investigated by Singh et al. [53, 167, 178] or Fornara and Colombetti [64], are considered as promises coming from agents' interaction where "*debtor*" agents become committed towards a "*creditor*" to bring about a given property, meaning that it is expected to make it true. In the context of Business Process Modelling (BPM), van der Aalst et al. propose declarative languages focusing on the properties that the system should exhibit. The DecSerFlow language [186], for instance, allows the users to specify which business activities are expected or prohibited to execute as a consequence of the execution or prevention of other executed activities. In the domain of legal reasoning and normative systems, Governatori and Rotolo [73, 74, 76, 77, 156] propose temporal logic frameworks and languages to represent legal contracts between parties: these tools are primarily focusing on compliance issues and simulate the possible course of actions of a system to evaluate whether the contract agreements are indeed respected.

Conformance in a broader sense

Most of these approaches, and many others, are implemented in a way to provide only a boolean answer to the problem of conformance. In other words, the question that these system are trying so answer is *whether the observed behaviour is conformant with the expected behaviour*,

and such answer may only be *yes* or *no*. According to our everyday experience, however, a richer, more informative answer is generally required in most cases. Sometimes, in fact, a given delay in a temporal deadline is still acceptable, only meaning that an execution is more or less flawless. Other times, the place in which an action is executed may settle in a plain success, a failure or something in between. Finally, there are cases in which an evident deviation from the expected behaviour may lead to a satisfactory evaluation or, conversely, a proper execution may not qualify as well as expected due to other contextual reasons.

In this regard, we propose to decouple the conformance problem from the representation of the answer and to dynamically bound by a function which singularly measures each context. In particular, we propose to replace the over-simplifying Boolean answers in favour of gradual answers such as degrees in the interval $[0, 1]$. Thanks to this assumption, it is possible to introduce several custom evaluators which provide answers in this range by considering all the different aspects that are involved in the specific context. Then, a flexible and robust methodology is exploited to combine these single contributions and provide an overall gradual evaluation of the conformance problem.

The following example may help to ground the discussion: consider a robot that is capable to move autonomously and suppose it has to reach a certain position within a given time limit. The robot may decide to move slowly but accurately, arriving exactly where you want even if slightly late. Conversely, the robot might decide to reach the destination as quickly as possible, even at the expense of the accuracy of its final position. In both cases, a Boolean evaluation of the conformance of the robot's actions with respect to the expected outcome would be a failure, while it would be almost a success (a value relatively close to 1.0) if we were using the degrees of truth. In particular, such result would arise from the combination of the evaluation of the robot's spatial and temporal performance: the inverse of the distance between the final and desired position for the spatial dimension of the task and 1.0 (if in time) or the inverse of the delay for the temporal dimension of the task. Moreover, consider that additional context information – such as the fact that no route is possible from the current position to the given destination – could lead to evaluate the fact the robot did not move at all as a full success.

4.2 EVENT–CONDITION–EXPECTATIONS RULES

The ECE-rules have been imagined by loosely following the idea of ECA-rules. ECA-rule is a term that is used to refer to the typical structure of *active* rules, as in Event-driven Architecture (EDA) and Active

DATABASE SYSTEM (ADS). As suggested by their name, these rules traditionally consists of three parts:

EVENT – the signal that triggers the invocation of the rule,

CONDITION – a logical test that, if successfully passed, it causes the action to be executed,

ACTION – a set of updates or invocations on the contextual data.

This distinction was introduced by the early research on ADSs where the term ECA-RULE appeared for the first time. ECA-RULEs can also be handled by rule engines that are based on advanced variants of the RETE algorithm for processing rules [59, 160]. Those engines, in effect, often natively support the concept of events, perform tests of local data to assess conditions and properly restructure the object attributes as actions. In ADSs, the condition is typically a query to the database whose result (if not trivial) is passed to the action as a parameter to update the database. In either cases, data updates are considered as internal events: as a consequence, the execution of a ECA-RULE's action can match the event of another ECA-RULE, thus triggering it if the condition is met as well.

From actions to expectations

As the reader may guess, ECE-RULEs are similar for two thirds to ECA-RULEs: they are invoked as well by some external signal (the event) and their execution is subject to some logical condition. The difference between them relies in the last part of the rule which expresses expectations on the behaviour rather than actions. Here the operative part of the rule is used to outline the directions in which the system is expected to evolve from the snapshot of the domain that is identified by the happening of the event and the fulfilment of the condition. The concept of expectation is general and orthogonal with respect to the definitions of deviations provided in Chapter 1.2 on page 5. In other words, exceptions (both implicit and explicit) and anomalies (such as operational errors or even frauds) can be modelled in terms of expectations.

Adapting rules to handle expectations

From a practical viewpoint, our implementation of the ECE-RULEs quite closely resembles the typical structure of *Drools* rules. We have extended the original parser of *Drools* to make it understand both the regular expressions and the new definitions of expectations. Standard statements do not require additional processing and are directly passed to the underlying *Drools*' engine. The information about expectations, instead, is redirected to a dedicated software layer for further interpretation (see Chapter 4.3 on page 98).

The grammar of expectations

As the reader can see in Figure 4.1 on the following page that presents the production rules to parse ECE-RULEs expressed in EBNF syntax, in fact, the default parsing strategy of a ⟨*rule*⟩ is maintained unaltered. This is not the case of the rule ⟨*consequence*⟩ which is overwritten by a production which seeks to identify a (possibly empty)

⟨*rule*⟩ ::= 'rule', ⟨*id*⟩, ⟨*attribute list*⟩,
'when', { ⟨*pattern*⟩ }, 'then', ⟨*consequence*⟩, 'end';
⟨*consequence*⟩ ::= { ⟨*expectation block*⟩ }, { ⟨*java statement*⟩ };
⟨*expectation block*⟩ ::= ⟨*expectation list*⟩, { 'or', ⟨*expectation list*⟩ };
⟨*expectation list*⟩ ::= ⟨*expectation*⟩, { 'and', ⟨*expectation*⟩ };
⟨*expectation*⟩ ::= [⟨*id*⟩, ':'], 'expect', ['not' | ['one'], ⟨*id*⟩, ':'],
⟨*pattern*⟩, ⟨*follow-up*⟩;
⟨*follow-up*⟩ ::= ['on', 'fulfilment', ⟨*fulfilment block*⟩],
['on', 'violation', ⟨*violation block*⟩];
⟨*fulfilment block*⟩ ::= '{', { ⟨*repair*⟩ }, ⟨*consequence*⟩, '}';
⟨*violation block*⟩ ::= '{', { ⟨*repair*⟩ }, ⟨*consequence*⟩, '}';
⟨*repair*⟩ ::= 'repair', ⟨*id*⟩, ';';

Figure 4.1: The sub-grammar covering ECE-RULES, in EBNF form.

sequence of ⟨*expectation blocks*⟩ first, and then a (possibly empty) sequence of ⟨*java statements*⟩ as before. Passing through ⟨*expectation lists*⟩, each ⟨*expectation block*⟩ turns out to be a disjunction of conjunctions of ⟨*expectations*⟩. Each ⟨*expectation*⟩ – to which an ⟨*id*⟩ can be associated – is introduced by the 'expect' keyword and it requires a ⟨*pattern*⟩ and a ⟨*follow-up*⟩. The ⟨*pattern*⟩ is still a regular *Drools* pattern, while the ⟨*follow-up*⟩ comes down to a couple of optional blocks – namely the ⟨*fulfilment block*⟩ and the ⟨*violation block*⟩ – that are respectively introduced by the keywords 'on' 'fulfilment' and 'on' 'violation'. Notice that the definition of ⟨*expectation*⟩ can be further refined by using two optional keywords: 'not' and 'one'. In the former case, the keyword "*negates*" the following ⟨*pattern*⟩ with the same semantics of standard *Drools* patterns. In the latter, instead, the keyword 'one' is used to distinguish between two possible behaviours: sometimes, in effect, it is appropriate that any set of facts or events that match the premise of an ECE-RULE also triggers an expectation, while some other times it is required to activate only once and neglect any additional activation after the first. Each ⟨*fulfilment block*⟩ and ⟨*violation block*⟩ is encompassed in curly brackets and contains a possibly empty list of ⟨*repair*⟩ statements (that will be discussed in a moment) and a ⟨*consequence*⟩ as well. As the reader can guess, the reference to a ⟨*consequence*⟩ in this spot allows the definition of nested expectations. Notice that the nesting of expectations to express complex constructs is not mandatory as the default behaviour of the PRS to join objects from several patterns (see Appendix A on page 153) can be exploited as well. Also notice that the semantics of these blocks agrees with the common sense: the ⟨*fulfilment block*⟩ refers to the case in which an expectation is met and conversely the ⟨*violation block*⟩ to the opposite case. Finally, the ⟨*repair*⟩ production is a *Java*-like state-

```
rule "ECE-Rule Example"
when
  $m: Message($s: sender, $r: receiver, content == "HELO
    ")
then
  $e: expect one Message( sender == $r, receiver == $s,
                  content == "+ACK", this after[0, 10s]
    $m )
  on fulfilment {
    insert(new Message($s, $r, "MAIL"));
  }
  on violation {
    expect Message( sender == $r, receiver == $s,
              content == "+RDY", this after[0, 2m] $m )
      on fulfilment {
         repair $e;
         insert(new Message($s, $r, "MAIL"));
      }
      on violation {
         insert(new Message($s, $r, "STOP"));
      }
  }
end
```

Listing 4.1: An example of ECE-RULE with nested expectations mimicking the behaviour of a trivial mail server muffled by requests.

ment introduced by the keyword 'repair' and concluded by a ';'. It requires a mandatory ⟨*id*⟩ to identify the expectation that needs to be repaired. This construct is used to acknowledge the system that an expectation that was violated can be now considered as fulfilled. As it is reasonable to assume, the ⟨*repair*⟩ statements only affect violations: any attempt to repair a fulfilment simply has no effect. The reference to the ⟨*repair*⟩ production in the ⟨*fulfilment block*⟩ should not be suppressed as it allows to repair a (*nested* or *joined*) violated expectation from a fulfilled one.

What to expect from expectations

EXAMPLE The following Listing 4.1 presents an introductory representative of ECE-RULE which models the expected behaviour of a toy mail server. In this example, the exchange of messages between the server and the client is realised by introducing an appropriate `Message` into the WORKING MEMORY (WM) of the PRS. Each `Message` is declared as a *Drools Fusion* event so that it is automatically contextualised in time (see Appendix A.3 on page 164) and it has a `sender`, `receiver` and a `content`. This ECE-RULE can be interpreted as follows:

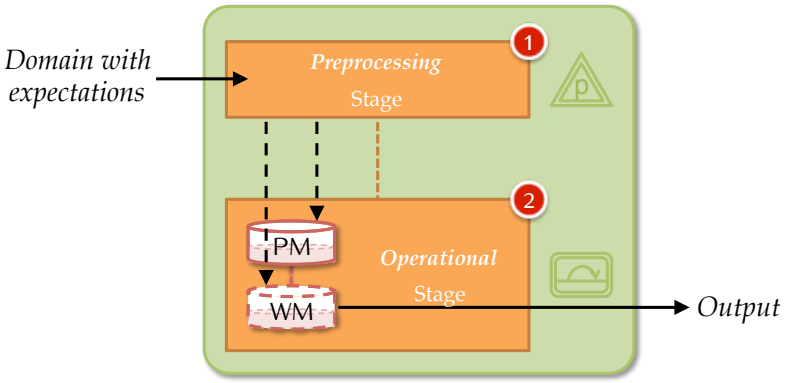


Figure 4.2: Outline of the system architecture that copes with ECE-RULES.

each time a client tries to handshake with the server (line 3), the same server (line 5) is expected to reply within ten seconds (line 6). Notice that several replies may be issued because of possible transmission problems, but only the first **one** actually triggers the following evaluation. If the server fulfils this requirement, the client proceeds with the interaction by requesting the list of new messages (line 8). If this expectation is disregarded possibly because the server is muffled by requests, a new nested expectation is introduced (line 11). This nested expectation requires the server to send a message to the client that is listening for communications within two minutes to notify that the peak of requests has passed and it is now ready to serve it (line 12). If this is the case, the former violated expectation is considered as compensated (line 14), the compound expectation is considered as a success and the client finally asks for the list of new emails (line 15). If this expectation is also not satisfied, the compound expectation is regarded as a failure and the client cancels its request (line 18).

4.3 META-MODEL AND SUPPORTING RULES

Architecture of the module

As in many other cases (some of which are presented in this dissertation: see, for example Chapter 2.3 on page 35 and Chapter 3.2.2 on page 82) the implementation of the system for ECE-RULE is organised in two layers as sketched in Figure 4.2. In this case, however, only the second layer is a proper PRS. The first layer in fact implements a preprocessing stage (marked by the number 1) which reads both standard *Drools* statements and rules augmented with expectations.

The preprocessing layer

This task is achieved by extending the default *Drools* parser with functions to recognise additional or overridden productions (as described in the previous Chapter 4.2 on page 94). The result of the parsing process is an ABSTRACT SYNTAX TREE (AST) with a regular and recursive structure that is derived from the default grammar plus the extension in Figure 4.1. Any standard piece of data is directly

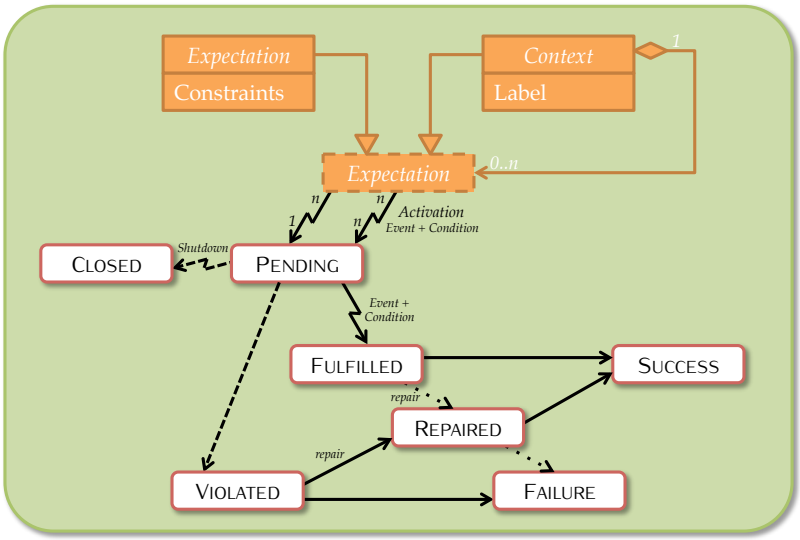


Figure 4.3: The meta-model for the expectations in ECE-RULES.

passed to the WM and PRODUCTION MEMORY (PM) of the underlying operational stage (that is identified in Figure 4.2 by the number 2) while any expression that is related to expectations is locally processed before being redirected to the following layer. By firing all the rules of this second layer, the domain with expectations is properly managed producing some output.

The operational layer: expectations' meta-model

As already mentioned, the main concept of this extension is the expectation. Expectations can be combined in disjunctions of conjunctions of expectations or in nested expectations. This ability to combine into more complex structures is quantitatively expressed in the top part of Figure 4.3 where it is suggested by the adoption of the *Composite* structural pattern [67]. This choice has only an explicative value since expectations are precisely maintained in AST structures. Notice however that for each kind of expectation that is specified by the domain an `Expectation` is introduced. This object contains fields to hold additional information or constraints about the expectation such as the `label` that is used to collect together the expectations referring to a single context. In addition to `Expectation` objects, there are several instances of `Closed`, `Failure`, `Fulfilled`, `Pending`, `Repaired`, `Success` and `Violated` that are used to decorate each expectation and keep track of how its state evolves over time. Notice that all these objects are defined as *Drools Fusion* events of null duration with the sole exception of the `Expectation` whose starting time is set to the time in which the triggering set of facts or events is individuated and the duration is determined by the time in which the expectation is satisfied or blatantly violated.

Enabling expectations

EXPECTATION RULES Notice in particular that as soon as a triggering set of facts or events of an expectation is found an open `Expectation`

decorated as `Pending` is asserted in the WM. Notice that expectations are converted into standard rules that are passed to the operational layer to be properly handled. Each (simple) expectation, in effect, is automatically translated into a set of standard rules composed. The first rule is precisely used to generated the `Expectation` and `Pending` instances: any time the *event* and the *condition* of the ECE-RULE are met and there are no `Expectation` objects for that activation, a properly initialised instance of both `Expectation` and `Pending` is added into WM. The second rule is optional and it is only available if the keyword **one** is present in the definition. This rule is used to defuse the expectation mechanism after the first success: it triggers on a `Pending` object and the first set of facts or events that match that the ⟨*pattern*⟩ that is specified by the expectation. As a consequence, it retracts the `Pending` object from the WM, so that the expectation is no more active for that activation. The process of activation of these two rule is depicted in Figure 4.3 on the previous page by the two little lightning symbols between the `Expectation` and the `Pending` objects. The idea that this symbol wants to convey is that an activation, being composed of an *event* and a *condition*, also contains temporal information.

The next couple of rules are mandatory as they detect fulfilments and violations of the given expectation. The rule for fulfilments has the same premise as above; its consequence is to assert a `Fulfilled` instance to further decorate the given activation of the `Expectation`. In Figure 4.3 a lightning symbol is used as well to indicate the evolution of the expectation from `Pending` to `Fulfilled` as it involves a ⟨*pattern*⟩ which is often resolved to an *event* and a *condition* too. The complementary rule for violations has a similar premise; in this, however, the pattern for the set of facts or events that fulfil the expectation is negated. The consequence is clearly the assertion of a `Violated` object for the given activation of the `Expectation`. Since the expectations typically contain temporal constraints, the meaning is that any expectation that is not satisfied within the given temporal deadline is considered as violated. Because of this reason, the state transition of the expectation from `Pending` to `Violated` is sketched as a dashed arrow.

The last couple of rules is optional: they are only required if the user has specified some *Java* statement into the ⟨*fulfilment block*⟩ or ⟨*violation block*⟩ of the expectation. Their premises are exactly like the premises of the two mandatory rules above and their consequences are simply the *Java* statements provided by the user, if any. Notice that the last four rules have higher salience than the second rule as otherwise they would be prevented from triggering for the anticipated disappearance of the `Pending` instance for the current activation. Also notice that nested expectations are managed accordingly by generating similar sets of rules in whose premises the ⟨*patterns*⟩ and the references to the `Pending` instances are accumulated.

Repairing expectations

REPAIR RULES Each time that a ⟨*repair*⟩ statement is encountered within ECE-RULES, a repair rule is added to the PM of the PRS that realises the second layer of the component (see Figure 4.2). The premise of these rules follows the same conventions of those introduced above: they contain a combination of conditions coming from ⟨*patterns*⟩ and `Pending` instances that are needed to identify the specific follow-up block in which they are contained. The consequence of these rules simply asserts a `Repaired` instance for the activation of the current `Expectation`. As already discussed in Chapter 4.2, an `Expectation` that is `Repaired` is fully equivalent to a `Fulfilled` expectation. We decided however to introduce a different state to distinguish the expectations that are repaired in a second time from those that are immediately satisfied.

Evaluating expectations

SUCCESS/FAILURE RULES With respect to the simpler case above, fulfilled or repaired expectations are obviously considered as successes while violated ones as failures. Assessing successes and failures for any expectation activation is performed by an apposite set of rules. For each activation, in fact, there is a rule whose premise looks for an `Expectation` and a matching `Fulfilled` or `Repaired` instance. As a consequence, that rule asserts a `Success` instance for the given `Expectation`. In a similar fashion, there is a rule whose premise identifies any `Expectation` that is matched by a `Violation`, whose consequence asserts instead a `Failure` instance for the given `Expectation`.

Each nesting level of nested expectations is still precisely treated in this way. In this context, a further complication is the possibility to issue ⟨*repair*⟩ statements if a block for another expectation. Notice however that once the all the repairs are applied, the logic defined by the rules above still applies. The case of disjunctions of conjunctions of expectations, unfortunately, is slightly more complicated. Once each single expectation of a compound statement is evaluated, their values may be combined together to provide the overall equivalent evaluation. The formulas to combine conjunctions and disjunctions of expectations are provided in Table 4.1 on the following page. These combination formulas are enforced by means of another set of rules. Two rules (that are recursively applied to cover formulas with several arguments) handle the conjunction of expectations. These rules are used to generate a `Success` or a `Failure` instance for the conjunction according to the state of two expectations: in particular, if only one of the inputs is a `Failure`, the conjunctions is a `Failure` as well, otherwise it is a `Success` (see Table 4.1a). Another couple of rules manages the complementary case of the disjunction of expectations: in this context, only two `Failure` instances as input lead to the assertion of a `Failure` instance for the conjunction. In all the other occurrences, a `Success` is generated (see Table 4.1b). Thanks to these rules, a `Fulfilled` or `Violated Expectation` may be evaluated as a `Success` or a `Failure`

Table 4.1: Combination formulas for the disjunctions of conjunctions of expectations.

(a) AND formula

and	S_1	R_1	F_1
S_2	S	S	F
R_2	S	S	F
F_2	F	F	F

(b) OR formula

or	S_1	R_1	F_1
S_2	S	S	S
R_2	S	S	S
F_2	S	S	F

. These state transitions are reported in Figure 4.3 on page 99: the more typical of them are sketches as solid arrows, other as dotted arrows. This typographical convention is used to distinguish between the most common transitions and the more uncommon ones. Notice however that this does not mean that those transitions are prohibited as they may happen on cross-referencing expectations.

Keeping the expectations consistent

WORLD CLOSURE RULES Expectations are typically managed by considering them closed towards the past, but open towards the future. Therefore some issues may arise when dealing with expectations that are still simply pending. Expectations that are still open, in effect, are not reported as failures by the monitoring framework when the system is suddenly halted. The procedure that identifies all the pending expectations and discloses them as unresolved with respect to the open time horizon is called "*closure*". Again, it is implemented by a set of rules that are fed into the PM of the operational layer's PRS. The deliberate halting of the monitoring framework is preceded by the assertion of a special `Shutdown` event into the WM. This object matches with the first part of the premises of the rules that identify all the open exceptions. A rule flags the `Expectation` as `Violated` if it contains a positive ⟨*pattern*⟩ (the expected behaviour did not manifest) and as `Fulfilled` if it is negated (no undesired behaviour has occurred). In addition to the assertion of these instances, a `Closed` object is generated as well to set apart closed expectations from the rest. This transition is represented as a dashed lightning in Figure 4.3 on page 99 to underline the special context in which it takes place. Notice that the set of rules for assessing the `Success` and `Failure` condition for each `Expectation` are applied before definitively halting the framework. The Listing 4.2 on the next page depicts the operational semantics of these rules.

Improving the implementation

OPTIMISATIONS The current state of expectations is also stored in a field of any `Expectation` instance. A set of ghost rules that mimics the behaviour of the rules introduced in the previous paragraphs and addresses this field instead of the instances of the state objects

```
rule "Closure - positive expectations"
when
  Shutdown()
  $e: Expectation(
        status == Status.PENDING,
        negated == false )
then
  $e.setStatus(Status.FULFILLED);
  $e.setClosed(true);
end

rule "Closure - negated expectations"
when
  Shutdown()
  $e: Expectation(
        status == Status.PENDING,
        negated == true )
then
  $e.setStatus(Status.VIOLATED);
  $e.setClosed(true);
end

query "Closed expectations" ()
  Expectation( closed == true )
end
```

Listing 4.2: Operational semantics of the rules for the world closure.

is provided. This allows us to insert a special object `Lite` which prevents the creation of any instance of the state objects. This second set of ghost rules can be disabled as well by asserting an instance of a special object `Full`. Consider that, it is not obviously possible to disable both sets of rules at the same time without tampering with the correctness of the framework. Therefore the assertion of an instance of one of those special objects causes the retraction of any instance of the opposite type that is currently included into the WM. This solution is motivated by the fact that preventing the generation of all the instances of these state objects possibly spares several computational resources. The adoption of the *full* mode is, however, strongly recommended as the reification of the state of any `Expectation` instance allows to keep track of the history of a process and extract statistics about it. The amount of violated activations of an expectation with respect to the amount of informations can highlight some critical steps and suggest a process of Knowledge Base Revision (KBR) to detect frauds and operational errors or promote exceptions to full-fledged norms (see Chapter 1.1 on page 4). In those domains in which sparing

resources is mandatory, it is possible to introduces adhoc rules that purge outdated objects from the WM (see Appendix A.3 on page 164).

4.4 GLOBAL CONFORMANCE

Beyond the strict evaluation of expectations

In the previous section we have introduced and detailed the notions of *fulfilment* and *violation* with respect to the single expectation, as well as the concepts of *success* and *failure* that encompass more complicated aggregation of expectations.

They have introduced the concept of both local conformance and compound conformance, but the notion of global conformance of a whole system should be considered as well. Such kind of conformance could be simply defined as the logical composition of individual conformance values, but we rather adopt a more sophisticated model in which users can introduce other criteria into the evaluation process. Metrics that are typically addressed as such criteria include – but are not limited to – semantic, statistic, gradual or imprecise reasoning, etc. Therefore, this evaluation process can virtually use any desired powerful matching function – be it regular expressions, fuzzy pattern matching or PROLOG unification, for example – it just need a flexible, robust mechanism to combine these contributions to a unique global score.

Considering other contributions

This vision guided the development of the main contributions discussed in this dissertation. All the components, in fact, are designed to be modular: any of the addressed reasoning techniques is handled by a self-contained set of rules and declarations; these cores can coexists within the WM and PM of a same PRSs and produce results in parallel. Notice moreover that any additional self-contained module that can share a PRS memories is a valid candidate as well. The only discriminating requisite is that any specific evaluator to be implemented as a CUSTOM OPERATOR (CO) that returns a degree as a result of its computation. The *Drools Chance* system, in effect, is already able to combine these degrees with a great level of flexibility (see Appendix A.4 on page 171).

Fuzzy deadlines

The fuzzy evaluation of time aspects is a feature that we have identified as extremely important and that we have not addressed yet. In this case it is not sufficient to put aside different components to accomplish it. In particular, in effect, *Drools Fusion* supports all the Allen's (crisp) temporal operators [7, 9, 10] and *Drools Chance* fuzzy expressiveness, but together they do not provide fuzzy temporal evaluation out of the box. In order to achieve this feature, we have decided to stick with the following approach: defining new COs for any specific fuzzy temporal need. At the moment, in fact, we have not considered any specific time formalism even though we are aware that many are available in literature. As an example, consider some recent

work within the MAS research community stressing on the importance of temporal aspects like deadlines [177]. The decision to implement adhoc evaluators is due to the fact that they greatly vary from case to case. Socio-technical systems for the CCG, for example, may need very relaxed evaluation of temporal constraints as several low-priority human tasks are involved. Technical systems, such as Web services for example, many require a more rigorous assessment of temporal aspects. In some cases like the handling of QOS and SERVICE LEVEL AGREEMENT (SLA), a given margin of imprecision still applies while in other contexts more oriented towards strict real-time evaluation not.

Considering other options

Since a rule-based semantic module is already available (see Chapter 3 on page 77), a more organic approach based on ontologies could be developed. We could define a top- or upper-level ontology to deal with any kind of temporal requirements so as to provide a reference to the knowledge modeller to precisely describe the specific temporal needs of any given domain. The semantic module could interpret these statements and automatically create the equivalent COs to be used within rules. Approaches like this are probably the only ones that can manage this great variability.

However, we wonder whether it is advisable to follow this idea. On one hand, in fact, many leading researchers emphasise the benefits deriving from a similar approach such as the reusability of the same established knowledge in several ontologies, the ability to more easily perform deep reasoning over the knowledge in a broad sense or even the advantage of providing a universal shared "dictionary" with whom to express the knowledge. On the other hand, other researchers that, instead, are more concerned about the practical feasibility of these approaches, argue that the definition of an upper-level ontology and the acquisition of the competences to properly use it are too costly for any average task.

Many players and vendors in the real world, in effect, prefer to internally develop their ontologies from scratch as it drastically reduces the time-to-market of their applications rather than adopting something that could lead to consistency problems with updates or even potentially promotes the sharing of their own competitive advantages. It is not our intention to fuel this debate as both positions have quirks and perks. However, given our practical experience in the field of eTourism where ontologies play a central role and the finding that the monitoring tasks in which we are interested involve practical limitations due to their necessity of reactively responding to the changes, we have decided to stick with the simpler, more practical and effective approach.

The true power of expectations with global conformance

EXAMPLE In the Listing 4.3 on the following page, we take again the same domain introduced in the Example 4.2 on page 97. This time, however, we also consider different kinds of custom evaluators

```
1  rule "Advanced ECE-Rule Example"
2  filter 0.8
3  when
4    $s: Service( this isA MailServer.class )
5    $r: Request( service == $s, $t: time )
6    ?holdsAt( MonitorOn, $t; )
7  then
8    @Imperfect(kind=="userOp")
9    $e: expect one Message( service == $s,
10                    $d: delay ~inTime $r, this ~isA Reply.
       class )
11   on fulfilment {
12     $s.setQuality(Quality.OPTIMAL);
13   }
14   on violation {
15     @Imperfect(family==MvlFamilies.GODEL)
16     $en: expect Service( this != $s, this after[0, 2m]
       $r )
17     on fulfilment {
18       repair $e;
19       $s.setQuality(Quality.GOOD);
20     }
21     on violation {
22       $s.setQuality(chance.getDegree());
23       insert(new Fine((1.0 - chance.getDegree()) * 100))
       ;
24     }
25   }
26 end
```

Listing 4.3: A more advanced example of ECE-RULE on the same subject of Example 4.1 on page 97 where various contributions of different nature are considered to asses a global conformance score.

to compute a score of global conformance for the specific execution that is being monitored. In line 3, for instance, the use the semantic **isA** CO to identify the services that are defined as mail servers (see Chapter 3 on page 77). In line 6, instead, we exploit EVENT CALCULUS (EC) to verify the condition that the monitoring framework is operative (see Chapter 2 on page 21). Starting from line 9, the example starts to consider expectations. The annotation on line 8 tells the engine to combine the scores coming from the evaluation of the main expectation with the contributions of any possible nested expectation by using the a user defined strategy (see Appendix A.4 on page 171). In particular, the expectation on line 9 relies on the fuzzy semantic

operator **~isA** to determine whether a message looks like a `Reply` (line 10, see Chapter 3 on page 77) and on the fuzzy temporal CO **~inTime** to determine whether the replay was received reasonably in time (line 10, see Chapter 4.4 on page 104). In case of fulfilment, we set the value equivalent to the fuzzy linguistic label `OPTIMAL` of the linguistic partition `Quality` for the current server (see Appendix A.4 on page 171).

In case of violation, we have a plain nested expectation (line 16). In this case, the annotation on line 15 tells that the contributions for determining the global conformance coming from this violation branch have to be combined according to the Gödel semantics (see Appendix A.4 on page 171). Fulfilling this nested expectation leads to tagging the server with the fuzzy linguistic label `GOOD` (line 19), as above. In case of violation, instead, the quality of the service is directly set to the current degree of the global conformance score (line 22, see Appendix A.4 on page 171). That value, however, is also used to compute the amount of the fine that the mail service provider has to pay for failing to honour the agreed SLA on the QOS (line 23) that in this example is issued by inserting a `Fine` instance into the WM. Finally, notice that the ECE-RULE only focuses on serious violation of the protocol as the rule attribute **filter** on line 2 ignores any violation whose global conformance score is lower than 0.8. Also notice that this example does not consider any statistical evaluator to asses the global conformance, but it could be used as well (see Appendix B on page 183).

4.5 SUMMARY

This Chapter includes the implementation of the mechanism that we propose to handle the deviations in complex processes (see Chapter 1 on page 1). This contribution is currently incubated by *JBoss* as a satellite project and features from it are progressively going to be included into the mainstream component in the upcoming releases. After a brief introduction on the problem where the most distinctive related works are discussed, we have introduced our idea of EVENT--CONDITION-EXPECTATION RULES (ECE-RULES) as an evolution of ECA-RULES to assess the conformance of processes.

Then, we have discussed the changes that need to be introduced to the syntax of *Drools* to make it understand expectations while introducing the concepts that distinguish them. An extensive example further clarifies this concept. In the following part of the Chapter we have sketched the diagram that represent the system architecture. We have also defined the meta-model behind ECE-RULES depicting all the possible states and transitions of the expectations. We have also described in details the set of static and dynamically generated rules

that are needed to assist expectations during their whole lifetime and a few optimisations as well.

In the last part of the Chapter we have introduced the concept of *Global Conformance* with which we believe it is possible to bring the conformance checking to a new level. This approach, in fact, allows us to take advantage of any additional specific evaluator to better asses the conformance of a process and to return a unique score that considers all these contributions. By doing so, we have also addressed a practical and a theoretical way to interpret temporal deadlines in a more gradual way, according to the requirements of the domain. The Chapter is closed by an advanced take of the same example introduced before in which we show how all the scientific contributions of this dissertation (see Part i on page 21) can be organically used to estimate the complex processes conformance. A few practical applications of these contributions are reported in the following Part.

Part II

USE CASES AND APPLICATIONS

5 PRACTICAL APPLICATIONS

«In theory, there is no difference between theory and practice. But, in practice, there is.»

— Johannes Lambertus Adriana van de Snepscheut
Dutch Computer Scientist and Educator, 1953-1994

This Chapter contains a few examples of practical applications in which the theoretical contributions discussed in the previous Part are applied. In a few contexts, a specific use case only addresses one of the suggested contributions, but generally they include all. This Chapter ultimately aims not only to show that the theoretical aspects discussed so far are really applicable, but also to underline effectively they improve the solutions in a few domains. In particular we will discuss an example in Computerised Clinical Guidelines (CCG) where we use the *Kinect*, a motion sensing device by Microsoft [1], and some Machine Learning (ML) techniques to monitor senior citizens to identify dangerous health conditions and a more standard approach to help providers to identify diseases and properly assist patients. Another use case is eTourism, where we show how the theoretical contributions of this thesis can be exploited to implement a precise recommendation system of touristic offers. The central part of the Chapter is devoted to the Web Service (WS), Service-Oriented Architecture (SOA) and Cloud Computing (CC): a first example, in fact, stems from the implementation of a formal and efficient monitor for services in a SOA environment that is further extended to manage the Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) layers as well as a second example in where we show how our contributions can solve complex orchestration problems. Last but not least, we show how to efficiently control a Waste-Water Treatment Plant (WWTP) with our technologies.

5.1 REAL-TIME POSE PREDICTION AND MONITORING

The application that is being described in this section was carried out as part of the *Depict* project of the former Department of Electronics,

1 http://www.microsoft.com/en-us/kinectforwindows/

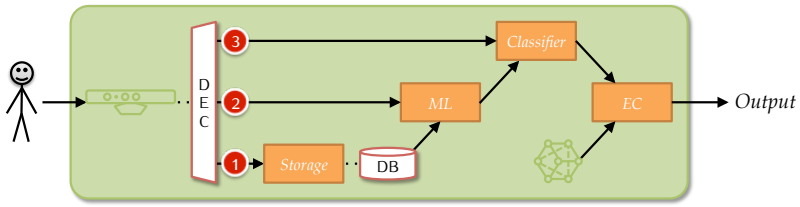


Figure 5.1: Operating Diagram of the Pose Prediction and Monitoring framework.

Engineering and Systems (DEIS) and the EU-FP7 *Farseeing* project by coordinating the work of some students.

Use case

These projects focus on the implementation of a Decision Support System (DSS) that remotely detects possible falls of elderly people and draw the attention of the medical staff and possibly the law enforcement personnel to assist the patient. The software is supposed to recognise any scene in which the senior citizens being monitored fall due to some illness or disease and, if the situation persists, to raise an alarm. In this kind of applications, both *recall* and *precision* are important.

The recall, for instance, is the fraction of proper falls that are recognised: it is mandatory not to miss any *false negative* [2] because early intervention significantly increases the chances of life saving and recovery. The precision, instead, is the fraction of reported falls that are actually falls: although a low precision is acceptable, it is advisable to minimise the number of *false positives* [3] in order to maximise the available resources. Each false alarm, in fact, usually involves sending a rescue team which is both a waste of money (moving people) and an inadequate allocation of resources (the personnel in a false rescue mission is not available in the event of a further real emergency).

Involved technologies

Our approach adopts several technologies, both hardware and software. From the hardware standpoint, it revolves around the *Kinect*, a motion sensing input device by *Microsoft* formerly introduced as an accessory for the *XBox 360* video game console and later adopted in several Computer Vision (CV) research projects. From the point of view of the software bundle, our approach mainly makes use of *Weka* [4], a collection of ML algorithms for Data Mining (DM) tasks by Weikato University, New Zealand and a tool providing an implementation of Predictive Model Markup Language (PMML) [5] based on *Drools Chance* [169]. These tools process the information captured by the *Kinect* to provide the input for the original components that we described in the Part i on page 21.

General architecture

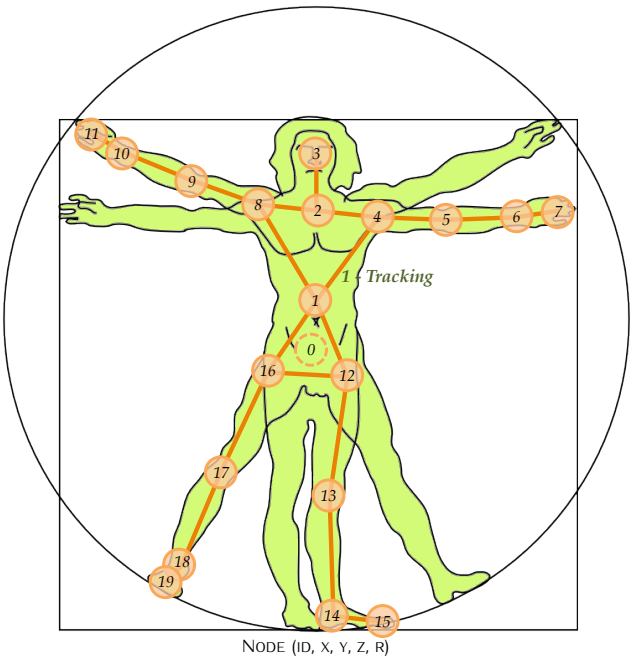
Figure 5.1 shows the general operating principle of our prototype.

2 In this context, a false negative is a falls that is not opportunely identified.

3 A false positive is any action like a suddenly sitting or standing still while sleeping, that is interpreted as a fall.

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

5 https://github.com/droolsjbpm/drools-chance/tree/master/drools-pmml



0. Hip centre
1. Spine
2. Shoulder centre
3. Head
4. Left shoulder
5. Left elbow
6. Left wrist
7. Left hand
8. Right shoulder
9. Right elbow
10. Right wrist
11. Right hand
12. Left Hip
13. Left knee
14. Left ankle
15. Left foot
16. Right Hip
17. Right knee
18. Right ankle
19. Right foot

Figure 5.2: Skeletal features in a frame as in a typical pose acquisition session.

First of all, we needed to set up a software stack to properly interact with the *Kinect*. There are several libraries that accomplish this task that are similar for characteristics, capabilities and ease of use: our choice is *OpenNI* (or *Open Natural Interaction*), an industry-led library bearing the same name of the non-profit organisation that develops it, focusing on "*certifying and improving interoperability of natural user interface and organic user interface for natural interaction devices, applications that use those devices and middleware that facilitates access and use of such devices*" [6]. This software is often complemented by the *Processing* library [7], an open source programming language and environment for creating images, animation, and interactions and the *NiTE* middleware [8] that understands the hand gestures and fully body movements and translates them into inputs suitable for software applications. As the reader can see, the data coming from the *Kinect* is redirected towards three routes (identified by the numbers 1, 2 and 3) which may be active in different times, so first we have wrapped an adapter DEC around the software stack to decouple the source of information from the parts that consume it.

Operating principle: first task

The first path stemming from the DEC adapter loads the information in a simple *storage* service which takes the skeletal poses captured by the *Kinect* and interpreted by the software stack at a rate of 30 frames per second. Each pose consists of twenty quintuplets in which the

6 http://www.openni.org
7 http://processing.org
8 http://www.primesense.com/nite

first integer value is the *identifier* of the joint, the following three real values are the coordinates x, y and z of the joint and the last real value is an estimation of the reliability of the measured joint's position (see Figure 5.2 on the previous page).

The joints identified on the user's skeleton are the following: *hip center* (or centre of mass, identified with the key-number 0), *spine* (1), *shoulder center* (or centre of shoulders, 2), *head* (3), *left shoulder* (4), *left elbow* (5), *left wrist* (6), *left hand* (7), *right shoulder* (8), *right elbow* (9), *right wrist* (10), *right hand* (11), left hip (12), *left knee* (13), *left ankle* (14), *left foot* (15), *left hip* (16), *left knee* (17), *left ankle* (18) and *left foot* (19).

The last value gives a measure of the reliability of the pose just acquired: a value of 0.0 is returned when the tracking fails and the coordinates are useless, 1.0 is returned when the tracking produces valid data and, finally, 0.5 is returned when the tracking enables the so-called *skeleton heuristics* to fix the coordinates of the few joints that were not properly acquired. In order to learn appropriate poses, we have discarded all the frames including joints detected with confidence lower than 1.0 and we have stored the remaining ones.

Operating principle: second task

The second task performed by the framework is ML: this step takes as input the poses that we have previously stored or those directly captured by the hardware and tries to "learn" the features that set each of them apart from the others. In particular, we have decided to learn the following poses (and sub-poses in parenthesis): *sitting*, *standing* (*walking*, *hands-up*), *crouching*, *laying* or *fallen*.

In order to make this process more robust, we have used a set of poses related to four different individual for body mass and height. The we have configured *WEKA* to perform supervised learning by considering several predictive models. In particular, we have adopted four learning algorithms pertaining to three different categories: Multi-Layer Perceptron (MLP), Decision Tree (DT) – J48, Logistic Model Tree (LMT) – and Support Vector Machine (SVM). A MLP is a *fully-connected feed-forward* Neural Network (NN) with at least three layers (*input*, *hidden*, *output*). This NN uses Backpropagation (BP) to reduce the error between the output and the expected value for each input. A DT is a tree-like diagram where nodes represents variables, arcs possible assignments and leafs predicted values for the goal variable by each path. The DT is built from the training set – the poses – and the validation set – the categories of poses. We adopted the J48 algorithm and the LMT. The SVMs, instead, represent the instances of the training set as points in a multi-dimensional space and tries to build hyper-plans to separate them into categories. More details on the learning process and the configuration of each single algorithms is available in [54], it suffice to say that the best model is passed to the classifying process as a PMML file.

Operating principle: third task

The third task performed by the framework is the real-time classification of the poses captured by the *Kinect*. To this aim, we use

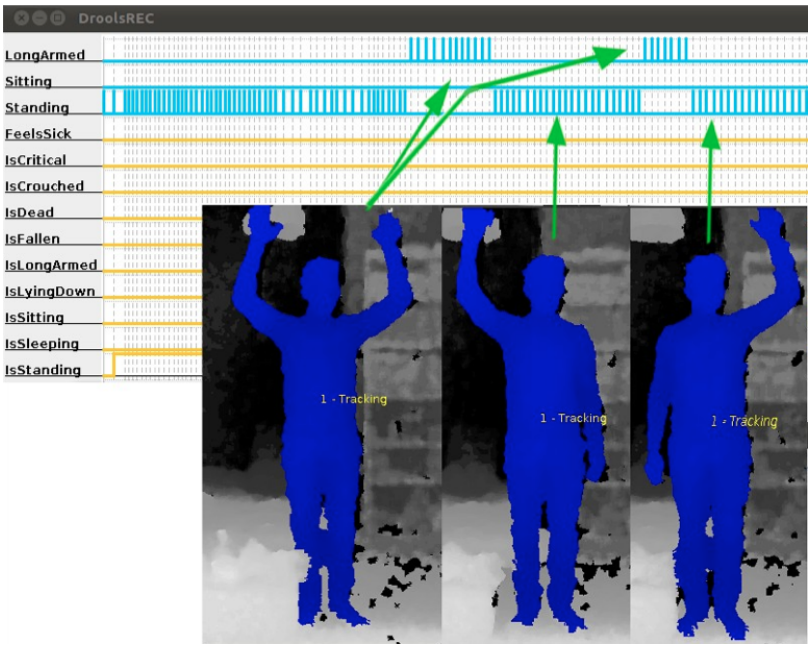


Figure 5.3: Event notification of the predicted poses: raising both hands up from standing and lowering one hand down at a time.

the *Drools Chance* extension for PMML which basically takes the best model that was learnt during the previous step to build a NN out of reactive rules which transforms each pose into an event notifications.

These events have a timestamp and represent the possible poses that we are considering: *sitting*, *standing* (*walking*, *hands-up*), *crouching*, *laying* or *fallen*. They are passed to the EVENT CALCULUS (EC) machinery that is already initialised with the domain model. This model introduces as many main fluents as the number of possible events: *is sitting*, *is standing* (*is walking*, *has hands-up*), *is crouching*, *is laying* and *is fallen*. So anytime the NN predicts a pose and notifies it as an event, the EC machinery sets its corresponding fluent to TRUE, making *false* all the others (see Figure 5.3). We have also defined a few derived fluents such as *is fine*, *feels sick*, *is critical* and *is dying* whose value is set also taking advantage of expectations.

Roughly speaking, as long as the subject continues to move (meaning passing from one pose to another or performing a continuous action like walking or keeping the hands raised up)we consider him in good health. Since we expect him to do something in a given timespan to testimony his health, if he does not we consider him sick. A prerecorded message is played to ask the subject to raise his hands up if he feels fine: if he does so, we consider him fine again otherwise he is considered critical and some assistance is sent to his place. Once arrived on site, the paramedical staff may decide if he is really dying and take action to stabilise him or determine the false positive and reset the state of the subject accordingly.

A video demonstration of this application is available at the following address: http://youtu.be/mVyCEufGq4E.

Conclusions

With these mutually exclusive fluents, we want to give a measure of the state of health of the subject being monitored so the whole system can be seen as a way to compute an elaborate fitness function for him. Of course, this approach is just a proof-of-concept that still needs to be evaluated by the medical partners of the projects. The procedure, in fact, requires some more fine tuning to minimise further the number of unneeded interventions. The height of the mass centre of the subject from the ground, in fact, could be used to determine if the subject has fallen or is sleeping. Such information may also be used to extend or reduce the time interval during which an action is required to consider the subject fine. In addition, we plan to introduce an affine transformation to move the origin of the 3D space from the *Kinect* to the centre of mass of the subject as in other domains it seemed to improve the accuracy of the learning and a Kalman filter to reduce the effects of the noise from the acquired data.

5.2 COMPUTERISED PATIENT MONITORING AND EVALUATION OF CLINICAL HISTORY

This application was realised in conjunction with some members of the HEALTH SCIENCES AND TECHNOLOGIES – INTERDEPARTMENTAL CENTRE FOR INDUSTRIAL RESEARCH (HST-ICIR) of the University of Bologna, the *Depict* project of the former DEPARTMENT OF ELECTRONICS, ENGINEERING AND SYSTEMS (DEIS) and the KNOWLEDGE MANAGEMENT RESEARCH (KMR) team at San Diego NAVAL HEALTH RESEARCH CENTER working on the U.S. Navy *Knowledge Management Repository II Project*.

Use case

This use case was proposed by KMR group which aims to develop functional ontologies and semantics that are typically required to define operational constraints to the execution of a rule. Notice that the opinions reported below do not necessarily state or reflect those of their respective employers, the United States Navy, the Department of Defence, or the United States Government, and shall not be used for advertising or product endorsement purposes. The *operational constraints* are meta-level rules aiming to supervise the domain by establishing expectations for the clinical context for which a given rule was designed for. In other words, they help to ensure that any resulting event or behaviour is appropriate for the setting. They can be used, for instance, to adapt the recommendations for a blood transfusion in a trauma case when the patient belief is that of a Jehovah's Witness.

Some desiderata

Keeping the logic of a discreet medical decision separated from that used by operational rules to manage generic clinical context has considerable implications. The rules that provide support to decisions should be authored with a focused clinical perspective. The final re-

sult, however, depends on other orthogonal perspectives that the underlying knowledge management system must precisely orchestrate. It follows that the execution of the rules does not necessarily require them to be activated all together at once or not activated at all. This evaluations of constraints rather resembles a cascading effect whose effect, once aggregated, is to ensure that the overall system behaviour is nuanced and individualised. This separation of concerns helps to clarify what are the best clinical practices on the basis of the available evidences and what are the non-medical restrictions on the care delivery that are imposed by the context and the individual patient.

A specific scenario

The following scenario was considered as use case. After returning from a yearlong deployment, a United States Marine is seen by his physician. The doctor's Clinical Decision Support System (CDSS) collects all relevant information about him and feeds it into a Post Traumatic Stress Disorder (PTSD) predictive model [169] that estimates that he has a 35% risk of developing PTSD within the next three years. Recognising, however, that several important historical facts about his past medical history are missing, the patient is asked to take an online survey at home. When he forgets to complete the survey, the system automatically sends a reminder SMS text prompting the Marine to complete the requested task. When he does so, the system then automatically recalculates the risk score. This time the risk is 80% and the confidence acceptably narrow, so an alert is instantly generated.

Principle of operations

In the wake of what we have presented in Chapter 4 on page 91 about Event-Condition-Expectation rules (ECE-rules), the various contributions that we have presented in the Part i on page 21 can be used to model the above use case. The Listing 5.1 on the next page contains an abstract solution to such a problem. The first rule (lines 1-22) manages the results of the PTSD predictive model evaluation represented by a `HasRisk` object (line 5) which also binds together a patient `$pat` (line 3) with a provider `$prov` (line 4). Notice that the evaluation of a predictive model within a rule engine is out of the scope of this work, but the interested reader can find the details in [169]. We expect the confidence of the results to be above a given threshold `C_THOLD` if the predictive model works properly (line 7). So, when the prediction is reliable enough (line 8), we expect the risk factor for PTSD of the `patient` to be below a given value `R_THOLD`) line 10). If this nested expectation is fulfilled, we log the patient as safe (line 12). Otherwise we have a violation and a specific procedure (that was undisclosed for privacy reasons) is applied (line 15). Notice, however, that a prediction that is not accurate enough means that more information is needed (line 18). Such information is requested by inserting a `Message` object into the Working Memory (WM) of the Production Rule System (PRS) whose effect is to trigger another rule which effectively sends a SMS from the provider `$prov` to the patient `$pat` to request the completion of a questionnaire (line 20).

```
rule "Risk factor evaluation"
when
  $pat: Patient( ... )
  $prov: Provider( ... )
  $risk: HasRisk( $pat, $disease, $factor, $conf; )
then
  expect HasRisk( this == $risk, confidence > C_THOLD )
  on fulfilment { // prediction is reliable
    expect HasRisk( this == $risk, factor < R_THOLD )
    on fulfilment {
      log($pat + " safe");
    }
    on violation { // manage high risk patient
    }
  }
  on violation { // request info from patient
    insert(new Message($prov, $pat, "quest"));
  }
end

rule "Fill Questionnaire Request Protocol"
when
  $m: Message( $prov, $pat, "quest"; )
  not Answers( $pat, $prov, "quest"; )
then
  $e: expect Message( $pat, $prov, $answ;
                this ~inTime[0,T] $m )
  on fulfilment {
    insert(new Answers($pat, $prov, "quest", $answ));
  }
  on violation { // TT > T, give patient more time
    expect Message( $pat, $prov, $answ;
              this ~inTime[0,TT] $m )
    on fulfilment {
      repair $e;
      insert(new Answers($pat, $prov, "quest", $answ));
    }
    on violation {
      alert($prov);
      insert(new SMS($prov, $pat, "quest"));
    }
  }
end
```

Listing 5.1: Abstract simplified example of ECE-RULES to asses the risk factor of a Marine to suffer from the POST TRAUMATIC STRESS DISEASE.

The second rule (lines 24-45), in fact, triggers on the assertion of such `Message` instances (line 26), provided that the additional condition that answers for the questionnaire are not yet available (line 27) is verified. Therefore we expect the patient `$pat` to send his `$answ` back to the provider `$prov` (line 29) roughly within a time `T` (line 30). In case of fulfilment, we assert the Marine's answers into WM (line 32) so that the ECE-RULE that we are currently considering does not trigger again anymore. Notice that as soon as the above answers become available, other ECE-RULES will trigger to further assist the patient `$pat` . In case of violation, the protocol is expected to give more time `TT` (line 37) to the patient `$pat` to provide his answers (line 36). If the Marine does so, his answers are asserted into WM with the same outcomes as above (line 40) and the violation of the root-level expectation `$e` is considered as repaired (line 39). If he does not, the Marine's provider `$prov` is alerted (line 43) and the patient is pressed with another SMS to complete the questionnaire (line 44). Notice that the same policies could have been written using standard rules, but the proposed syntax makes the definition more compact and, most of all, ensures that the results of the constraint checks are recorded formally. The additional time that the Marine may take to fill the questionnaire, in fact, could be considered as a further symptom of PTSD to be taken into account. Finally, notice that this second rule takes advantage of some of the additional features that we have described in Chapter 4.4 on page 104 to better asses the conformance of the human task with the expected behaviour.

5.3 ADVANCED RECOMMENDATION SYSTEMS IN ETOURISM

This application was developed within the Italian MIUR PRIN 2007 PROJECT No. 20077WWCR8 on "*Correlation Forms Among Italian Style, Tourism Flows and Consumption Trends of Made in Italy*" as a use case to show the potential of the tool for hybrid reasoning that we have presented in Chapter 3 on page 77.

Use case

The use case consists in a DSS for the tourism domain where ontologies are used to formally describe customers' profile and tour operators' offers and rules are exploited to evaluate the consistency of such information and possibly to suggest the proper offers to the right client. In this applicative context, in fact, the turnover is very large and the competitors are very aggressive, so a targeted advertising can result in a competitive advantage and big gains. The overall architecture of the DSS is sketched in Figure 5.4 on the following page: the hybrid reasoner is the core of the framework, suitably supported by a persistence layer. The hybrid reasoner is initially fed with fuzzy, semantic and rule-based knowledge about the domain. These

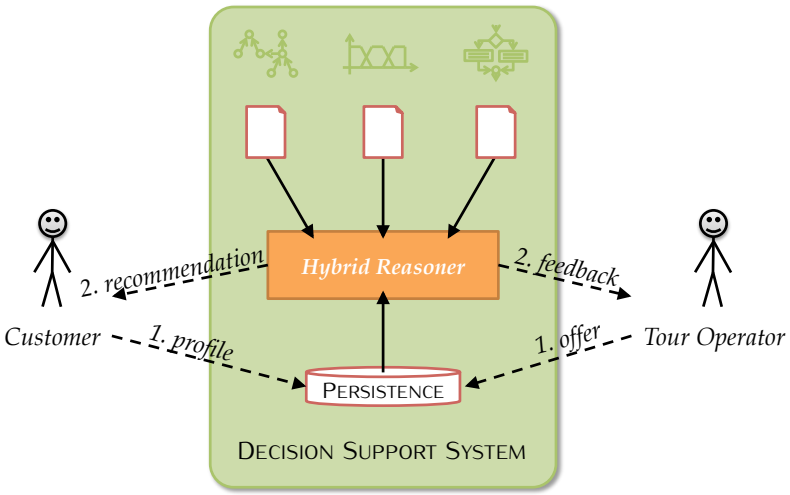


Figure 5.4: Architecture of a DECISION SUPPORT SYSTEM for eTourism combining, fuzzy, semantic and rule-based information.

knowledge bases are built upon studies on the domain, other similar ontologies that are publicly available and on the insight of domain experts. The principle of operation of the tool is the following: initially tour operators and customers insert respectively touristic offers and personal information into the DSS, these pieces of information are processed in order to classify the instances in memory into meaningful categories and, finally, the offers that fall into each category are recommended to the clients of the relevant groups and a feedback for the tour operator may also be generated.

A typical scenario

EXAMPLE A first preliminary example is the following. Suppose that the domain expert decides that "*cultural package tours should be preferably offered to senior customers*" (see the resulting Listing 5.2 on the next page). Here, the concepts of "*senior*" and "*cultural*" are not yet properly defined in an ontological sense, but it uses some object fields as ontological relationships and evaluates them in a possibly fuzzy manner. A "*cultural*" offer is an offer that is located at least in a place (line 6) that is somehow related to *art* or *history* (line 5). Notice that the correlation with history is precisely evaluated that requires an explicit or implicit definition of a place as such, while the correlation with art is evaluated more loosely leaving the engine to determine this degree. Similarly, a "*senior*" customer is a customer whose age seems old according to some fuzzy partition on age provided by the domain expert (line 7). Therefore the following rule triggers any time a couple `(customer, offer)` that matches the above conditions is found, with the result that a mail is sent to the customer to advertise that offer [9] (line 9). Notice that a rather high filtering value is used

9 In this example, the task of sending an email is carried out by asserting a `Mail` object into WM.

```
1  rule "Cultural offers to senior customers"
2  filter 0.85
3  when
4    $p: Place(
5          this isA HistoryPlace or this ~isA ArtPlace )
6    $o: Offer( hasPlace == $p )
7    $c: Customer( age ~seems Age.OLD )
8  then
9    insert(new Mail($c.getMail(), "Offer!", $o));
10 end
```

Listing 5.2: Example of hybrid rule in eTourism.

to ensure that only the really relevant offers are sent to interested customers (line 2).

Some desiderata

As the reader can see, this kind of hybrid rules can be very expressive, allowing the knowledge engineers to express particularly refined reasonings. The true potential of the tool, however, allows reasonings that are much more interesting than this. As we said, in this domain the tour operators struggle to let the customers have access to the most suitable offers. In practice, due to the high number of customers, offers, technologies and competing tour operators that strive to offer the same service, there is a gap between customers and offers which prevents the former to reach the latter. As the reader can see in Figure 5.5 on the following page, such gap is bridged over thanks to two distinct levels of reasoning. The first involves vertical reasonings that allows to move from the specific instance to its general category and vice versa. The subsumption mechanism provided by the DESCRIPTION LOGIC (DL) part of the hybrid reasoner covers this type of reasoning. On one side of the Figure, in fact, it is possible to pass from `Customer` instances to a `Stereotype` classes and back, while in the other from `Offer` instances to `Typology` classes and back. The other level of reasoning is horizontal and binds together `Stereotype` and `Typology` classes. The number of concepts at this higher level of abstraction is much smaller, so it is easier for a domain expert to express the correct combinations. This task is performed by a matching function that implements such combinations. Notice that both levels of reasoning are completely decoupled but they outline an alternative path that allows the customers to bridge over the gap and reach the much desired customised offers.

Improving the scenario

EXAMPLE A first, intermediate passage to make the previous example as general as just outlined requires the definition of all the *stereotypes* and *typologies* into ontological terms. When these definitions are

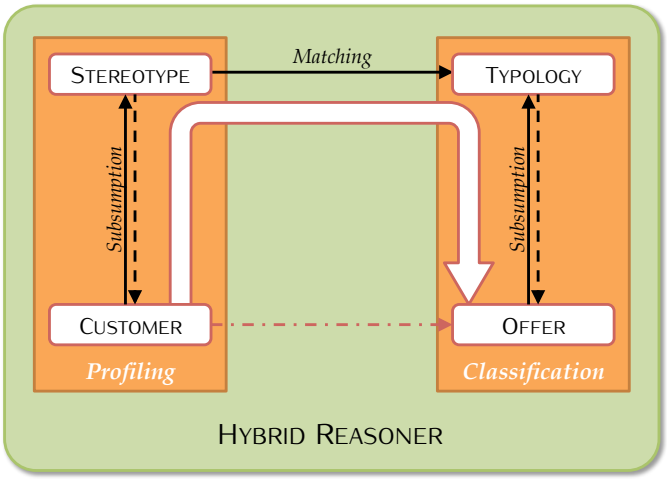


Figure 5.5: The recommendation mechanism inside the hybrid reasoner as the sum of vertical and horizontal reasoning processes.

```
1 rule "Cultural offers to senior customers"
2 filter 0.85
3 when
4   $o: Offer( this isA CulturalOffer )
5   $c: Customer( this isA SeniorCustomer )
6 then
7   insert(new Mail($c.getMail(), "Offer!", $o));
8 end
```

Listing 5.3: Example of intermediate hybrid rule in eTourism.

finally available, the previous rule – involving only the vertical reasoning processes – becomes as follows [10]:

A higher level of abstraction

Unfortunately, the pairings between `Typology` and `Stereotype` objects are hardcoded inside the rules. If one imagines to implement the horizontal reasoning process by describing the connections suggested by the domain expert inside the domain ontology, the whole recommendation process encompassing all the stereotypes and typologies becomes more flexible and it could be implemented by a single rule. If we suppose, for instance, that the matching function is expressed by means of a relationship `matches` between instances pertaining to the concepts `Typology` and `Stereotype` and that the abstraction process is evaluated with fuzzy subsumption, we obtain the rule in Listing 5.4 on the next page. In this case, in fact, the rule identifies the `Stereotype` (line 4) and `Typology` (line 6) of any couple of `Customer` (line 5) and `Offer` (line 7). Then it determines if an association between `Stereotype` and `Typology` exists and, if so, it implicitly computes the strength of such correlation (line 6). If it is too weak,

10 Fuzzy evaluation implicitly comes into play in the semantic statements of the ontology

```
1  rule "Offers to customers"
2  filter 0.85
3  when
4    $s: Stereotype ( )
5    $c: Customer ( this ~isA $s )
6    $t: Typology( matches some $s )
7    $o: Offer( this ~isA $t )
8    // Get the ADT for priorities, defined as:
9    // Map<Customer, Map<Offer, Degree>>
10   ?priorities( $p; )
11 then
12   Queue queue = $p.getQueue($c);
13   queue.insertOrd($o, chance.getDegree());
14 end
```

Listing 5.4: Example of advanced hybrid rule in eTourism.

namely a result whose degree is lower than 0.85, the activation is purged from the AGENDA of the PRS. Conversely, we retrieve a reference to the data structure that holds the results in order of relevance on a per customer basis (line 10), so that we can update it as a result of the activation of the rule (lines 12 and 13). Notice that we still discard the results that are too irrelevant, but we organise the surviving ones in ordered queues to populate the customer's home page of a eTourism site with a list of offers presented in order of relevance in which he may be interested.

5.4 MONITORING AND CONFORMANCE OF SERVICES

This use case is provided by the former DEPARTMENT OF COMPUTER SCIENCE (DCS) now DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DISI) of the University of Bologna and carried out as part of the *Depict* project of the former DEPARTMENT OF ELECTRONICS, ENGINEERING AND SYSTEMS (DEIS) now DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DISI) of the University of Bologna and the EU-FP7 *Farseeing* project.

Use case

The goal of this work is to verify whether it is feasible to use our rule-based implementation of the EC to observe the state of a WS server in a context of SOA. Instead of using a traditional WS framework, we conducted our experiments on a new, very promising technology called *Jolie* that was introduced by the former DEPARTMENT OF COMPUTER SCIENCE (DCS) of the University of Bologna. *Jolie*[11] is a full-fledged programming language and development platform based

11 http://www.jolie-lang.org/

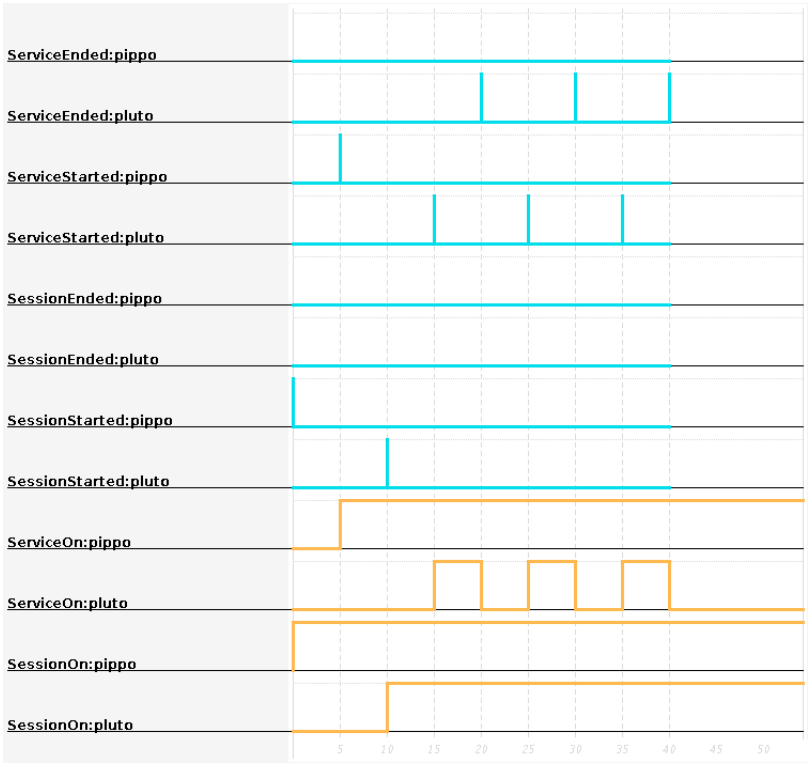


Figure 5.6: The graphical output of the service that displays the outcome of the deductive reasoning performed by the Event Calculus module when observing the first steps of a *Jolie* server: starting and ending events for services and sessions are captured (cyan, top) and their effects are reflected on their fluents (light orange, bottom).

upon the service-oriented programming paradigm, suitable to both the rapid prototyping of new services or the composition of existing ones to deliver new functionalities. It offers an easy to learn syntax, a formal theoretical semantics and a strongly modular approach. Thanks to its extensible development Application Programming Interfaces (APIs), *Jolie* is suitable to make lightweight services, very complex SOA or bridge systems based on different technologies or communication standards. Being so highly customisable, it is an ideal candidate for monitoring to verify that it is conformant to user's requirements and possibly to adapt its future behaviour according to its past performances. As explained in the continuation of this section, such attempt in the automation of the server is matter of future work, while the initial purpose of this study is simply to determine whether it is feasible to observe such a complex domain with EC and possibly to extract its features in soft real-time. According to the Just-In-Time (JIT) philosophy, in fact, any type of intervention is possible only if the monitoring is sufficiently responsive.

Principle of operations

The *Jolie* interpreter relies on a Virtual Machine (VM) implemented in *Java* that is capable of both deploying new services on the fly and

keeping track of each working session and all their inner operations. We have modelled our EC knowledge base accordingly:

- we extract the notification of initiation and termination of each session and all the operations in it and we feed it to the deductive reasoner as events,
- sessions and operations become fluents whose activity intervals have to be determined.

We have slightly modified the behaviour of the *Jolie*'s VM which now sends any of this information to each proper monitoring service that registered itself to the server. We have have implemented a simple monitor service which basically logs any notification and displays it in a separate web page, as well as a more complicated full-fledged monitor based on EC. This second monitor is organised as a MODEL-VIEW-CONTROL (MVC) application:

- the MODEL is a service containing an instance of the EC module that consumes any notification to maintain a representation of the current internal state of the server,
- the VIEW is a graphical application that depends on the MODEL's content to draw the last few steps of the server's history (see Figure 5.6 on the facing page for a small excerpt of an execution trace),
- the CONTROL at the moment does not allow any change and it is simply concerned with the initialisation of all the components of the pattern.

Finally, we have prepared an additional scripted service that orchestrates some calls to a few fictitious trivial services that were deployed on the *Jolie* VM in a repetitive pattern. Despite the not too excessive complexity of the services and of the interaction patterns, we had evidence that the deductive EC monitor managed to handle all the information flow originated by the *Jolie* interpreter without incurring in any particular problem.

Further improvements

We have further modified this infrastructure to take advantage of our forward multi-valued version of the EC and handle richer indicators of the server state like the amount of available free memory or the reply time. At the moment we have just finished to update the VIEW and we are going to submit the entire system to new tests. If the software passes this additional test phase, it will be used as a key component in the following research that is briefly discussed below.

Towards the Cloud Computing domain

This new research is carried out in collaboration with also the DEPARTMENT OF MATHEMATICS of the University of Padua as well as ITALIANASOFTWARE [12] and CRS4 [13] on focuses on the PAAS and IAAS aspects

12 http://www.italianasoftware.com
13 http://www.crs4.it

that are related with the management of a Software as a Service (SaaS). The problem that we would like to solve is the negotiation of the resources. Here the *service*, the *platform* and the *infrastructure* are seen as independent agents that are collaboratively working in the common environment in a way that is possibly profitable for everyone. The service, for example, may realise that it is struggling to serve clients as the free memory drops or the reply time raises. Therefore it may ask the platform to deploy a new copy of the service to which it can divert part of its traffic. The platform, however, may realise in a similar way that all the resources that it handles are allocated to other services and ask the infrastructure to provide a new VM where to migrate some services and balance the computational load between them. The opposite scenario is possible as well: the infrastructure may need to reserve some computational power to a given customer, so it asks to all the platforms the it operates if it is possible to free some VMs. The platform in turn can ask to the service that deploys if some copies may be merged or some service even suspended to free resources. Notice that the interaction may be started by the platform that tries to rationalise the active services to return some resources to the infrastructure and possibly get back some monetary discount. The desired goal is to use the ECE-rules to model the above interaction patterns and execute them. Notice that fuzzy partitions and labels could be used as well to make those ECE-rules more intuitive for the domain engineers that will implement the management policies in each level.

A tentative solution

At the moment we are identifying the variables that we believe that are needed to address the problem. We have built a model of the domain that we internally use to represent the state of an agent and the type of actions that we expect him to perform. The code contained in Listing 5.5 on the next page shows a possible execution in which a poor Quality of Service (QOS) of services of a given kind (real-time services) or associated to a given class of customers (premium customers) trigger an interaction with the underlying layer to improve it. Unfortunately, however, it still consists in a toy example as we are currently trying to extend it to the real case.

The event that triggers this ECE-rule is the receiving of a `Request` between a `client` *`$c`* and a `service` *`$s`* (line 4). We verify the corollary conditions for that `service` *`$s`* by semantically checking that it is a `RealTime` or a `Premium` service (line 6) and by noting the memory roughly seems full (line 7). In this context, the service is expected to reply (line 10) as immediately as it can (line 11) so, in case of violation, it negotiates with the platform *`$p`* the deployment of a new service like *`$s`* to hopefully better serve the client *`$c`* (line 13).

```
rule "Example policy"
filter 0.7
when
  $r: Request( $c: client, $s: service )
  Service( this == $s,
    this isA RealTime or this isA Premium,
    resource ~seems Memory.FULL )
  ?platform( $p; )
then
  expect Reply( service == $s, client == $c
    this ~seems Time.INSTANT )
  on violation {
    negotiate($p, $s, $c);
  }
end
```

Listing 5.5: Example policy for the automatic handling of SaaS, PaaS and IaaS.

5.5 ENGINEERING THE POLICY-MAKING LIFE CYCLE

The following considerations are work in progress that is carried on for the *e-POLICY* EU-FP7 STREP project No. 288147.

Use case

It aims to support policy makers in their decision process across a multi-disciplinary effort aimed at engineering the policy making life-cycle. In particular, global and individual perspectives on the decision process are finally merged and integrated for the first time. The project focuses on regional planning and promotes the assessment of economic, social and environmental impacts during the policy making process (at both the global and individual levels). For the individual aspects, *e-POLICY* aims at deriving social impacts through opinion mining on e-participation data extracted from the web. To aid policy makers, citizens and stakeholders, *e-POLICY* heavily relies on visualisation tools providing an easy access to data, impacts and political choices. The *e-POLICY* case study is the *Emilia Romagna* Regional Energy plan. *e-POLICY* aims to provide a tool that supports the regional planners when they create an energy plan that is in line with strategic European and national objectives, consistent with financial and territorial constraints, participated including opinion mining results, well assessed from an environmental perspective and optimal with respect to one or more metrics. In addition to the regional plan, *e-POLICY* will provide a portfolio of implementation instruments (namely fiscal incentives, tax exemption, investment grants) for pushing the society and the energy market to go in the direction envisaged by the plan.

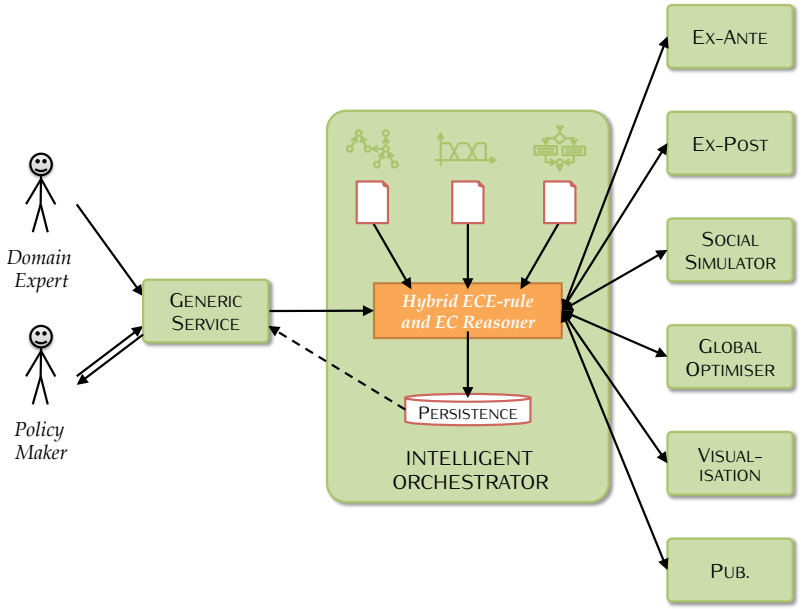


Figure 5.7: Tentative workflow for the *e-POLICY* architecture.

Desiderata

This project is very ambitious because it tries to combine several technologies that are greatly different from each other. These technologies are not only based on different assumptions and perspectives, but they also have contrasting operating modes, executions times, involved actors with diverse roles. From a purely technical point of view, integrating such systems is a non-trivial challenge. A first step in this direction consists in converting each partner's prototype into a WS. SOA was introduced in fact to minimise the impact of local technical choices on large scale heterogeneous systems by defining standards and effectively decouple each single part from the others. What SOA unfortunately does not address is the semantics of the interactions. Consider for example one of the partners' services [14]: it is very likely that it requires some other data – possibly some results computed by other services – to provide its answer (see Figure 5.7). Some of these services have fast computation time and they provide an answer in almost no time. Others, unfortunately, require long simulation process, so they can provide their answer only after a long wait. In order to make answers always available, a persistence layer is needed to cache results for later use. Nevertheless, a persistence layer is not sufficient to completely solve the problem as a service may always need an updated set of input which has not yet been computed by other services. Visualisation tools, for instance, may need to access several alternative sets of results for each single problem since they need to show the users a few alternative results depending on the most common input parameters. If a user moves the parameters

14 The generic service is any service listed on the right side of Figure 5.7 plus the front-end service.

out of the most typical intervals, however, new swift computations or long simulation may be needed again. Consider that the final tool will have at least two kind of users, the policy maker – which uses the software to see the outcome of some possible choices before promulgating the policy, and the domain or regional expert – that tunes the components to the specific territory. If we assume that each service loads the configurations that are specific of a territory at the time of its deployment, we delegate to the persistence layer the maintenance of this information and the system should be able to handle both kinds of users with no need for any additional technology or service.

Outlining a possible scenario

Currently, two prototypes of the various services of the systems are available and the others will follow soon so we started to do some experiments with them. Our idea is to build an additional service as suggested in Figure 5.7 on the facing page(possibly reusing the same technology presented in the previous section) to include an intelligent orchestrator which keeps track of the state of the information included into the persistence layer (see Chapter 2 on page 21) whose reasoning core is based on ECE-RULES (see Chapter 4 on page 91) and hybrid reasoning (see Chapter 3 on page 77). At the moment, due to the low amount of practical and realistic use cases, we can only conclude that the mechanism for the intelligent management of persistence is based on the concept described by the pseudo code available in the following Listing 5.6. The first rule is an ECE-RULE whose meaning is that each time a service requests some data, that data is expected to be provided within a couple of seconds. If so, the request is evaded and the expectation is archived as a success. If not, a `Guard` object is asserted into the WM for later use and the client is notified with a message that warns him that the computation will take a lot, and a new message will inform him when the results will be available to see, letting him playing with other settings whose results are possibly already available. The `Guard` object mentioned above is a condition for a new rule that triggers when the data that is being computed will eventually become available. When such a condition applies, the rule repairs the violated expectation that is still pending, retracts the `Guard` which is obviously no more needed and finally notifies the client about the now available results.

Notice that other customised rules will be needed as well to delegate a computation to one of the services or even orchestrate a few services to provide a complex result. This is probably the most intellectually exciting and challenging aspect of the problem to solve. Unfortunately it is not possible to start to work on a possible solution until the dependencies between services, and the data that is supposed to be exchanged between them is clearly identified. It is our intuition, however, that the kind of hybrid reasoning that we have described in this dissertation will address the problem, especially if a simple ontology that models the orchestration is provided.

```
rule "Managing persistence"
when
  $r: Request( $c: client, $id: data )
then
  $e: expect Data( this == $id, this after[2s] $r )
  on fulfilment {
    // results available or computed on the fly
  }
  on violation {
    insert(new Guard($id, $e, $c));
    notify($c);
  }
end

rule "Notifying completion of simulation"
when
  $g: Guard( $id: data, $e: expectation, $c: client )
  Data( this == $id )
then
  repair($e);
  retract($g);
  notify($c);
end
```

Listing 5.6: Declarative pattern to address the persistence problem in *e-POLICY*.

5.6 FORMAL VERIFICATION AND CONTROL OF SEQUENCING BATCH REACTORS

This practical application is the result of a joint work with the Protezione Unità Tecnica Valutazioni Ambientali – Laboratorio Protezione e Gestione della Risorsa Idrica (UTVALAMB-IDR) of the Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile (ENEA), the Dipartimento di Ingegneria Idraulica, Ambientale, Infrastrutture viarie, e Rilevamento (DIIAR) of the Politecnico di Milano and the Biomedical Informatics Department of the Arizona State University.

Use case

Given the complexity of a water treatment plant, its automated management requires an organic combination of different pieces of information, from background knowledge to control policies. The various criteria need to be evaluated almost in real-time, matching them against data coming from various sources, including the probes installed on the plant. Depending on the result of the evaluation, appropriate actions have to be taken. When an Environmental Decision Support System (EDSS) is dedicated to implementing these

functionalities, it is essential that it can be integrated with the information sources and the control channels, effectively closing the loop between the plant and itself. Moreover, the internal structure of the EDSS should be flexible and modular to deal with the various sub--tasks. An overview of possible EDSS architectures can be found for example in [168], where a prototype architecture is also discussed, which has been evolved to integrate the concepts discussed in this paper.

A general architecture

The architecture we propose is based on an ENTERPRISE SERVICE BUS (ESB), where events are collected and routed to those subsystems for which they are relevant. The approach, in fact, is based on a combination of a SOA and EVENT-DRIVEN ARCHITECTURE (EDA), where the interaction between the services can either be tightly coupled, loosely coupled or even decoupled as needed. These components have several roles, such as services for data storage, user interaction, internal event delivery and external integration.

Principle of operation

In this section we will cover the work that revolves around the component that handles the knowledge about the domain and exhibits the more interesting reasoning capabilities. This component formalises and enacts some control policies for the SEQUENCING BATCH REACTOR (SBR) that have already been partially used in some of our previous works, however we propose here a new model that extends and reconciles the available expertise in a more general, robust and elegant way being based on the theoretical contributions described in this dissertation. In fact, we argue that the role of the SBR control system consists essentially in monitoring the state of the process and any factor influencing it, tracing its progress and ultimately trying to ensure that its outcome matches the plant operator's expectations in terms of effectiveness and efficiency. This goal is reached by:

A. defining an adequate event and fluent ontology to model the state of the process and its relevant events,

B. incorporating a (complex) event processing system which delivers or generates the events appropriately,

C. using the EC formalism to infer the state of the process on the basis of the detected events,

D. using the ECE-RULE to define the desired behaviour and link the policies and actions to be executed in case of informations and violations.

WWTP ONTOLOGY Figure 5.8 on the following page sketches a fictitious and idealised representation description of a WWTP operated with SBR. In order to provide a formal model of the domain, we are in the process of authoring a full ontology which contains and defines the concepts that are related to the automation and control of

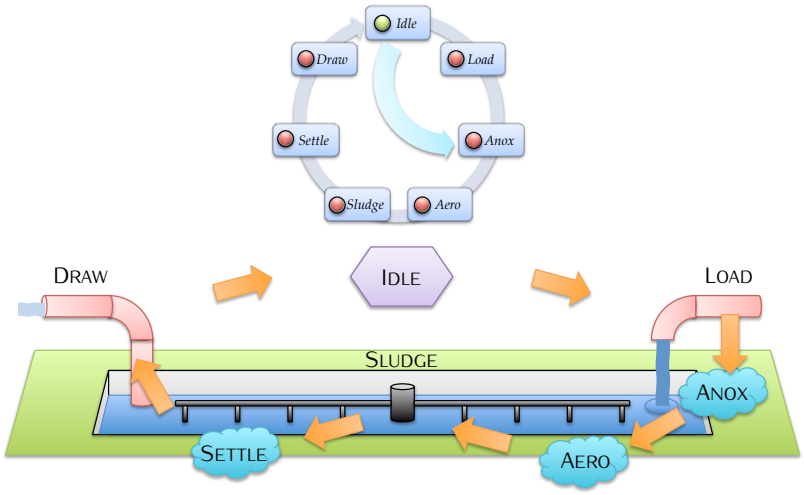


Figure 5.8: An idealised sketch of a Waste-Water Treatment Plant operated with a Sequencing Batch Reactor.

the WWTP. This activity is still work in progress, so at the moment we use a simplified version which contains some of the relevant concepts and their properties and provides a proof-of-concept of the vocabulary used by the EC and ECE-RULES. All events have at least a timestamp and may have different payloads to store the information that they are notifying. We use the following lexicographic convention: if `Payload` is the name of a payload, then `PayloadEvent` and `PayloadFluent` are respectively the names of the event and fluent that have such payload. As seen in Chapter 3 on page 77, these definitions are later converted into classes and interfaces or other equivalent definitions for properly executing the domain. An outline of the relevant events and fluents is shown in Table 5.1 and Table 5.2 on the facing page.

The scenario

In particular, events and fluents are defined at different levels of abstraction, according to the COMPLEX EVENT PROCESSING (CEP) principles. At the lowest level of abstraction, events are used to capture and deliver the values collected by the probes installed in the plant. In particular, different type of `Sample` instances are relative to different physical and chemical signals collected in the tank. In our concrete system, the raw samples are first preprocessed (denoising, outlier filtering, error interval estimation, etc.) and then streamed into the various signal processing submodules which try to extract more relevant events and states. When dealing with SBRs, `Trend` and `TrendChange` objects are particularly relevant: a trend is a state related to the first time derivative of the time series, and may be further specified in `Stable`, `Rising` or `Falling`. Other trends might be defined (e. g. `Oscillating`), but are not usually part of the SBR domain. Dually, a `TrendChange` is an event marking the transition from one trend to another: in particular, we are interested in `Apex` (local minima and maxima) and `Knee` (stabilising rising or falling trends followed by a new stabilising or falling

EVENT	PROPERTIES	DESCRIPTION
Sample	Signal, Amount, Filtered	Probe-sampled data
TrendChange	Signal, Type, Extension, Slope	Characteristic points (max, min, ...)
EndReaction	ReactionId	Reaction complete
NewCycle	CycleId	Start of treatment cycle
NewPhase	PhaseId	Start of process phase
Switch	PhaseId	Process phase to set
Next		Switch to next phase

Table 5.1: List of events with properties and description for the Sequencing Batch Reactor.

FLUENT	SUBTYPES	DESCRIPTION
Trend	Rising, Stable, ...	Signal trends
Process	Denitrification, Nitrification, ...	Chemical processes
Phase	Idle, Load, ...	Treatment phases
Cycle	UrbanCycle, ...	Cycles

Table 5.2: List of fluents with sub-types and description for the Sequencing Batch Reactor.

trend) since they usually allow to identify characteristic points in the process.

At an intermediate level of abstraction, we use events and fluents to model the bio-chemical processes taking place within the tank, such as the cited nitrification and denitrification. On top of this, we model the cycle and its phases, using one fluent for each phase. So, the state of `IdlePhase`, `LoadPhase`, `AnoxicPhase`, `AerobicPhase`, `SettlingPhase`, `DrawPhase` and `DischargePhase` determines whether the current cycle is in a given phase or not. The commutation between two phases is marked either by the `Switch` event, making the source and target phases explicit in the transition, or by the `Next` event, implicitly switching from the current phase to the next, as defined by the canonical treatment cycle (see Figure 5.8).

COMPLEX EVENT DETECTION Our system is completely agnostic with respect to the way the events are generated or detected. This, in theory, would allow to include a number of the different techniques

```
rule "EndOfDenitrification - EC"
when
  $max: Max( $time: timestamp, signal == "pH" )
  $dkn: FallingKnee( signal == "ORP",
                     this overlaps[-5m,5m] $max )
  $den: ?denitrification()
  ?holdsAt( $den, $time; )
  $aph: ?anoxicPhase()
  ?holdsAt( $aph, $time; )
then
  insert(new EndOfDenitrification(...));
end
```

Listing 5.7: Recognition based on EVENT CALCULUS of the `EndOfDenitrification` event.

for the process phase detection among those available in literature. From an architectural perspective, they can either be deployed as independent, external services or even embedded within the reasoning component, exploiting the concept of hybrid knowledge base [31]. Regardless of their integration mechanism, they can be roughly divided in two categories:

PATTERN-MATCHING TECHNIQUES where `Sample` $\mapsto$ `lstinline`

CHARACTERISTIC POINT TECHNIQUES where `Sample` $\mapsto$ `lstinline` $\mapsto$ `EndReaction`

The scenario

The former category uses sub-symbolic data processing techniques – i. e. neural networks, PRINCIPAL COMPONENT ANALYSIS (PCA), clustering, etc. – to analyse the time series, while the latter include an intermediate step where characteristic points in the signals (correlated to the process) are first detected and then used to recognise the state of advancement of a process. The second category is particularly interesting, since it can be revisited and analysed in the context of CEP and EC. First of all, some techniques focus on the identification of `Trend` objects (more or less explicitly as fluents) and use `TrendChange` events to mark the transitions, (clipping and declipping the fluents as needed) while others look directly for the relevant characteristic points and ignore the intermediate behaviour of a signal. `Trend` and `TrendChange`, then, can be used to define the conditions for the recognition of an `EndReaction`. For example, it is well known that the contemporary detection of a local maximum in the pH and a falling knee in the OXIDATION-REDUCTION POTENTIAL (ORP) during the anoxic phase is an indicator of the completion of the denitrification reaction.

This could easily be modelled in terms of reactive EC in a form similar to the one shown in Listing 5.7. Notice, however, that our framework would also support the stronger form shown in Listing 5.8. This

```
rule "EndOfDenitrification - ECE"
when
  $max: Max( $time: timestamp, signal == "pH" )
  $den: ?denitrification()
  ?holdsAt( $den, $time; )
  $aph: ?anoxicPhase()
  ?holdsAt( $aph, $time; )
then
  expect $dkn: FallingKnee( signal == "ORP",
                  this overlaps[-5m,5m] $max )
  on fulfilment {
    insert(new EndOfDenitrification(...));
  }
end
```

Listing 5.8: Recognition based on EVENT-CONDITION-EXPECTATION RULES of the `EndOfDenitrification` event.

alternative version would state that, while the denitrification process is taking place during the anoxic phase, the detection of a local maximum in the pH can only happen at roughly the same time a knee is detected in the ORP, leading to the recognition of the end of the process, otherwise it will be considered a violation. In our system, we actually prefer the first form, since the signals are noisy and spurious local trend changes are not uncommon, but we still support rules of the latter form as monitoring guards.

STATE MANAGEMENT The cyclic nature of the SBR plant allows to model its core functioning with the seven fluents introduced in the domain ontology, one for each phase. The current state determines univocally the plant configuration for that phase, which is obtained issuing an appropriate set of commands to the plant actuators. While production rules can manage this aspect directly, we preferred to separate the state management from the control aspects. In particular, we modelled the operational aspects of the plant management as a business process and delegated its execution to a workflow engine [30]. In a nutshell, we used the business process to reflect the current state of the plant, while the fluents model the process from the perspective of the monitoring system. The socio-technical business process, then, includes external human tasks to model the actual process phases and internal tasks to configure the plant appropriately. The fluents and the external tasks are synchronised, assuming that the plant reconfiguration is instantaneous. These levels of separation allow to decouple the general principles, suitable for any SBR, from the concrete, context dependent details such as the specific commands and channels to interact with the plant. While preferable from a conceptual point

of view, in our case this does not even imply an integration overhead, since the platform acts as a hybrid rule-process engine.

The scenario

The scope of this work, then, will remain focused on the process management aspects, assuming that an appropriate subsystem translates the currently decided state into an appropriate set of commands for the plant. The state, then, can be controlled using two main types of events, `Switch` and `Next`. `Switch(from, to)` assumes that the fluent associated to **`from`** is currently holding, *terminates* it and *initiates* the fluent associated to the phase `to`. The event `Next`, instead, is used to abstract away the sequence of the phases, potentially allowing to reconfigure the sequence to apply different treatment processes. Internally, `Next` is mapped into `Switch` by simple rules which could be expressed in pseudo-language as `Next() and ?holdsAt( $anoph, $time; ) ↦ Switch($anoph, $aerph)`.

The use of `Next` and `Switch` further decouples the act of commuting from one phase to another from the policy adopted to make the switch. At least two of the most diffused policies can be integrated trivially:

- Manual Policy The events are generated directly, through some user interface,
- Reaction Complete Policy According to this optimal policy, the `Next` event is generated when an `EndReaction` event is detected while the appropriate `Phase` *holds*.

The third classic policy that sets a fixed duration for the phases can be modelled in a slightly different way, as shown in Listing 5.9. We assume that a notification event is generated upon entering a new phase, such as the anoxic phase. This event triggers an expectation for a `Switch` within a deadline, corresponding to the maximum allowed duration for that phase. In case that no explicit `Switch` is detected, the system is forced to commute to the next phase. This representation has several advantages. First of all, it allows to express the basic policy, which is more often than not delegated to a component different from the EDSS, in the same logic framework as the more intelligent ones. Second, it can be integrated with other policies which will usually override it. Third, it will apply in case none of the more optimal policies is applicable: in this sense, a commutation triggered by a violation will highlight the fact that something has not gone as expected in the system, potentially triggering some kind of diagnostic process.

However, other types of commutation policies are possible and can be added incrementally, provided that their final output is a `Switch` or `Next` event. For example, the ``Abort policy'' can be considered the dual of the ``Reaction complete'' policy, since it opts for a commutation when it is clear that the current process *cannot* be completed, so it would be useless to wait for the phase to reach its maximum duration.

```
rule "Anoxic Phase Watchdog"
when
  $init: NewPhase( $time: timestamp, $id: id=="anoxic" )
  $aph: ?anoxicPhase()
  holdsAt( $aph, $time; )
then
  expect $swt: Switch( from == "anoxic",
                this after[0,90m] $init )
  on violation {
    insert(new Next());
  }
end
```

Listing 5.9: Example of a maximum phase duration policy.

PROCESS COMPLIANCE The rule in Listing 5.9 is a relevant example of a more general class of rules which can be added to the EDSS knowledge base. In fact, water treatment processes are characterised by a high degree of variability and noise, which has to be added to the measurement noise introduced by the probes and the observation noise introduced by the estimation algorithms. For this reason, the consistency of each measured or estimated quantity or states should be checked. The notion of expectation is particularly suitable in this context, since it allows to introduce soft constraints [15] to describe the ideal behaviour of the system in a given circumstance, as defined by the domain experts. The fulfilment (respectively the violation) of these expectations may lead to the execution of appropriate management policies, or – in the case of violations – diagnostic and recovery procedures. However, an expectation is weaker than a constraint, since its violation usually denotes something which is possible but undesirable, as opposed to something which is impossible or prohibited.

In practice, a relevant amount of domain and process knowledge can be formalised in terms of expectations, and used to define soft constraints on the whole execution of the process, in terms of both desired and undesired behaviour. This approach generalises the idea of automating the plant management by trying to recognise the end of the process reactions (implicitly assuming that the process is successful). Here the plant automation, instead, is achieved by continuously monitoring the process, trying to recognise both desired and undesired situations and acting accordingly.

The scenario

For a process to be considered ideal several expectations are involved at different levels of abstraction: these expectations, then, can be formalised as ECE-RULES which use the domain events and fluents

15 Typically in Computer Science, soft constraints are constraints that can be violated in opposition to hard constraints that must be always satisfied.

as triggers and conditions. Most of the events defined in Table 5.1 are involved in at least one expectation criteria. The criteria themselves may derive from literature, domain expertise or statistics on the historical data.

SAMPLE All signals have expected ranges during the various process phases. The Dissolved Oxygen (DO) concentration is expected to be close to 0 during the anoxic phase, around $2mg/l$ during the nitrification process and then saturate to around $6mg/l$ for the rest of the aerobic phase. The pH is expected to stay within the range 6-8. Redox potential is expected to be negative during the denitrification process (roughly $-200mV$), strongly negative during the anoxic phase, after the end of the denitrification process, but to become positive (around $200mV$) during the aerobic phase.

TRENDCHANGE Trend changes are expected to be present and correlated temporally over different signals, since they are indicators of the completion of the process reactions. Other trend changes, or the same trend changes without an appropriate temporal correlation, are generally not expected.

ENDREACTION The end of denitrification is expected to be recognised only during the anoxic phase; likewise, the end of nitrification is admissible only during the aerobic phase.

SWITCH Some transitions are not allowed or meaningful, such as commuting to the load phase from any phase different from the idle phase (following the discharge which is expected to have emptied the tank).

Likewise, expectations may be expressed on some of the fluents themselves:

TREND pH and ORP are indicators of reactions, so are not expected to be stable during the process. The DO concentration, instead, is expected to be stable (at different levels) while the reactions are taking place.

PROCESS The duration of the nitrification and denitrification processes should be compatible with the expected average durations, conditioned by factors such as the period of the year, the time of the day and the current weather (temperature and rain).

The use of ECE-rules has the advantage that fulfilments (respectively violations) can be managed explicitly, especially in the case of positive expectations (respectively negative) within the rules. Moreover, the reification of expectations, informations and violations also facilitates the addition of specific additional rules, facilitating the expansion of the system, and makes the full process automatically more traceable, since any of these relevant events is implicitly recorded.

5.7 SUMMARY

This Chapter is a showcase for some of the practical applications that have been addressed thanks to the theoretical aspects and technologies that have been presented in this dissertation. This thesis, in fact, tries to show how these contributions can be interesting and innovative from a research point of view, and this Chapter strives to present their several practical implications in many application fields. This Chapter, in particular, includes two applications in CCG one involving machine learning and human interfaces through a *Kinect* and the other a double human interaction between providers and patients, a very refined recommendation system in the context of eTourism, a quite complex use case that stems from SOA and covers some aspects of CC, a contiguous problem that involves the effective handling and orchestration of services and, last but not least, a foray into automated controls domain and industrial plants in where we show how our contributions may handle WWTP in a better and greener way.

Part III

FINAL CONSIDERATIONS

6 LAST THOUGHTS

«Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.»

— Sir Winston Leonard Spencer-Churchill
British Orator, Author and Prime Minister during World War II, 1874-1965

This Chapter that concludes the dissertation contains some final reflections to draw the conclusions. These conclusions are presented in a proper section after that the main contributions of this work are summarised. After that, we propose some additional insights which aim to suggest directions for further and new research. Despite being the concluding Chapter, the thesis still contains interesting Appendices. In the first one, we introduce Production Rule Systems (PRSs) and we discuss some interesting features of one of them. This Appendix contains introductory notions on *Rete*-based Rule-based Systems (RBSs) that should assist the reader who is not very familiar with this topic and provide the pieces of informations

This is not original research, but it assists the reader who is not very shod on RBSs based on the *Rete* algorithm during the reading of the dissertation that takes those concepts for granted. In addition, the several original examples included in that Appendix can be considered as an effort to improve author's teaching skill. The second Appendix focuses instead on solving probability graph problems within a PRS environment. Although this topic was not initially considered, it proved to have some interesting potential after some tests on preliminary proofs of concept. Since this work is still in progress, we decided to present in an Appendix rather than in a separate Chapter.

6.1 CONCLUSIONS

A quick summary of this thesis' content

In this dissertation, we have faced the problem of assessing the conformance of a complex process with respect to a description of the expected behaviour while the process evolves. This a well-known problem in literature that spans across several fields such as Cloud Computings (CCs) and Web Services (WSs) or more generally speaking Service-Oriented Architecture (SOA), in Multi-Agent System (MAS), Business Process Modelling (BPM) and Computerised Clinical Guidelines (CCG), just to name a few.

The approach that we have followed is also adopted by a number of other researchers in the scientific community. This approach initially consists in assisting the domain where the processes take place while they evolve in order to always have an updated representation of its state. We have addressed this task by introducing an optimised implementation of the Event Calculus (EC) based on forward rules (see Chapter 2 on page 21). This implementation is inspired by the most famous and performing families of EC and it exploits the features of the underlying PRS (in particular its ability to reason about temporal events) to meet some common criteria of robustness and usability, even enabling more advanced reasonings thanks to Fuzzy Logic (FL).

Then we have introduced a framework with whom we compare the current state of the domain with some definition of ideal behaviour that was provided in advance (see Chapter 4 on page 91). This comparison adheres to the Just-In-Time (JIT) philosophy, meaning that it is computed in line, as the domain's process execution progresses and the monitor detects these changes. Such framework still takes advantage of forward rules for its implementation and introduces the concept of *expectation* as a way to express the desired outcome that a process should ideally match. The expectations may be seen as well as a method to decouple the comparison process from the evaluation process that determines the extent to which the actual behaviour corresponds to the ideal one. This is achieved by aggregating the results of several evaluations into a single conformance score. This method is general and flexible as it allows the knowledge engineers to tune the comparing function in the most suitable way for the domain. We called *global conformance* the outcome of this operational method (see Chapter 4.4 on page 104). In this way it becomes possible to precisely manage different domains and obtain values for different executions of the same process that are comparable with each other.

This additional degree of freedom enables the possibility to include several customised reasoning styles to better evaluate every possible domain, provided that they return a result in form of a degree to be accumulated in a single global conformance score. In this regard, we have introduced another module that is based again of forward rules that combines Description Logic (DL) and FL reasoning with the rule-based reasoning style (see 3 on page 77). To the best of our knowledge, this is the first piece of software that tries to combine together these technologies so tightly. Such result is not trivial for both the involved practical and theoretical aspects. From a practical point of view, in fact, the *Tableaux* algorithm had to be partially rearranged as some features (like *backtracking*) were not available in the host environment. In this context, FL not only increased the expressiveness of the tool but also allowed us to overcome this limitation by providing an alternative way to compute the results in parallel – with no need of

backtracking – by solving Mixed-Integer Linear Programming (MILP) problems.

From a theoretical viewpoint, instead, we had to face the forced cohabitation of almost opposing background assumptions. RBSs, in fact, typically adopt Closed World Assumption (CWA): an aggressive simplified assumption that allows to consider as false anything that is not explicitly asserted as true. Any DL, instead, adheres to Open World Assumption (OWA): a much more conservative assumption that allows to consider TRUE or FALSE only the facts that can be deducted by the knowledge base, and UNKNOWN all the other cases. Once again, the FL proved to be the solution: every reasoning is performed according to FL since it allows a possibly more expressive but at least richer domain modelling and then specific Custom Operator (CO) are provided to allow the user to interpret those results as in CWA or OWA. Notice, however, that it is still difficult to imagine a rule involving aspects that need to be addressed in CWA rather than in OWA at the same time, but this framework at least allows to practically and feasibly include rules working in OWA and CWA within the same rule base.

Also notice that this approach could be extended towards other kinds of logics or technologies, as it is suggested in Appendix B on page 183 where some efforts are described to support Probabilistic Inductive Logic Programming (PILP) within a PRS.

6.1.1 Some considerations

Less is more

The general approach that we have adopted in this dissertation – which can be summarised in monitoring and verification of conformance – is not really innovative as it is already largely addressed in literature (see Chapter 1 on page 1). Our goal was however to make the approach more adoptable and, in a sense, more user friendly. For example, we have devoted much time and resources to understand how to make the approach more flexible and expressive as possible. With respect to other available solutions, it consisted initially in understanding what parts or features it was possible to hive off. This may sound counterintuitive, but it is the only way to make space for additional features. Also notice that, in some cases, getting rid of something means to relax some constraints, possibly leading to better overall results or performances. An example of such behaviour is the adoption of FL which actually simplified the implementation in forward rules of the *Tableaux* algorithm. Another example is the concept of a generic conformance function to whom delegate the conformance checking that is defined by the user on a domain basis. In other former solutions, for instance, such evaluation was encoded so precisely to force the user to over-complicated domain modelling to address specific needs like the relaxed temporal evaluation of human tasks in socio-technical systems. Moreover, we were able to fill the

empty space left by applying this streamlining approach with additional challenging styles of reasoning.

Once such a method is outlined, it is possible to introduce as many additional evaluators to make it more and more adaptable and powerful. While this is a big advantage it also poses a serious problem of knowledge representation. As we have noticed while developing the hybrid reasoner in Chapter 3, it is not easy to mix several kinds of knowledge together as they may refer to different assumptions or semantics, requiring the user to be consistent in all its definitions. We will discuss and propose a possible solution to this issue in the following section.

Comparison with very similar solutions

It is also interesting to consider and discuss about the main differences between our own contribution and other rule-based approaches (based on PROLOG) which are affine and already adopted in the past by our research group. Both approaches are declarative and it is the author's opinion that it represents a major advantage over former approaches. The declarative paradigm allows to express the logic of a computation without describing its control flow. Complex instances of a problem can be solved more easily and clearly by only focusing on what computation to perform rather than how to compute it: the rule base for both variants of EC in Chapter 2 on page 21 and the handling of *expectations* in Chapter 4 on page 91 are an example. The most evident difference between these approaches relies instead in the chaining. The more classic solutions adopt backward-chaining while our contribution uses forward-chaining. The state-of-the-art implementations of both these sub-paradigms, in fact, include optimisations that make the comparison a draw, in the sense that no platform clearly outperforms the other so the user is free to choose the one that he likes the most or that best fits his other project requirements. The comparison between the expressiveness of the two solutions may be considered a draw as well as they both are versatile and include several modules addressing specific issues.

Limitations of our solution

One of the greatest criticism that is moved against modern forward-chaining RBSs is that they are downgraded by side effects. These implementations, in fact, are not *referentially transparent* as they try to ground a *purely functional* style on OBJECT-ORIENTED PROGRAMMING (OOP). Backward-chaining tools like PROLOG, instead, are generally praised for their logical soundness which is used, for example, to formally prove some property of a computed result. There are recent works, as it is emerged from the analysis of other PROLOG-based implementations of the EC, in which those advantages are traded for speed thus making the distinction between the two approaches increasingly blurred.

Some very personal final considerations

If we want to determine a discriminating aspects between the two models at any cost, we can say that the former continues a longer tradition in where it is possible to find inspiring solutions and works

and the latter is more intuitive and easier to learn. This intuition relies on the perception that the generative approach is simpler to imagine and the pattern matching easier to master with respect to unification. Once again, especially in the second case, it depends on the tastes and preferences of the user, so it is really difficult to find a reason that makes a system generally preferable over the other. This conclusion is supported by much research that has been done to determine similarities and differences between these kinds of rules to define general common languages to express that are capable to express both of them.

6.2 FUTURE WORK

Improving global conformance: languages for proper knowledge representation

One of the first aspects that still need to be properly addressed is to determine a proper language to express this kind of problems. At the moment, in fact, we adopt a language that ultimately relies on a slightly modified version of the *Drools* grammar. Such result, however, is not easy to achieve because it should reconcile assumptions and semantics that are very different from each other.

Another non-trivial drawback consists in the variability that we have introduced by considering a customisable function for assessing the global conformance. In effect, if we allow the user to customise the evaluation function at will, we can expect that – sooner or later – they will introduce concepts that may collide with the assumptions that we have made, voiding the feasibility of our approach. This freedom must be limited in some way to ensure the accuracy of the method, as the reader can imagine, however, it is not easy to understand where and how to put these constraints.

If we assume to have found this compromise, we could adopt a DL-based technique similar to the one that we have described for the specific case of *expectations* in Chapter 4. This method requires the definition of a top-level or at least upper-level ontology that contains an outline of what it is possible to express with the tool. Notice that the open nature of ontologies would allow to extend it to include other possible kind of evaluators that might prove to be useful to compute the conformance.

We may force these extensions to derive from other basic concepts or associations in order to limit in a way what the user could add to some safe principle. Notice that such approach often leads to reason about a domain at meta-level. Ontologies support this kind of higher-level reasoning, but it typically implies the explosion of the reasoning complexity. This happens, for example, with OWL ontologies where the more complex instances are not tractable (OWL-Full). Recently, however, the introduction of OWL2 includes a wiser characterisation that suggests possible developments.

Another interesting candidate is Semantics of Business Vocabulary and Business Rules (SBVR) that combines formal logic with modelling in natural language. It defines an environment which natively supports a few formal modality in where to define ontologies and rules in a natural way. Although initially intended as a tool to facilitate the logical reasoning between humans, there are already some software implementations. Many are abandoned and no longer available, but at least one is currently supported and still in active development [1]. These improvements could be regarded as a term of comparison to guide the research in the field of knowledge representation for conformance problems as well.

Probabilistic inductive logic programming and knowledge base revision

Another interesting topic to investigate is how to properly solve probabilistic inductive graph problems with PRSs. The generative and highly-parallelisable nature of these system suggests possible improvements over former solutions, many of them based on Logic Programming (LP) platforms like Prolog.

As the reader may have noticed, some introductory work is presented in Appendix B on page 183. We have presented three approaches that solve problems of increasing complexity in an elegant way thanks to the combined contribution of the declarative forward-chaining rules. The most general solution, however, requires an external imperative tool to determine an efficient strategy to determine probability by means of Binary Decision Diagrams (BDDs).

This step however breaks some advantages of the proposed architecture. The issue could be addressed instead by introducing a new set of rules to handles BDD within the current PRS's Working Memory (WM) with no need to rely on external tools which need to duplicate the representation of the current problem instance. In addition, a few other solution based on even more efficient diagram representations have been proposed.

In either cases, it would be interesting to implement them in forward rules and compare the performances of these implementations. On top of that, some effective grammar could be introduced to express this kind of probabilistic graph problems in a more abstract, higher level. The resulting system would provide an additional evaluator for the global conformance as well as a stand-alone tool for Knowledge Base Revision (KBR).

Imagine, for instance, a two-layered architecture like that described in the Part i on page 21 of this dissertation: the first layer would perform any kind of reasoning required to handle the underlying domain, and the second some statistical analysis about the computation on the companion layer. The statistical analysis could detect unexpected results and trigger probabilistic reasoning to determine possible actions to fix it. Similar approaches, for example, are successfully exploited in Machine Learning (ML) and Deductive Reasoning (DR).

1 http://rulemotion.com

Towards purely functional Production Rule Systems

Although there are many other small or off-topic things or to try, there is at least one last interesting topic to investigate. As suggested above, forward-chaining RBSs are often blamed not to be referentially transparent or to lack any meaningful semantics. This is a recurring criticism which apparently has not yet been satisfactorily resolved.

On the contrary, many high-profile implementations have taken an opposite direction by hybridising with OOP imperative languages that are particularly appreciated by development teams but introduce yet more potentially harmful side effects. As an example, consider for instance a simple rule base. It consists of two rules that are both triggered by the presence in the WM of a given fact: however, the first asserts an additional fact while the second retracts the triggering fact. This rule base can be expressed by the following informal writing:

$$a \rightarrow b.$$
$$a \rightarrow \neg a.$$

In absence of further details, it is unclear whether the expected result is an empty knowledge base (the second rule triggers for first) or a knowledge base containing the fact b only (the first rule triggers first and then the second) when the triggering fact a is asserted.

Depending on the implementations and current configuration of the *conflict resolution* system of the AGENDA (for more detail, see Appendix A on page 153), in fact, either rules may trigger first leading to the two distinct outcomes. There are already a few works suggesting to use the *stable model semantics* of ANSWER SET PROGRAMMING (ASP) to model the several contexts that we may identify by triggering the rules with different contexts [102, 103]. Notice that it would be possible to overcome many of the limitations mentioned above by adapting PRSs to this semantics.

Another criticism that is often directed at PRSs is that any operation involving its WM – despite being similar database transactions – is not ACID. ACID in fact is a very common acronym in DATA BASE MANAGEMENT SYSTEM (DBMS) which summarises the characteristics that a database must exhibit to qualify as a purely *transactional*: atomicity, consistency, isolation and durability. In this regard, in effect, several authors believe that if PRS were ACID, they could adopt the same semantics of databases. A clear semantics or other solutions to make them more functional, in fact, would make the tool for global conformance and any PRSs in general more formal.

Part IV

APPENDIX

A | DROOLS

«The golden rule is that there are no golden rules.»

— George Bernard Shaw
Irish literary Critic, Playwright and Essayist.
1925 Nobel Prize for Literature, 1856-1950

This Appendix provides a brief introduction to Production Rule Systems (PRSs). This Chapter is not meant to be a complete and exhaustive commentary on the architecture, the operating mode and the advanced features of modern declarative forward-rules systems. It is rather an agile collection of references to the environment where the more practical contributions of the thesis have been developed, organically gathered in a single place for the reader's convenience. Although it contains no original scientific contribution from the author, it represents an effort on divulgation – especially in relation to teaching tasks – as it formulate a few meaningful examples on the topic.

The reader that is specifically interested in these topics may find more detailed and useful insights in Forgy's seminal work [62] or in Doorenbos's Ph.D. dissertation [55] on the RETE algorithm which typically operates rule-based systems. Details on other common optimisations for PRS may be find in Batory's work (see [23] as a starting point).

A.1 PRODUCTION RULE SYSTEMS

Rules are one of the most common and basic way to represent the knowledge in many areas of Artificial Intelligence (AI). There are of course several kinds of rules and logics to support them. Among these, *logic programs* and *production rules* are probably the most common form of rules. They are both based on an argument form that defines a *function* which takes some premises and returns a conclusion. This way of reasoning is called *modus ponendo ponens* (the Latin for "the way that affirms by affirming"), often abbreviated in Modus Ponens (MP). More formally if one proposition P implies a second proposition Q and P is true, MP concludes that Q is also true. This implication translates into the following formula:

Rule of inference: modus ponens

$$\frac{P \rightarrow Q, P}{\therefore Q}$$

Consider the following syllogism as an example:

- P → Q: *if* an apple contains poison *then* it is harmful,
- P: this apple is poisoned,
- ∴ Q: *therefore*, this apple is harmful.

Comparing production rules and logic programs

Both are widely used, yet there is a great deal of confusion and disagreement about their differences and mutual relationship [97]. The rules of PRSs – also called RULE-BASED SYSTEMS (RBSs) – have the form "IF *conditions* THEN *actions*" and appears to be similar to conditionals in logic. The most popular textbook on AI [155, p. 286], for example, considers production rules as mere conditionals for forward reasoning. One of the main textbooks on COGNITIVE SCIENCE (CS) [166, p. 43], however, asserts that "*rules are if-then structures*" that, despite being "*very similar to the conditionals*", "*they have different representational and computational properties*". A logic program, instead, is presented as "*a programming language that uses logic representations and deductive techniques*" [176], but it is also included "*among the production systems widely used in cognitive simulations*" [166].

Distinctive features of logic programming

The most famous language for logic programs is probably PROLOG. It is rooted in FIRST ORDER LOGIC (FOL), a formal logic used in several disciplines to express theories about some topic of interest. Unlike many other programming languages, PROLOG is *declarative* as it expresses the logic of a computation without describing its control flow [108]. In these systems, in effect, the program logic is expressed in terms of relations – represented as facts and rules – and the computation is initiated by running a query over these relations [106]. Here rules are called *clauses* and they have an *head* and a *body* each. A body consists of conjunctions and disjunctions of *predicates* that are called the clause's goals. Therefore an head becomes true whenever its body is true. Conjunctions and disjunctions can only appear in the body, not in the head of a rule. Clauses with empty bodies are called *facts* and are always true. Because of the peculiar way of entailing clauses, these tools are said to embrace the *backward-chaining* philosophy. Any time a query is called, in fact, the interpreter tries to resolve it with the heads of available clauses, possibly assigning any free variable with ground terms of the domain. This strategy is called *unification* and it is one of the most distinguishing features of this kind of systems. If the body of the matching clause is not empty, the predicates contained therein are added to the set of conditions to satisfy for a positive response to the query and the whole procedure is applied recursively. In conclusion, whenever the interpreter manages to trace a query back to some facts, a solution is found. Such process is known as SELECTIVE LINEAR DEFINITE CLAUSE RESOLUTION WITH NEGATION AS FAILURE (SLDNF) and it is performed by investigating a derivation – a set of conditions that represent a possible justification

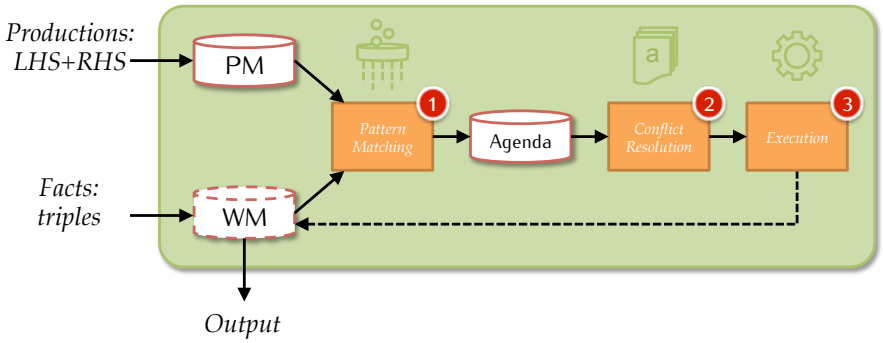


Figure A.1: Architectural outline of a PRODUCTION RULE SYSTEM.

– for the query at a time. When the current derivation leads to no progress or further derivations are needed, the interpreter undos the last operation in favour of the following resolving clause and then continues with the derivation. This behaviour is called *backtracking* and it is another peculiar aspect of these systems.

Distinctive features of production rules

Rules in PRSs are called instead *productions* and they primarily express some behaviour that transforms the available information about the domain in new information. A PRS in practice provides the mechanism necessary to execute productions in order to achieve some goal for the system. Each productions consist of two parts: a sensory precondition which is often called LEFT-HAND SIDE (LHS) representing the conditional expression of an IF statement, and an action which is said RIGHT-HAND SIDE (RHS) and embodies the consequent THEN part of the statement. When the premise of a precondition matches the current state of the world, the production is said to be activated or triggered. When the consequence of a production is eventually executed, the production is said to have fired. A typical PRS is a computer program that contains three databases and performs three different actions (see Figure A.1). The first memory contains all the productions and it is called PRODUCTION MEMORY (PM); the second maintains the data about the current state, knowledge or belief and it is named WORKING MEMORY (WM); the last one is said AGENDA and stores the activations of all the productions.

The first action performed by any PRS is the *pattern matching*. This task is accomplished by building a network of constraints singularly called patterns which correspond to premises of the productions. The algorithm that manages the creation and usage of the network is called RETE (the Latin for "*network*") [62]. The network is fed with the domain knowledge and gradually filters it to identify the data sets that fulfil the preconditions of some production. For each similar set, an activation that includes a reference to both the production and the data set is generated and passed to the AGENDA.

When more than a production is activated at the same time, the PRS requires a mechanism to prioritise them and decide which one to

execute first. The activations simultaneously stored in the Agenda are called *conflict set* and the process the tries to impose an order relation over them is named *conflict resolution*. This is the second task that is demanded to any PRS and it is probably the most fragile part of the whole system. Depending on the specific implementations, the conflict resolution may decide to favour certain activations than others, or even to discard some. Therefore it is easy to understand why these systems produce different results depending on the implementation choices and for a long time they were considered not really rigorously logic tools [96].

The last task performed by PRS is *execution* which takes the already ordered activations and fires them one at a time. The firing of a production consists in performing its actions. There are two main kinds of action: *logical actions* and *side effects*. The logical actions relate to the revision of belief about the knowledge on the domain that is obtained by asserting or retracting data from the WM. As a result, the new state of the world may activate again some of the productions and entail the chaining of rules. This cascade propagation of rules is said *forward-chaining* in opposition to what happens in logic programming systems. Notice that the threads of reasoning that are achieved in this context by concatenation are somehow similar to the derivations that are obtained with SLDNF resolution. Nevertheless they result to be inverted by construction and they are all build simultaneously instead of one at a time. Conversely side effects are actions that do not involve operations on the WM but have some irreversible effect on the domain, like the consumption of some precious resource [1].

Final arguments

Starting from the operating principles described above, we can conclude that these systems perform more or less the same task but they follow two almost opposite approaches. They rely, in fact, on the same rule of inference MP but they make different assumptions so that the resulting systems are not perfectly interchangeable. From a more practical stand-point, PRSs generally use more memory than PROLOG. The reasons behind this statement are essentially two. The first depends on the fact that the WM is a distributed and redundant memory to speed up the pattern matching process, as we will see in the following section. This is why we have represented it in Figure A.1 with a dashed stroke rather than with a continuos line as the PM and the AGENDA. The second motivation depends on the different exploration strategy of the two solutions. PROLOG essentially performs a *depth-first* search finding a solution at a time; a PRS instead adopt a *breadth-first* search that determines solutions in parallel. If we consider this argument in a more abstract way, we can also conclude that PRSs generally perform better than PROLOG implementations. This qualitative consideration does not imply that PRSs are faster than PROLOG interpreters on single runs (in practice it really depends on the specific implemen-

1 The irreversible nature of these actions suggests why backtracking is not supported.

tations and the nature of the problem being solved) but in a general sense. When the first goal is answered, in fact, PROLOG still needs to determine any other solution while the PRS has already computed all of them. PRS are therefore preferable when we can assume the alternation of a processing phase and an interrogation phase which involves several queries at the same time.

Introducing the Drools platform

The reference system that we have chosen to implement our contributions is DROOLS [2], JBoss' free open source alternative of its ENTERPRISE BRMS. DROOLS is an open source "*knowledge modelling and business logic integration suite*" with several modules. The core production system is called DROOLS EXPERT. With respect to standard implementations, it uses *Java* PLAIN OLD JAVA OBJECTS (POJOs) as facts to represents the knowledge about the domain rather than *(subject, predicate, object)* triples similar to the *N-triples* of SEMANTIC WEB (SW) (see inputs in Figure A.1). DROOLS EXPERT therefore combines the power of declarative programming that stems from the adoption of the RETE algorithm for productions' premises with the practicality and comfort of *Java* OBJECT-ORIENTED PROGRAMMING (OOP) for the productions' actions. DROOLS also addresses many specific contexts by introducing modules. Among these, we will cover DROOLS FUSION and DROOLS CHANCE later in this Chapter. DROOLS FUSION [3] is the official extension of Drools that allows *temporal reasoning* within productions and facts, and COMPLEX EVENT PROCESSING (CEP). DROOLS CHANCE is a novel entry among modules which is about to become an official extension as well. It focuses on the uncertain and imprecise reasoning and provides routines to seamlessly manage and propagate these figures through productions. It is described in Sottara's Ph.D. dissertation and currently incubated in `https://github.com/droolsjbpm/drools-chance`.

A clarification on the purpose of this Appendix

Notice that an appropriate introduction to *Drools* would require a more elaborate and detailed discussion on the specific tools mentioned above, as well as the others that are part of their ecosystem. It would also require a meticulous explanation of all the terms of its syntax, the meaning of its annotations, the semantics of its concepts and methods (like, for instance, **`not`**, **`modify`**, **`update`**, **`insert`**, **`retract`**, **`accumulate`**, **`extends`**, **`salience`** and so on) [4] – especially because the same syntax of *Drools* has been chosen as specification language.

Such a detailed explanation, however, would involve an amount of work that is at least comparable to that required for this dissertation. Therefore we have preferred to provide an introduction to the tool by means of practical examples that allow the reader to familiarise with it and get an insight of the operational semantics of its commands. We suggest the reader that is interested in reading the above details

2 `http://www.jboss.org/drools/`

3 `http://www.jboss.org/drools/drools-fusion.html`

4 Annotations, lists and other peculiar aspects of the *Drools* dialects derive from OOP and, in particular, from the underlying *Java* environment.

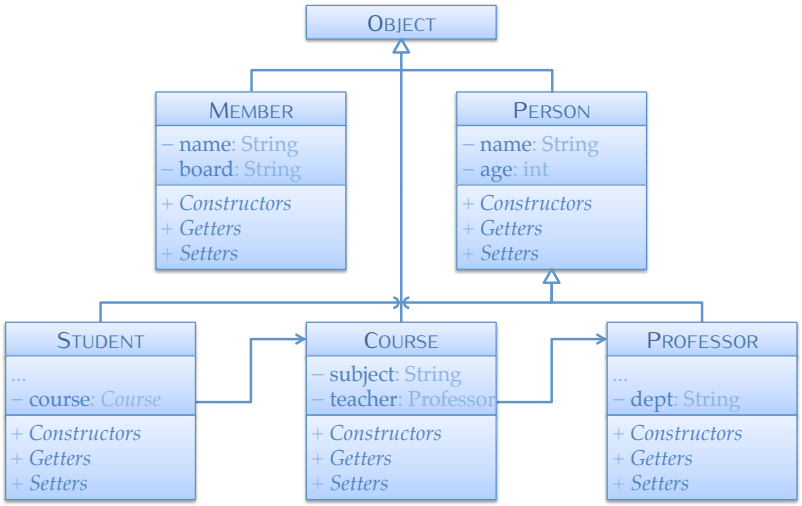


Figure A.2: The hierarchy of classes used in the example.

to visit the DROOLS Web site with the official documentation [5] or to read one of the books about this specific tool [13, 20, 33].

A.2 INTRODUCTORY EXAMPLE

Domain description

This example involves the bunch of related classes that is sketched in Figure A.2. The Figure uses a UNIFIED MODELLING LANGUAGE (UML) class diagram with relaxed conventions about `constructor`, `getter` and `setter` methods that are just sketched. This diagram describes a portion of the world in which there are `Courses`, `Persons` and `Member`s. `Persons` are characterised by a `name` (represented by a string) and an `age` (an integer). `Courses`, instead, are characterised by a `subject` (a string) and a `teacher` which is a reference to a `Professor`, a specialised instance of a `Person`. A `Professor` inherits all the fields of a `Person` and also belongs to a department `dept` (encoded as a string). The domain also includes another specialised class of `Persons` that is `Student`: this class contains a reference to a `Course` that the student is attending in addition to the usual fields of a `Person`. `Members` associates a `person` (a `Person`) to a `board` (a string).

DRL formalisation: facts

The Listing A.1 on the next page shows how to express the same hierarchy of concepts with DROOLS RULE LANGUAGE (DRL) statements: it first declares a `Person` as an entity with a `name: String` and an `age: int` (lines 1–4), it extends that class to introduce a `Professor` with an additional `dept: String` field that is initialised to `"Engineering"` (lines 6–8) and a `Student` with a `course: Course` field (lines 10–12), then it connotes any `Course` with a `subject: String` and a `teacher: Professor` (lines 14–17) and finally it introduces a `Member` with a `person: Person` and `board: String`.

5 http://www.jboss.org/drools/documentation

```
1  declare Person
2    name: String
3    age: int
4  end
5
6  declare Professor extends Person
7    dept: String = "Engineering"
8  end
9
10 declare Student extends Person
11   course : Course
12 end
13
14 declare Course
15   subject: String
16   teacher: Professor
17 end
18
19 declare Member
20   name: String
21   board: String
22 end
```

Listing A.1: The DRL statements to introduce the hierarchy of classes in Figure A.2 on the preceding page.

DRL formalisation: rules

The KNOWLEDGE BASE (KB) on this domain is completed by the rule and the query that are included in Listing A.2. Let us consider the *rule* first. The rule presented here helps to automatically fill the seats of a hypothetical *Advisory Board* for AI. This board will include all the professors older than 50 years old that are teaching some AI course. As a consequence of the rule's activation, any person that is eligible of being member of such a board are endorsed for their role by asserting a `Member` object that basically binds its name to that given subject.

Let us analyse the specific parts of this rule in detail. The *premise* of the rule is a *compound pattern* that consists of two (simple) *patterns*. Each pattern refers to a specific class of objects of the KB; notice that there may be multiple patterns that relate to the same class of objects within the same premise. Each pattern is composed of several *constraints*: the first is about the class type itself; the others refer any comparison involving the values of the fields of the class. The *assignments* of objects or fields to *labels* (also called *variables*) do not really constraint the matching objects but, for practical reasons, they are handled in the same way.

The first pattern (line 4), in particular, selects all the professors older than 50 and binds the instances itself of `Professor` and its `name`

```
1  rule "AI board"
2   salience 5 // rule attributes
3   when
4    $p: Professor( $n: name, age > 50 )
5    Course( $s: subject == "AI", teacher == $p )
6   then
7    System.out.println($n + " is a member of " + $s + "
         board");
8    insert(new Member($n, $s));
9  end
10
11 query students( String $n, int $a, String $s )
12  Student( $n := name, $a := age, $c: course )
13  Course( $s := subject, this == $c )
14 end
```

Listing A.2: A simple KNOWLEDGE BASE on the hierarchy of classes in Figure A.2 on page 158.

field to the variables `$p` and `$n` respectively for a later use. The second pattern (line 5) identifies all the courses whose subject is `"AI"` and held by the above professor `$p`. The subject, associated to the label `$s`, is also returned for later use.

Notice that this second pattern makes use of the `$p` variable that was introduced by the first one: this *cross-reference* between patterns binds together de facto the instances that are singularly determined by the two sets of constraints so that only the couples that satisfy this additional cross-reference requirement are identified by the rule.

The *consequence* of the rule includes a sequence of actions that must be applied to each set of objects filtered by the premise, as soon as they are found. In particular, this *consequent* logs a message on the screen (line 7) and introduces a new instance of a `Member` object to instruct the WM of the adequacy of `$n` as board member for `$s` (line 8).

DRL formalisation: queries

Let us consider now the *query*. The parametric query on lines 11–14 of Listing A.2 works in a way that is similar to a rule's premise. Queries, in effect, are like rules without actions and, in fact, they only return the sets of instances that match with their compound pattern.

With reference to the specific example, for instance, the query identified by the name `students` discovers all the triples `name $n`, `age $a` and `subject $s` of all the students attending a course. Notice that we have again a cross-reference that binds together the instances of `Students` and `Courses`. Queries can be called from both the external *Java* world to return some values that are present in the WM and the premise of the rules. In the latter case, there are several ways to interact with queries, as exemplified by Listing A.3. The simplest mode of interaction is presented in lines 1 and 2: the query identifies all

```
1 student()
2 student( $s == "AI" )
3 student( "Harry Smith", 25, "AI"; )
4 ?student( $n, $a, "AI"; )
```

Listing A.3: Ways of interaction with queries.

the entries that match the pattern (line 1), possibly applying all the constraints that are expressed within the call (line 2). The second call, for instance, returns name and age of all the students that are attending AI courses. Since assigning every single parameter in this specific fashion could be tedious, an alternative positional syntax is provided.

Positional queries

This modality is activated by expressing the ordered list of parameters that are going to be passed to the query separated by commas and terminated by a semicolon (line 3). Notice that, by default, each field of any class receives an index equal to the position of its definition itself within the declaration. These indices are used to pass the value of each parameter to the appropriate field. It is possible, of course, to bypass the default numeration by annotating each field with a statement like `@position(1)`, where the number precisely indicates the index to be assigned to the field of the class that is being defined.

Pull-only queries

By preceding the call with a "?" (line 4), the engine knows that the query is *pull-only*. Usually, in fact, each time that a set of instances that satisfies the query becomes available in the WM, the query is reactivated and the new result is notified. corollaryPull-only calls, instead, respond with the list of matching instances that are currently present in the WM, without reactivating in a second time in case new knowledge becomes available.

Backward- and forward-chaining

Since query calls can be nested one inside another, the authors of our reference PRS claim that this feature enables *backward-chaining* reasoning within a *forward-chaining* tool. Although backward-chaining reasoning is generally more expressive and sophisticated than that, we can consider it a first rudimentary approximation.

Last but not least, notice the usage of the ":=" assignment symbol within the query declaration in Listing A.2. This symbol lets the engine to autonomously interpret the statement as an assignment or an equivalence constraint, depending on whether the involved variable is bound or free. If the variable is set, in fact, it is used to filter the instances that do not match with the current value; conversely this value is assigned to the variable and returned as an output.

Unification

Considering again the parallel with forward-chaining tools, the authors of our reference PRS call this feature *unification* because it resembles the way in which variables are managed in those systems.

Building a RETE network

Now we focus again on the rule in Listing A.2 on the facing page to understand how patterns are interpreted to generate RETE networks

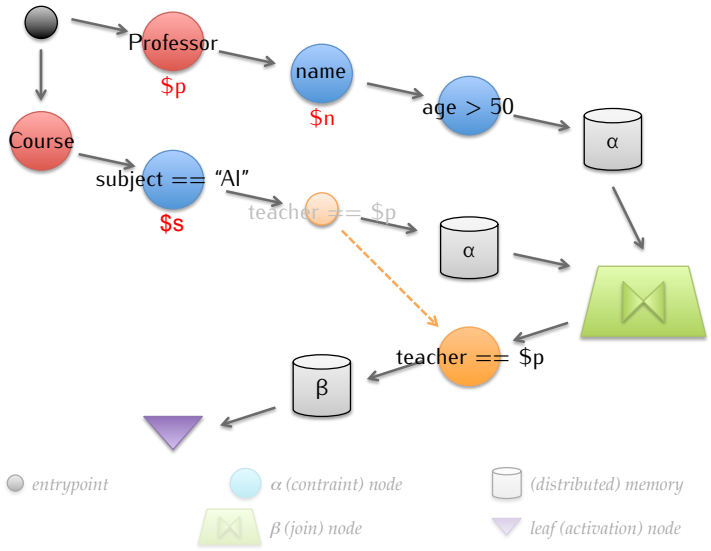


Figure A.3: The RETE equivalent to the premise of the rule in Listing A.2 on page 160.

that efficiently filter objects instances. The diagram in Figure A.3 represent the RETE network that is generated starting from the premise of the rule in Listing A.2 on page 160 (lines 19–27). In this Figure, you can recognise entry-point nodes (drawn as black circles), constraints or α nodes (red, blue or orange dots – more details will follow), joins or β nodes (green trapeziums), (α or β) memories (gray cylinders) and activation or leaf nodes (purple triangles).

Constraint nodes and α memories

Each pattern that is part of the premise becomes a chain of nodes which departs from an entry-point like the default one in Figure A.3. Each pattern refers to an object class and the first node of each chain, in effect, is a constraint on the class type that is mentioned (red constraint nodes). The first pattern of the rule in Listing A.2, for instance, predicates over `Professor` objects, while the second over `Course` objects. Notice that the first pattern also assigns the instances of Listing-Professor that it identifies to the symbol *`$p`* for later use.

Then the constraints on the fields values of the objects of each pattern are resolved one at a time (blue constraint nodes). The first chain, for example, assigns the `name` of the `Professor` to the symbol *`$n`* and then it verifies that the `age` of the individual that is being processed is higher than 50. Conversely, if the `age` is less than or equals to 50, the whole individual does not qualify as a matching instance of the pattern and it is discarded. Only the objects that manage to cross all the nodes of the chain fully satisfy the corresponding pattern.

These objects are stored in an α memory (gray nodes) to be ready for subsequent efficient propagation in other parts of the RETE network. Notice that there are generally several α memories in a RETE network and it is possible that the same individual satisfies several patterns and therefore appears in several memories. In a moment we

will see that β memories behave in the same way. This is the reason why the WM which is a collection of all these memories is said to be *distributed* and *redundant*.

With respect to the second pattern, we have a node which combines an equality constraint for the field `subject` to the value `"AI"` and its assignment to the symbol `$s`. The following node is more complicated to handle because it refers to a symbol that appears in another pattern which may not be available yet (orange constraint nodes). It is evident that the evaluation of such constraint must be deferred to a later time when the foreign value will be available.

At the moment, a dummy node which judges as valid every instance that passes by, is inserted in its place. The proper constraint node is displaced after the join node that merges the chains providing the objects with the fields that act as terms of comparison for the constraint. The individuals who do not meet this requirement are discarded at this point.

Merge nodes and β memories

As individuals accumulate in α memories, the β node (green node) use them to generate all the possible pairs [6] and propagates them (gray nodes). If some constraint has been postponed, as in our case, it is evaluated now. After the deferred nodes there is a β memory which contains all the pairs of individuals meeting the cross-referencing constraint. This memory improves the efficiency of the system making the results of the matching process immediately available to anything that wishes to use this data in other parts of the RETE network. It works just like the α memories that we have already seen: the only difference is that they contains tuples of objects rather than single objects.

Entry-points and activation nodes

At the end of each (possibly compound) chain of nodes that correspond to a whole premise of a rule there is a leaf node (purple nodes). So, as long as instances are introduced into the WM, they are injected into the RETE network via the entry-point. Notice that it is possible to define additional entry-points and that the patterns may hook specific initial nodes with the construct `from entry-point "name"` rather than the default one. Therefore they can be used to partition the WM and allow even more refined reasoning where it is appropriate to keep separated different sources of information. Regardless of the entry-point, these instances flow through the network possibly combined in tuples until they are filtered by a node or they reach an activation node. Whenever a tuple hits a leaf node, an activation of the rule that is associated to the leaf node on this tuple is generated and appended to the AGENDA. Notice that the rules also allow to reason in the so-called *negative logic*, which means that patterns and constraints can be negated to let the engine "recognise the absence"

6 A merge node computes the power set of the two incoming sets of instances. These nodes, therefore, can potentially produce a large number of tuples that consume resources; fortunately cross-references constraints and hashing techniques keep the memory consumption low.

(instead of the presence) of objects of a given kind in the WM. It means that negated patterns and constraints are positively verified when no object matching them is asserted into the WM or when the last matching object is retracted from the WM so that its deletion "propagates" through the RETE [7].

A.3 PROCESSING TEMPORAL INFORMATION

Complex events processing

In 2002, Luckham has proposed an "*emerging technology for building and managing information systems*" [8] whose goal is to process all the multiple events that are occurring on a target domain in order to identify meaningful events within this "event cloud" and take proper measures. This discipline, which is called CEP [109, 110], uses techniques like the detection of complex patterns of many events, the event correlation and abstraction, the event categorisation thanks to relationships between events such as causality, membership, and timing to deal with large amount of information. It has been successfully applied to Business Activity Monitoring, Business Process Management, Enterprise Application Integration, Event-Driven Architectures, Network and business level Security and Real time conformance to regulations and policies.

Drools Fusion

As the reader may guess, although CEP cannot be reduced (accidentally or intentionally) to a RBS augmented with the support to EVENT-DRIVEN ARCHITECTURE (EDA), PRS natively supporting the idea concept of event do provide several CEP features. In this regard JBoss offers *Drools Fusion*, a module for *behavioural modelling* that enables event processing capabilities within *Dools Expert*'s rule engine. Such result is possible because events are now understood and handled as first class citizens for the platform, features were added to identify sets of interesting events out of the stream of events, to detect relevant relationships among events and, of course, take appropriate actions accordingly.

Events

So, what exactly is an event? Events are special entities that represent some significant change in the state of the application domain that are associated with a specific temporal context. They have several unique and distinguishing characteristics, like having a timestamp (and possibly a duration), being usually immutable, having strong temporal constraints and relationships. They can be used as well to express subpatterns in the premise of the rules by predicating on both the presence or the absence of matching individuals.

Events can be defined from generic facts by properly decorating them with the "*event*" role. If we consider the legacy POJO in List-

7 Notice that the propagation of a deletion through a given RETE works much like the migration of electron holes in semiconductors.

8 http://complexevents.com/event-processing/

```java
package example;

import java.util.Date;

/**
 * Payload object
 */
public class MobileCall {
  private String dialler;
  private String diallee;
  private Date startTime;
  private long durationTime;
  ...
}
```

Listing A.4: Legacy payload objects.

```
import example.MobileCall;
import java.util.Date;

declare MobileCall
  @role( event )
end
```

Listing A.5: Legacy objects as events.

ing A.4 that abstracts the features of a phone call, we can automatically turn it into an event by annotating an homonymic declaration with `@role(event)` (see Listing A.5). Notice that any declaration of facts is implicitly (if not explicitly) annotated with `@role(fact)`.

Binding events with times

From that point on, the PRS treats any `MobileCall` instance as an *event* by autonomously setting its timestamp to the time of its assertion and (possibly) updating its duration. These time values are stored in a couple of fields of the system class which decorates the payload instances to declare events and therefore they do not coincide with the fields `startTime` and `durationTime` that we have introduced above. Timestamp and duration, in fact, are transparently managed by the engine with respect to the user. Notice that, by default, the duration of a Drools Fusion *event* is set to `0`: such events are said to be *point-in-time events*; events with non-empty duration are called instead *interval events*.

Accessing time values

Sometimes, however, it is appropriate to directly access these values: the engine offers primitives to bind them with some explicit fields of the POJO by further annotating the declaration with labels like `@timestamp(...)` and `@duration(...)`, as in Listing A.6 on the next page. Unfortunately, these annotations override the default be-

```
1 import example.MobileCall;
2 import java.util.Date;
3
4 declare MobileCall
5  @role( event )
6  @timestamp( startTime )
7  @duration( durationTime )
8 end
```

Listing A.6: Explicating the events' temporal data.

```
1 declare MobileText
2  @role( event )
3  @timestamp( sent )
4  @expires( 5m )
5  dialler: String
6  diallee: String
7  sentTime: Date = new java.util.Date()
8  text: String
9 end
```

Listing A.7: Declaring native events.

haviour of the engine and consequently the times must be manually managed by the user. When new instances of `MobileCall` are generated, for instance, the user must specify the values for all the fields of the object thus including `startTime` and `durationTime`, whose default values are overwritten. The problem of their non-automatic assignment can be partially mitigated by the forcing an additional explicit assignment of the current time value, as in Listing A.7 (line 7). Notice that this time we have declared the POJO `MobileText` directly within the rule engine environment, therefore with no need to import it from the external *Java* world. The Listing above (line 4) also introduces another feature of EDAs: just like planes, events can have an Estimated Time of Arrival (ETA) which defines a sort of deadline after which it is assumed that their journey through the KB is over, having delivered their information content. So, according to this metaphor, events are no more needed after reaching their destination: the engine knows how to identify and dispose these events from memory, freeing resources and scaling well on growing volumes. With reference to the above Listing, for example, when mobile texts are sent, the infrastructure that takes over them tries to deliver them immediately or, if it is not possible, to do so within few minutes – i. e. 5 minutes. Thus, after 5 minutes, it is assumed that the messages are sent or that their missed delivery is notified. In either cases, the events representing an attempt to send a test are no more needed and the annotation

Disposing dispatched events

```
1  rule "Counts chars sent by a mobile in last few texts"
2  when
3    accumulate(
4      MobileText( dialler == ID, $t: text )
5        over window:length( 10 ),
6      $c: sum($t.length)
7    )
8  then
9    System.out.println("Chars: " + $c);
10 end
11
12 rule "Counts the texts sent by a mobile in the last day"
13 when
14   accumulate(
15     $t: MobileText( dialler == ID )
16       over window:time( 1d ),
17     $c: count($t)
18   )
19 then
20   System.out.println("Texts: " + $c);
21 end
```

Listing A.8: Example of rules with sliding windows.

`@expires(...)` tells the engine that it is safe to mark them as disposable for the next GARBAGE COLLECTOR (GC) execution.

Sliding windows

This behaviour also applies in the context of sliding windows. The sliding windows mechanism is indeed a way to restrict the events to consider to the last few of them that are falling inside a moving time window which is anchored to the current time instant. There are basically two kind of sliding windows: those based on a time interval and those maintaining a queue of fixed length. The Listing A.8, for instance, contains an example of both kinds of sliding windows. The first rules accumulates the number of characters (line 6) of the last ten texts (line 5) sent by a given mobile (line 4): the keywords **`over window:length`** introduce the number of the last few events that we want to consider. Similarly, the second rule counts the number of texts (line 17) over the last day (line 16) sent by the same mobile (line 15). Here the keywords **`over window:time`** introduce the width of the temporal interval that we want to consider, or how far back in time with respect to the current time we want to look for events. Notice that the temporal horizon imposed the statement in line 16 conflicts with the much shorter ETA of five minutes suggested earlier. The temporal extension of the rule engine has a mechanism that resolves such conflicting statements which promotes the higher value as the proper threshold for the disposability of events. The engine

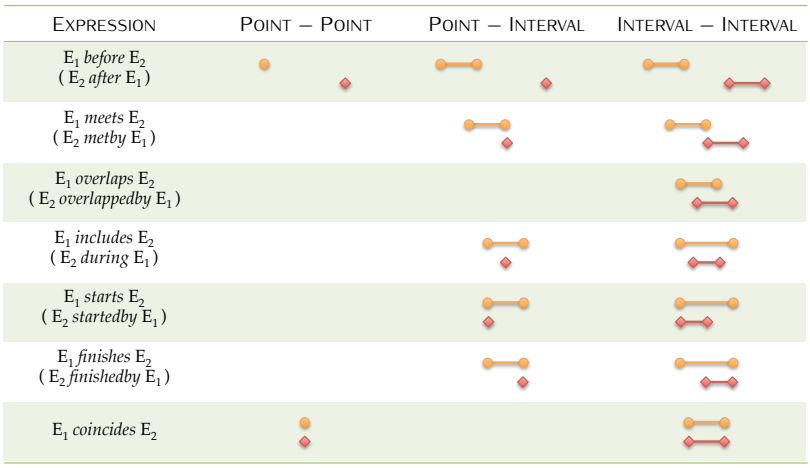


Figure A.4: Allen's temporal operators among couples of events.

```
1  rule "Counts texts received by a mobile during calls"
2  when
3    accumulate(
4      $t: MobileText( diallee == ID )
5      exists MobileCall( dialler == ID or diallee == ID,
6        this includes $t),
7      $c: count($t)
8    )
9  then
10   System.out.println("Texts during calls: " + $c);
11 end
```

Listing A.9: Example of rules with simple temporal operators.

manages threshold values for each kind of event that is defined in the current KB. Notice that, if the second rule above is removed from the PM for some reason, then the threshold of 5 minutes becomes valid again.

Temporal constructs

Usually there are strong relations and constraints intervening among events so several temporal operators have been introduced to take care of them. These operators capture the relationship between couples of events – be they point-in-time events or interval events – such as precedence, concurrence or overlapping. Such temporal constructs have been defined for the first time in 1981 by Allen [7, 9]. The table in Figure A.4 includes all 13 kinds of relationship between events and provides an intuitive graphical description for each of them. Our reference rule engine supports all of them and also their logical complement (negation) [26, 197]. It is possible to see temporal operators in action in Listing A.9 where the number of texts received by a given mobile (line 4) during incoming or outgoing calls (line 5 and 6) is counted (line 7).

```
rule "Call-back protocol"
when
  $c: MobileCall( dialler == ID, $d: diallee,
      durationTime <= 1s )
  not MobileCall( dialler == $d, dialler == ID,
    this after[ 0, 30s ] $c )
then
  System.out.println(ID + " one-bell " + $d + " again...
      ");
end
```

Listing A.10: Example of rules with parametric temporal operators in negative logic.

Negative logic

In addition to the default syntax, all the temporal operators allow to include specific times to further refine the temporal relationship between events. In Listing A.10, for instance, `after[0,30s] $c` (or simply `after[30s] $c`) means that we expect the current event to happen after `$c`, but not later than 30 seconds. Notice that the rule engine also allows to reason about events in *negative logic*, which means that it predicates over their absence rather than their presence. The following rule, in particular, simulates a simple protocol that is often adopted by people when they are low on credit: they telephone someone and let the mobile ring once before hanging up. The desired outcome is to be called back within a few seconds, otherwise repeating the process. So if after 30 seconds no call is incoming from the diallee, the rule suggest to ring her again.

Cloud mode vs. stream mode

All the features described so far are included in any basic download of Drools, but they need to be enabled. This step is required because the rule engine can operate in two distinct modes: the *cloud* and the *stream modes*. The default mode is the *cloud mode* which is exactly the way in which pure forward-chaining rules engine work.

In this mode, events are still events but they are treated as simple facts because there is no notion of the flow of time: the idea of ETA is meaningless because there is no concept of "now" to match against. The *stream mode* has indeed the notion of time and allows to process events in the way that we have described in this section. Actions involving events are synchronised through a session clock that settles all the time conflicts.

Real-time clock vs. pseudo clock

Two implementations of the session clock are supported: *real-time clock* and *pseudo clock*: the former relies on the system clock, the latter requires the user to manually advance the time. Roughly speaking, the *pseudo clock* is generally used to test temporal rules since the time can be explicitly manipulated, the *real-time clock* instead is used during execution to manage a domain. The following Listing A.11 on the next page shows how to configure an instance of the engine to work

```
import org.drools.*;
import org.drools.builder.*;
import org.drools.conf.*;
import org.drools.io.*;
import org.drools.logger.*;
import org.drools.runtime.*;

/**
 * Sample class to launch the engine
 * in STREAM mode with REAL-TIME/PSEUDO clock.
 */
public class DroolsTest {

 public static final void main(String[] args) {
  try {
   // Load the knowledge from the Domain.drl
   Knowledgebld bld =
    KnowledgebldFactory.newKnowledgebld();
   bld.add("Domain.drl", ResourceType.DRL);
   KnowledgebldErrors errors = bld.getErrors();
   if (errors.size() > 0) {
    for (KnowledgebldError error : errors)
     System.err.println(error);
    throw new IllegalArgumentException("Errors in KB");
   }
   // Configure a KB in STREAM mode
   KnowledgeBaseConfiguration kCfg =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
   kCfg.setOption(EventProcessingOption.STREAM);
   KnowledgeBase kb =
KnowledgeBaseFactory.newKnowledgeBase(kCfg);
   kb.addKnowledgePackages(bld.getKnowledgePackages());
   // Start a session with REAL-TIME/PSEUDO clock
   KnowledgeSessionConfiguration sCfg =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
   sCfg.setOption(ClockTypeOption.get("realtime"));
   /* sCfg.setOption(ClockTypeOption.get("pseudo")); */
   StatefulKnowledgeSession session =
    kb.newStatefulKnowledgeSession(sCfg, null);
   // Operates the domain
   session.fireAllRules();
  } catch (Throwable t)
   t.printStackTrace();
 }
}
```

Listing A.11: Configuring the PRODUCTION RULES SYSTEM to perform COMPLEX EVENT PROCESSING: enabling STREAM mode, REAL-TIME and PSEUDO clock.

in *stream mode* (lines 27–32) with the *real-time clock* (lines 34–39) or the pseudo clock (see the commented line 37) with respect to a default instantiation of the engine. More details are available in the official online documentation [9].

A.4 PROCESSING UNCERTAINTY AND IMPRECISION

During the last decades uncertainty – like imprecision and vagueness – has gained considerable attention. This is principally due to that fact that some of the information about the world is inherently imprecise or vague like the concept of a "tall" person or a "partly cloudy" sky. In many applications like *information retrieval*, *multimedia information analysis*, in effect, most facts are not simply true or false, but rather a matter of similarities or ranking degrees.

Although some interesting solutions have been proposed in the web context by composing ontology and rule-based languages, there are still many cases where these languages fail to represent the knowledge of our world. In particular these languages are not able to face the uncertainty introduced in real application knowledge and information (like multimedia processing [99, 171], information retrieval [100], pattern recognition [85], decision making [200] and many more).

The need for covering uncertainty has been stressed out in literature many times [87, 121, 170]. It has been pointed out that dealing with such information would improve many domains like portals [199], multimedia application in the semantic web [24, 171], e-commerce applications [3], situation awareness and information fusion [121], rule languages [87, 170], medicine and diagnosis [72], geospatial applications [37] and many more.

In *Drools Chance*, facts about the world can include a specification of a "degree" (a truth value between 0 and 1) of confidence with which one can assert that a combination of facts. In the following paragraphs a few examples of use will be presented to show the capabilities of the framework.

The Listing A.12 on the following page, for example, shows how to use uncertainty to model a distribution of probability. The example first declares a person class (line 1) which is characterised by a field name (line 2) and a field age (line 3). The age field is decorated with an `@Imperfect(...)` annotation which is used to pass "imprecise" directives to the rule engine. In particular, this annotation means that a distribution of probability over discrete values modelled with a simple degree will be associated to the value of this field (line 3).

The following rule (line 6) has an empty premise which makes the rule to trigger at the beginning of each session activation. The consequence of the execution of the rule is the creation of an instance

9 http://www.jboss.org/drools/documentation

```
1  declare Person
2    name: String
3    age: int @Imperfect( kind=ImpKind.PROBABILITY,
               type=ImpType.DISCRETE, degree=DegreeType.
     SIMPLE )
4  end
5
6  rule "Init"
7  when
8  then
9    Person john = new Person();
10   john.setName("John");
11   john.setAge("15/0.3, 22/0.2, 34/0.4, 48/0.1");
12   insert(john);
13 end
14
15 rule "Evaluation"
16 when
17   $b: Person( age == 34, age ~== 34 )
18 then
19   Degree x = chance.getDegree();
20   System.out.println(x); // 0.4
21 end
22
23 rule "Custom Evaluation"
24 when
25   $b : Person( age ~!= [family=MvlFamilies.GODEL] 34 )
26 then
27   Degree x = chance.getDegree();
28   System.out.println(x); // 0.3
29 end
30
31 query person( String $name, int $age, Person $x )
32     $x := Person( $name := name; age ~== $age )
33 end
```

Listing A.12: Using degrees of truth to define distributions of probability.

`john` of the `Person` class with name John and age which is 15 (3 times out of 10), 22 (2 times out of 10), 34 (4 times out of 10) and 48 (1 time out of 10). It could be used, for example, to model the distribution of ages of the persons named John in a given environment.

The following rule (line 19) shows how the patterns involving such imperfect field are evaluated. Line 17 contains three atomic patterns: the first verifies if the instance under test is a `Person` and it is evalu-

```
1  /*  SLIM      FAT
2   1.0 |.       .|
3       | \     / |
4       |   \ /   |
5       |   / \   |
6       | /     \ |
7   0.0 |.       .|
8     --+---------+-->
9       0        100   */
10 declare enum Weight
11   @LinguisticPartition
12   SLIM( "slim", new FuzzyTriangle(-0.01, 0, 100) ),
13   FAT( "fat", new FuzzyTriangle(0, 100, 100.01) );
14   label: String
15   set: FuzzySet
16 end
```

Listing A.13: Enumerations and linguistic partitions.

ated to TRUE with a degree of 1.0, the second is a "crisp" equivalence constraint which is evaluated to 1.0 as well because the most probable option for the age of John is 34 and finally there is an "imprecise" equivalent constraint that is evaluated to 0.4, the value associated to 34. These degrees are automatically combined by the engine to compute the overall degree of the rule: the default combination strategy consists in multiplying the single contributions so the rule prints the value 0.4 (lines 19 and 20) as the result of its consequence.

The last rule (line 23) demonstrates how to customise patterns to the level of the single constraint. In this case we have just two evaluations: the first again verifies that the instance being tested is a `Person` and the second that ensures that the age of that `Person` is "imprecisely" different from 34 (line 25). The first constraint is again evaluated to 1.0 while the second is evaluated to 0.3 as the `~!=` is further customised by an annotation that enables the Gödel model of FUZZY LOGIC (FL) for the evaluation (line 25). The Gödel model, in fact, takes the most probable option among the remaining ones: excluding 34 which is the term of comparison, the second most probable option is 15 and it probability 0.3 is actually returned. These degrees are combined again with the default strategy and a value of 0.3 is printed (line 28).

The final query (line 31) is included only to show that imprecise evaluators may be used within queries as well.

Another typical usage for this module is to define Fuzzy linguistic labels to be used over a given multivalued crisp domain. The Listing A.13, for example, introduces enumerations and shows how to customise them to handle fuzzy partition sets. The declaration of an

```
1  declare PersonA
2    name: String
3    age: int @Imperfect( kind=ImpKind.PROBABILITY,
               type=ImpType.DISCRETE, degree=DegreeType.
     SIMPLE )
4    body: Weight @Imperfect( kind=ImpKind.FUZZINESS,
               type=ImpType.LINGUISTIC, degree=DegreeType.
     SIMPLE,           support="weight" )
5    weight: Double
6  end
7
8  declare PersonB
9    name: String
10   age: int @Imperfect( kind=ImpKind.PROBABILITY,
               type=ImpType.DISCRETE, degree=DegreeType.
     SIMPLE )
11   body: Weight @Imperfect( kind=ImpKind.FUZZINESS,
               type=ImpType.LINGUISTIC, degree=DegreeType.
     SIMPLE,           support="weight" )
12   weight: Integer @Imperfect( kind=ImpKind.PROBABILITY,
               type=ImpType.DISCRETE, degree=DegreeType.
     SIMPLE )
13 end
```

Listing A.14: Using linguistic partitions.

enumeration is introduced by the `enum` keyword (line 10) and then requires the sequence of possible options. In order to turn the enumeration into a linguistic fuzzy partition, the `@LinguisticPartition` annotation is required (line 11). Each item of the enumeration always needs two additional parameters: an identifying label and a fuzzy function. The labels are stored in field of the enumeration called `label` (line 14). The fuzzy functions are accumulated instead into a fuzzy set (line 15).

The example presented in Listing A.13, in particular, defines a linguistic partition for the `Weight` domain: the labels are `SLIM` and `FAT` and they are described by the two fuzzy triangle function at the end of lines 12 and 13 and graphically represented by the sketch at the beginning of the Listing (lines 1–9). Both the triangles insist on the same range $0 - 100$ however their domain is slightly larger to avoid degenerate values (0 for `SLIM` and 100 for `FAT`).

Listing A.14 shows how to use this linguistic partition. It declares two kinds of persons – `PersonA` and `PersonB` – that are slightly more complex than the `Person` seen before. Both classes have a `name` (a string, lines 2 and 9), an `age` (a distribution of probability, lines 3 and 10), a `body` (an assignment for a linguistic partition supporting a

```
1  rule "Init"
2  when
3  then
4    PersonA john = new PersonA();
5    john.setName("John");
6    john.setAge("15/0.3, 22/0.2, 34/0.4, 48/0.1");
7    john.setWeight(70.0);
8    insert(john);
9
10   PersonB mark = new PersonB();
11   mark.setName("Mark");
12   mark.setAge("15/0.3, 22/0.2, 34/0.4, 48/0.1");
13   mark.setWeight("60/0.05,80/0.05,100/0.9");
14   insert(mark);
15 end
16
17 rule "Evaluation"
18 when
19   $x: PersonA( $w: weight ~is [ label=is ] Weight.FAT )
20         @Imperfect( label=ptrnAnd )
21   and @Imperfect( label=outerAnd )
22   $y: PersonB( $a : age ~!= [ label=notForty ] 48
23         && @Imperfect( label=innerAnd )
24         weight ~> [ label=betaJoin ] $w )
25 then
26  // 0.7
27  System.out.println(chance.getDegree("is").value);
28  // 0.9
29  System.out.println(chance.getDegree("notForty").value);
30  // 0.95
31  System.out.println(chance.getDegree("betaJoin").value);
32  // 0.7
33  System.out.println(chance.getDegree("ptrnAnd").value);
34  // 0.55
35  System.out.println(chance.getDegree("outerAnd").value);
36  // 0.85
37  System.out.println(chance.getDegree("innerAnd").value);
38  // 1.0
39  System.out.println(chance.getDegree("PersonA").value);
40 end
```

Listing A.15: Complex evaluations involving several kinds of degrees of truth.

`weight` field, lines 4 and 11) and a `weight` (line 5 and 12). In the first case, the weight is a simple Double value while it is a distribution of

```
import org.drools.*;
import org.drools.bld.*;
import org.drools.chance.*;
import org.drools.io.*;
import org.drools.logger.*;
import org.drools.runtime.*;

/**
 * Sample class to launch the engine
 * with imprecise and vague extensions.
 */
public class DroolsTest {

 public static final void main(String[] args) {
  try {
   // Initialise the Chance extension
   Chance.initialize();
   // Load the knowledge from the Domain.drl
   Knowledgebld bld =
    KnowledgebldFactory.newKnowledgebld(
     Chance.getChanceKbldConfiguration());
   bld.add("Domain.drl", ResourceType.DRL);
   KnowledgebldErrors errors = bld.getErrors();
   if (errors.size() > 0) {
    for (KnowledgebldError error : errors)
     System.err.println(error);
    throw new IllegalArgumentException("Could not parse
    KB");
   }
   // Configure a KB
   KnowledgeBase kb =
    KnowledgeBaseFactory.newKnowledgeBase(
     Chance.getChanceKnowledgeBaseConfiguration());
   kb.addKnowledgePackages(bld.getKnowledgePackages());
   // Start a working session
   StatefulKnowledgeSession session =
    kb.newStatefulKnowledgeSession();
   // Operates the domain
   session.fireAllRules();
  } catch (Throwable t)
   t.printStackTrace();
 }
}
```

Listing A.16: Configuring the PRS to handle degrees of truth.

probability in the second case. Notice that the `body` field is annotated again with an `@Imperfect(...)` directive which tells the engine that it involves a fuzzy definition over a linguistic partition using a simple degree and supporting a `weight` field, of course.

The Listing A.15 on page 175 shows an example of use for these classes. The first rule simply creates a `john` instance of `PersonA` and a `mark` instance of `PersonB`. They share the same distribution of probability for their `age` that was discussed in Listing A.12. `john`, however, has a `weight` of 70.0 and `mark` a weight that is 60 in the 5% of the times, 80 in another 5% of the times and 100 in the 90% of the times. The second rule named `"Evaluation"` exhibits a complex pattern in which several annotation are included to extract the temporary degrees that are specific of the different steps of the evaluation. These values are printed on the standard output as a result of the execution of the rule. Notice that in the case of the first instance, the fuzzy set is updated accordingly to the weight set and vice versa. In the second case, the process is more complicated because the supporting field is a distribution of probability.

The valid values for the `@Imperfect` annotations are as follows. The *kind* may be *probability* or *fuzziness*. The *type* may be *discrete*, *dirichlet*, *linguistic* or *basic*. The *degree* may be *simple* or *double*. Finally the valid *families* are *Gödel*, *Łukasiewicz*, *product* or *sum*, but additional custom families may be defined as well. Finally, consider the following Listing A.16 on the facing page where we show how to initialise a session of the PRS that supports imprecise reasoning. It is very similar to the equivalent Listing for the temporal reasoning that we have seen before (Listing A.11 on page 170). The most peculiar steps are on line 17 – where the imprecise extension is initialised, on line 21 – where a proper configuration for the knowledge builder is passed and on line 33 – where a proper configuration for the knowledge base is propagated.

A Fuzzy control example

The Listings in this section contain a full example taken from the literature about a room and with a fan to regulate its temperature. In the first Listing A.17 on the following page, we declare two linguistic partitions for the air temperature and the fan speed of the room. The temperature may be `COLD`, `NICE` or `HOT`, while the fan speed is `SLOW`, `MEDIUM` or `FAST`.

In Listing A.18 on the next page we introduce the declarations for `Rooms` (which uses the `AirTemperature` to support a temperature `temp` field) and `Fans` (that uses the `FanSpeed` to support the `rpm` value of the fan). Some additional queries are provided as well to optimise the retrieval of information about the room and the fan. The Listing A.19 on page 179 also includes the rule that sets up the domain model by

```
declare enum AirTemperature
@LinguisticPartition
  COLD( "cold", new FuzzyTriangle(-0.01, 0, 20) ),
  NICE( "nice", new FuzzyTriangle(0, 20, 35) ),
  HOT( "hot", new FuzzyTrapez(20, 35, 100, 100.01) );
  label : String
  set   : FuzzySet
end

declare enum FanSpeed
@LinguisticPartition
  SLOW( "slow", new FuzzyTriangle(-0.01, 0, 500) ),
  MEDIUM( "medium", new FuzzyTriangle(250, 500, 750) ),
  FAST( "fast", new FuzzyTriangle(500, 1000, 1000.01) );
  label : String
  set   : FuzzySet
end
```

Listing A.17: Linguistic partitions for the air temperature and the fan speed.

```
declare Room
  id: String
  temperature: AirTemperature @Imperfect( kind=ImpKind.
    FUZZINESS, type=ImpType.LINGUISTIC, degree=
    DegreeType.SIMPLE, support="temp" )
  temp: Double
end

declare Fan
  id: String
  speed: FanSpeed @Imperfect( kind=ImpKind.FUZZINESS,
    type=ImpType.LINGUISTIC, degree=DegreeType.SIMPLE,
    support="rpm" )
  rpm: Double
  roomId: String
end
```

Listing A.18: Declarations for defining the "imprecise" domain.

introducing a `"roomA"` with a temperature of 3.0 degrees and a `"fanA"` initially off.

The next three Listings of this example (Listing A.20 on page 180, Listing A.21 on page 180 and Listing A.22 on page 181) include the rules to compute the contribution to the fan speed produced by the temperature of the room. The current temperature is evaluated as `HOT`,

```
30 query fan( String $roomId, Fan $fan )
31   $fan := Fan( $roomId := roomId )
32 end
33
34 query room( String $roomId, Room $room )
35   $room := Room( $roomId := id )
36 end
37
38 rule "Init"
39 when
40 then
41   Room roomA = new Room();
42   roomA.setId("roomA");
43   roomA.setTemp(3.0);
44   insert(roomA);
45
46   Fan fanA = new Fan();
47   fanA.setId("fanA");
48   fanA.setRoomId(roomA.getId());
49   fanA.setSpeedValue(null);
50   insert(fanA);
51 end
```

Listing A.19: Initialising the "imprecise" domain.

NICE and COLD by fuzzification and a suggested speed for the fan is computed. In Listing A.20 on the following page, the contribution to FAST for the fan speed is determined from the degree of HOTness of the room temperature. Similarly, in Listing A.21 on the next page, the contribution to MEDIUM speed is computed from the degree to which the room temperature qualifies as NICE. In Listing A.22 on page 181, instead, the contribution to SLOW for the fan speed is determined from the degree of COLDness of the room temperature.

Finally, in the last Listing A.23 on page 182 of the example, we introduce two rules: the first is a simple rule with low priority (line 107) to print out some statistics and the second is the rule that computes the values to control the fan speed. This second rule is triggered every 1,500 milliseconds (line 123) and its effects are not propagated in loop (line 124). It uses some peculiar physical model to determine the difference in temperature due to the speed of fan in the last time interval. This value is computed by combining the contributes provided by the three rules above. Finally the speed fan is reset (line 138) to compute the contribution for the next interval and the temperature of the room is defuzzified and updated (line 142).

```
rule "HOT -> FAST"
when
  $b: Room( $id: id,
            temp ~is [label=hot] AirTemperature.HOT )
  ?fan( $id, $fan; )
then
  System.out.println("Temp is HOT to degree " +
    chance.getDegree("hot"));

  Degree act = chance.degree;
  System.out.println(
    "Setting speed to FAST, with degree " + act);

  System.out.println("\t >> " + $fan.speed);
  modify($fan) {
    updateSpeedValue(FanSpeed.FAST, act);
  }
end
```

Listing A.20: Computing the contribution to FAST fan speed by HOT room temperature.

```
rule "NICE -> MEDIUM"
when
  $b: Room( $id: id,
            temp ~is [label=nice] AirTemperature.NICE )
  ?fan( $id, $fan; )
then
  System.out.println("Temp is NICE to degree " +
    chance.getDegree("nice"));

  Degree act = chance.degree;
  System.out.println(
    "Setting speed to MEDIUM, with degree " + act);

  System.out.println( "\t >> " + $fan.speed );
  modify ($fan) {
    updateSpeedValue(FanSpeed.MEDIUM, act);
  }
end
```

Listing A.21: Computing the contribution to MEDIUM fan speed by NICE room temperature.

A.5 SUMMARY

In this Appendix we have presented the principle of operating of a typical PRS and we have provided several examples to clarify its us-

```
88  rule "COLD -> SLOW"
89  when
90    $b: Room( $id : id,
91              temp ~is [label=cold] AirTemperature.COLD )
92    ?fan( $id, $fan ; )
93  then
94    System.out.println("Temp is COLD to degree " +
95      chance.getDegree( "cold" ));
96
97    Degree act = chance.degree;
98    System.out.println(
99      "Setting speed to SLOW, with degree " + act);
100
101   System.out.println("\t >> " + $fan.speed);
102   modify($fan) {
103     updateSpeedValue(FanSpeed.SLOW, act);
104   }
105 end
```

Listing A.22: Computing the contribution to SLOW fan speed by COLD room temperature.

age. We have also introduced the most important aspects of *Drools* and some of its extensions, namely the extension for processing temporal information and imprecise and vague knowledge. The content of this Chapter does not contain any original scientific contribution but the examples are original for the most part. Ultimately the purpose of this chapter is to provide some insight that are useful to understand the theoretical and practical aspects behind the original contributions that are included in this dissertation.

```
106 rule "Status"
107 salience -10
108 when
109   $f: Fan( $speed: speed, $rpm: rpm, $roomId: roomId )
110   ?room( $roomId, $room; )
111 then
112   System.out.println("Current room temp is " +
113     $room.getTemperature());
114   System.out.println("\t Matching C is "  +
115     $room.getTemp());
116   System.out.println("Fan speed (fuzzy) is " +
117     $f.getSpeed());
118   System.out.println("\t Matching rpm is " +
119     $f.getRpm());
120 end
121
122 rule "Apply fan"
123 timer ( intv: 1500 )
124 no-loop
125 when
126   $f: Fan( $speed: speed, $rpm: rpm, $roomId: roomId )
127   ?room( $roomId, $room ; )
128 then
129   System.out.println(
130     "Guess the delta T given the fan speed");
131   System.out.println(
132     "by applying some 'physical' model.");
133   double deltaT = (500 - $rpm)/100.0;
134   double temp = $room.getTemp() + deltaT;
135   System.out.println("Temp changed by " + deltaT);
136   // Clear the fan speed for a new inference
137   modify($f) {
138     setSpeedValue(null);
139   }
140   // "Measure" the new temp
141   modify($room) {
142     setTemp(temp);
143   }
144   System.out.println(
145     "After changing, the current room temp is " +
146  $room.getTemperature());
147   System.out.println("\t Matching C is " +
148     $room.getTemp());
149 end
```

Listing A.23: Enforcing the control of the fan speed according to the room temperature.

B | PROBABILITY

«The probability that we may fail in the struggle ought not to deter us from the support of a cause we believe to be just.»

— Abraham Lincoln
16th President of the United States of America,
1809-1865

In this Appendix we make a small digression in the field of probabilistic reasoning. In particular we briefly contextualise Probabilistic Inductive Logic Programming (PILP) and we describe Logic Programs with Annotated Disjunctions (LPADs), one of the formalisms introduced in this area which is appreciated for its readability and expressiveness. Then we describe some preliminary original approaches and optimisations to solve this class of problems by means of production rules. As we will show in the following sections, some of them can only tackle sub-classes of the original problem and only one is general enough to solve any kind of problem. These sub-classes consist in graphs with specific shapes whose characteristics will be precisely described in following sections. This approach, however, does not take advantage of the very nature of Production Rule System (PRS) that we used to implement these algorithms. This is the reason why we still consider this contribution as work-in-progress and we have decided to provide it as an Appendix rather than a proper Chapter of the dissertation.

The software that we have described in this thesis would certainly benefit from probabilistic knowledge on the working domain, however this kind of information is not really essential to handle processes' deviations. In any case, the goal of the work described in this Appendix is ultimately to deploy a stand alone module to perform probabilistic reasoning over graph problems. However, being an independent component, it could be plugged in the above software to enrich it. Any knowledge base on a specific domain, in fact, implicitly define a network of concepts and relationships that the module could process to determine statistical information. Such procedure could be enclosed in a *Drools*' Custom Operator (CO) to provide an high level mechanism to perform probabilistic reasoning.

B.1 PROBABILISTIC INDUCTIVE LOGIC PROGRAMMING

Probability is a quite general term which is used in several contexts. It is often involved in application where it is used to model confidence, relevance, likelihood and other similar or complementary concepts. The goal of this work is to provide some line guides to favour the handling of probability in general terms within forward-chaining tools like a PRS.

Related works

Logic and probability have been widely investigated within ARTIFICIAL INTELLIGENCE (AI) [34, 66, 161]. In recent years, due to the advances in STATISTICAL RELATIONAL LEARNING (SRL) [68] and PILP [48], this topic has been the subject of a renewed interest. Several formalisms combining relational and statistical aspects, such as PROBABILISTIC LOGIC PROGRAMS (PLPs) [45], INDEPENDENT CHOICE LOGIC (ICL) [135], PRISM [159], PD [65], BAYESIAN LOGIC PROGRAMS (BLPs) [86], LPADs [187, 188], PROBLOG [49] and P-LOG [21] have been proposed.

Complexity of the task

LPADs are a particularly interesting formalism because they are rather expressive but also easily readable by humans: their appealing is due to the easiness with whom they may express at the same time cause-effect relationships among events, possible effects of a single cause and the combined contribution of more causes to the same effect. Notice that it has been shown that the majority of the above formalisms are semantically equivalent [47], thus the greater expressiveness depends on the syntax which is simply an extension of LOGIC PROGRAMMINGS (LPs). Unfortunately, the class of problems that these probabilistic formalisms typically cope with is very hard. Its complexity class is #P [89], defined as the set of the counting problems associated with the decision problems in NP. Clearly, #P problems are at least as hard as the corresponding NP problem, but usually they are more difficult [181]. This is the reason why, in the last few years, several algorithms for the "approximate" inference have been proposed [32, 88, 142, 143] where accuracy is traded for efficiency. The idea behind this work is that we believe it is possible to achieve better results by adopting forward-chaining tools – rather than backward--chaining tools like PROLOG, on which most of the cited formalisms are relying. In a forward-chaining system, in fact, all the probability contributions for a given goal could be determined in parallel and combined immediately instead of finding them one by one as with backward-chaining programs and delegating their subsequent combination to an appropriate external tool.

B.2 LOGIC PROGRAMS WITH ANNOTATED DISJUNCTION

A reference formalism

LPADs [187, 188] are a probabilistic extension of LP [107] based on *disjunctive* logic programming. Each LPAD consists of a finite set of annotated disjunctive clauses where the head is a set of mutually disjunctive logical atoms annotated with a probability value in the interval $[0, 1]$ and the body a classic (possibly empty) disjunction of conjunctions of logical literals. Notice that the probability values in the head of a disjunctive clause sum up to 1: if the result is lower than 1, than an additional logical atom is missing with the residual probability, which means that none of the explicated options is chosen. The choice of a logical atom in the head of a disjunctive clause is subjected to the probability with whom such atom is annotated. The grounding of an LPAD is the union of all the possible ground instantiations of the its clauses. Each clause in the grounding of an LPAD represents a probabilistic choice between the non-disjunctive clauses that are obtained by selecting a single atom from its head.

If we choose a logical atom from the head of each disjunctive clause of an LPAD we obtain a so-called *world* of the LPAD whose probability is equal to the product of the probability values associated to the chosen logical atoms. If we sum up the probability of the worlds in which the desired goal is derived we obtain the probability itself of the goal.

A traffic example

Consider for instance the following example in which we want to model that the probability that a road is busy largely depends from the fact that there was an accident or, in a lesser extent, by the bad weather:

$$\texttt{traffic}(X) : 0.9 \leftarrow \texttt{accident}(X).$$
$$\texttt{traffic}(X) : 0.4 \leftarrow \texttt{bad\_weather}(X).$$

If we know that '$\texttt{accident}(\texttt{abbey\_road})$.', we can conclude that $\texttt{traffic}(\texttt{abbey\_road})$ with a probability of 0.9. Similarly, if we know that '$\texttt{bad\_weather}(\texttt{lombard\_street})$.', we can similarly determine that $\texttt{traffic}(\texttt{lombard\_street})$ with a probability of 0.4. Finally, if we know that '$\texttt{accident}(\texttt{long\_mile})$.' and '$\texttt{bad\_weather}(\texttt{long\_mile})$.', we can conclude $\texttt{traffic}(\texttt{long\_mile})$ with a probability of 0.94, which is higher than before because in this case both causes are applying at the same time.

Trivial way of determining probability

Each clause of the program, in fact, contains an implicit logical atom *null* with a probability of 0.1 and 0.6 respectively. This simple LPAD generates 4 worlds: to obtains the first world we chose the first atom of both clauses, for the second we chose the second atom of the first clause and the first atom of the second clause, for the third one the first atom of the first clause and the second atom of the second clause and, of course, for the forth world we chose the second atom of both clauses. As said, the probabilities of these worlds are

given by multiplying the values associated to the chosen atom and are respectively: $0.9 \times 0.4 = 0.36$, $0.1 \times 0.4 = 0.04$, $0.9 \times 0.6 = 0.54$ and $0.1 \times 0.6 = 0.06$. Now, the goal $\texttt{traffic(abbey\_road)}$ can be derived in the first and third worlds so its probability is $0.36 + 0.54 = 0.9$. In a similar fashion, the goal $\texttt{traffic(lombard\_street)}$ can be derived in the first and second worlds and its probability sums up to $0.36 + 0.04 = 0.4$. Finally, the goal $\texttt{traffic(long\_mile)}$ can be derived successfully in the first three worlds thus the resulting probability is $0.36 + 0.54 + 0.04 = 0.94$.

State-of-the-art way of determining probability

This method of computing probabilities is not practical and other more efficient algorithms have been proposed [144, 146] with several subsequent optimisations [145, 147, 148]. The basic algorithm uses a meta-interpreter that tries to resolve the goal while storing the information about the logical heads of the disjunctive clauses that were chosen during the process. The derivations accompanied by this data are called explanations. When all the explanations are found one by one, according to the operating schema that is typical of any backward-chaining system, they are converted into a compact diagram called BINARY DECISION DIAGRAM (BDD) that represents the choices done to obtain the explanations. This diagram has a root node and two terminals: by traversing it from the root to the positive end and by properly taking into account the probability values that are found along these paths, it is possible to efficiently compute the probability of the goal. The vast majority of approximated algorithms are derived from this one. The reader interested in these procedures may find more details in [29, 144, 146].

B.3 SOLVING PROBABILISTIC GRAPH PROBLEMS

Probabilistic problems as graph problems: an example

Many problems and almost all the datasets used to assess the efficiency of the algorithms on LPADs are formulated in term of networks. In these networks, nodes represent entities in a given domain. The edges between pairs of entities represent some relationship between them and they are typically associated with a probability value indicating the strength of the bond. Problems like this are addressed by considering a program that verifies the existence of connected paths between any pair of domain entities. The probability value returned by the program indicates the intensity of the indirect link between those entities. Figure B.1 on the facing page shows a fictional example of a real-world network extracted from a rather large biological graph [80] of 5,220 nodes and 11,530 edges on 4 genes responsible of Alzheimer's disease. The nodes represent biological entities such as proteins, genes, tissues, bio- logical organisms, molecular functions, etc. The edges express the relationships between those entities, whose strength is given by their probability values. Notice that the probabil-

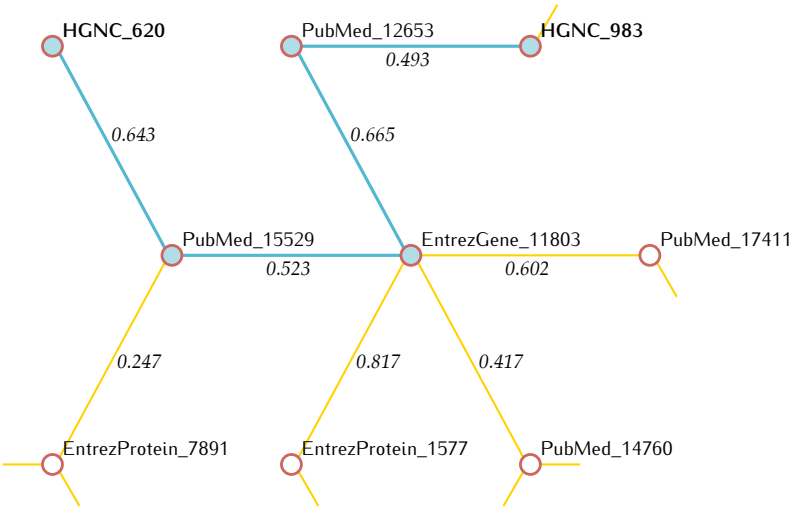


Figure B.1: A fictional example of an extract of biological network where a possible path between two genes responsible of Alzheimer's disease (HGNC_620 and HGNC_983) is highlighted.

ity of an indirect association among couples of entities is the same as computing the probability that a path exists between their nodes. The goal, in fact, consists in determining the probability that two genes, namely HGNC_620 and HGNC_983, are related.

Preparing a solution

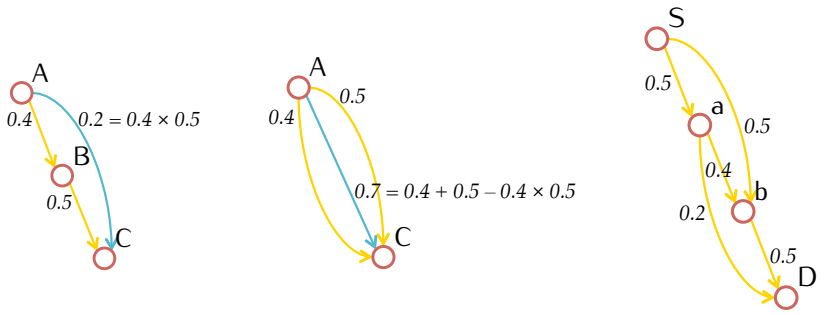
Our work starts from here: we made a simple interpreter that reads the information about nodes and edges (with probabilities) of this kind of problems and reifies them inside the Working Memory (WM) of a PRS. Starting from knowledge bases of this kind we have tried to build an algorithm to compute the probability that paths exist between any two nodes of the network. The initial results were encouraging, but only because we were considering networks exhibiting some regular patterns. In some networks, for instance, the paths were always branching, exploding the graph without ever merging the paths; in other cases, the paths were branching and merging but in a regular manner so that there were no intersections between subbranches. When we have considered more general problems, we have achieved performances that were reasonably in line with those of exact inference in backward-chaining environments, but never faster.

Topological, flow and classical approaches

We have addressed this class of problems by using three different approaches: a *topological* approach, a *flow* approach and a more *classical* approach, more similar to the algorithm seen in the previous section.

Description of the declarative algorithm

TOPOLOGICAL APPROACH The topological approach is based on the idea of transforming the network in a simpler form at each passage. The network is defined by introducing several instances of an `Edge` object into the WM of a PRS. The `Edge` object is declared as in List-



(a) Propagation **(b)** Merging **(c)** Unfeasible instance

Figure B.2: Transformations used to reduce the graph with the topological approach and an instance of the problem that is unfeasible for the approach.

```
1 declare Edge
2   tail: String
3   head: String
4   prob: float
5 end
```

Listing B.1: Basic Abstract Data Type to work with probability graph problems.

ing B.1 on the next page, where the tail and head nodes are distinct by string identifiers.

The progressive reduction of the graph eventually ends up in a single edge between the source and destination node, whose probability is equivalent to the probability that a path exists between them. The actions that are possible at each passage are two: the *propagation* and the *merging*. The propagation is aimed to identify two consecutive edges with no other ramification that are being substituted by a single edge whose probability is equal to the product of the probability values of the two former edges. This action is exemplified by the Figure B.2a. The rule that performs such task is included in Listing B.2 on the next page.

The merging, instead, identifies structures like eyelets composed by two distinct edges insisting on the same couple of nodes and, again, it tries to substitute them with a single edge. The formula to compute the resulting probability is slightly more complex because it implements the rule of the NoisyOR. This rule states that the result value is equal to the sum of the two former probability minus their product:

$$P_r = P_1 + P_2 - (P_1 \times P_2)$$

```
rule "Propagation"
when
  $e: Edge( $t: tail, $j: head, $pe: prob )
  $f: Edge( tail == $j, $h: head, $pf: prob )
  not Edge( this != $e, this != $f, tail == $j or head
    == $j )
then
  retract($e);
  retract($f);
  insert(new Edge($t, $h, $pe * $pf));
end
```

Listing B.2: Topological declarative algorithm for probabilistic graph problems: propagation.

```
rule "Merging"
when
  $e: Edge( $t: tail, $h: head, $pe: prob )
  $f: Edge( this != $e, tail == $t, head == $h, $pf:
    prob )
then
  retract($e);
  retract($f);
  insert(new Edge($t, $h, $pe + $pf - $pe * $pf));
end
```

Listing B.3: Topological declarative algorithm for probabilistic graph problems: merging.

An example of merging can be seen in Figure B.2b and the rule that performs this kind of transformation is shown in Listing B.3 on the next page.

Practical example and its limitations

Unfortunately, this approach only supports fully directed graphs [1] and fails to solve the more general instances of the problem like the simple example in Figure B.2c on the facing page.

Description of the declarative algorithm

FLOW APPROACH This second approach was loosely inspired by the Ford-Fulkerson algorithm which is used to compute the maximum flow in a flow network. The probability on the edges represents the capacity of the pipes of the network. The main difference with respect to the original algorithm is that, in this case, we try to build all the augmenting paths in one step to take advantage of the forward-chaining environment.

1 The algorithm may be extended to also support undirected – or partially undirected graphs – however we have not considered yet such case for sake of simplicity.

```
1  declare Prob
2    node: String
3    prob: float
4    degree: int
5    counter: int
6  end
7
8  declare Used
9    edge: Edge
10 end
```

Listing B.4: Flow declarative algorithm for probabilistic graph problems.

We need to introduce a few instances of a couple of objects to keep track of part of the graph that has already been visited. In Listing B.4 on the next page we introduce `Node` to decorate the nodes and `Used` to keep track of the edges that have been already propagated. The `Node` object has four fields: `node` – which is a string identifier that univocally identifies a node of the graph, `prob` – which is a probability value of the contributions of probability accumulated so far on the node, `degree` – which is the number of incoming edges of the referred node which is often colloquially called *indegree*, and `counter` – which is, as its name suggests, a counter to keep track of the number of incoming contributions currently received. The `Used` object contains only a reference `edge` to the associated `Edge`: when an instance of this object is generated for an edge, it means that the contribution of probability that flows across that edge has been already accounted.

As for the previous approach, we have two rules: the first one manages the simple propagation of a probability value across an edge and the second accumulates any additional contribution of probability that is received in a `Node`. These rules perform a computation that is similar to that of the rules of previous approach. We decided to use the same names to keep the comparison alive: `"Propagation"` simply multiplies the probability value of the tail `Node` with the probability of the `Edge` to return the probability of the head `Node` and `"Merging "` takes advantage of the NoisyOR formula to update the probability value of the head `Node` with the product of the probability of the tail `Node` with the probability of the `Edge`.

If the reader looks at the following Listing, he may distinguish three distinct blocks of patterns in the premise of the `"Propagation"` rule:

- the first one (line 14) identifies the `Node` of any node `$t` for whom the `counter` has exactly the same value of the *indegree* of the `Node` – in other words it finds the nodes that have already received all the expected probability contributions,

```
11 rule "Propagation"
12 when
13   // Node that has received all the prob contributions
14   Node( $t: node, $d: degree, counter == $d, $pt: prob )
15
16   // Edge departing from the above node, not used yet
17   $e: Edge( tail == $t, $h: head, $pe: prob )
18   not Used( edge == $e )
19
20   // No Node on the target node
21   not Node( node == $h )
22   accumulate( $ee: Edge( head == $h );
23     $c: count($ee)
24   )
25 then
26   insert(new Used($e));
27   insert(new Node($h, $pt * $pe, 1, $c.intValue()));
28 end
```

Listing B.5: Flow declarative algorithm for probabilistic graph problems: propagation.

- the second one (lines 17 and 18) selects an instance of `Edge` `$e` that departs from the tail node identified by the previous block and verifies that it has not been used yet – it checks that there is no `Used` object for `$e`,
- the third and last block (lines 21-24) checks that no `Node` object exists for the head node `$h` of the `Edge` instance `$e` identified above and contextually counts the number `$c` of its incoming edges – the *indegree* – for a later use.

As a consequence, the rule asserts a `Used` instance for the `Edge` `$e` – since it can be considered used now (see Listing B.5, line 26), and also an instance of `Node` for the head node `$h` whose `counter` is set to `1`, its `degree` to `$c` and its `prob` to the result of the product between the probabilities of the tail `Node` `$t` and the `Edge` `$e` itself (line 27).

The other rule identified by the string `"Merging"` addresses the case in which a `Node` instance for the head node `$h` already exists (see Listing B.6 on the next page). The premise of the rule is composed of three blocks of patterns as well: the first two blocks are exactly the same as the first two of the `"Propagation"` rule. Instead, the third and last block of this rule (line 39) simply detects the `Node` `$n` of the head node `$h` and the current value of its `counter` and `prob` fields. The consequence of the rule asserts again a `Used` instance for the `Edge` `$e` to state it as used (line 41), and then it modifies the status of the `Node` `$n` object (line 42). In particular, since a new probability contri-

```
29 rule "Merging"
30 when
31   // Node that has received all the prob contributions
32   Prob( $t: node, $d: degree, counter == $d, $pt: prob )
33
34   // Edge departing from the above node, not used yet
35   $e: Edge( tail == $t, $h: head, $pe: prob )
36   not Used( edge == $e )
37
38   // A Prob exists for target node, counter and prob
39   $n: Prob( node == $h, $c: counter, $ph: prob )
40 then
41   insert(new Used($e));
42   modify($n) {
43     setCounter($c + 1);
44     setProb($pt * $pe + $ph - $pt * $pe * $ph );
45   }
46 end
```

Listing B.6: Flow declarative algorithm for probabilistic graph problems: merging.

bution has been just accounted, it increments the value of the `counter` (line 43). Last but not least, it exploits the NoisyOR formula to update the probability value of the head node with the contribution coming from the edge under scrutiny (line 44).

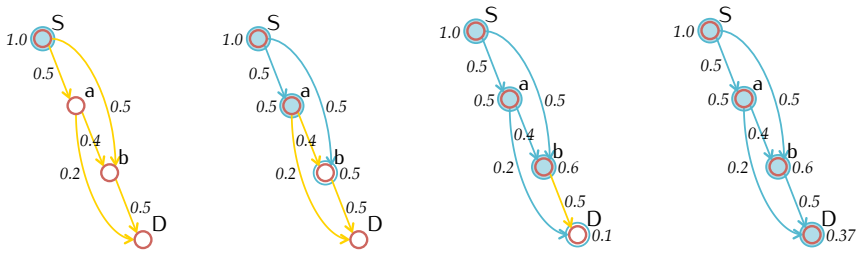
The covering of the network is triggered by asserting a `Prob` instance for the node from which the paths towards the destination node should start, whose probability of existence we want to discover. We trigger such reasoning by running the following statement: `session.insert(new Prob("S", 1.0, 0, 0));`. Notice that we impose `1.0` as the probability value for the starting node and we force both the `counter` and the `degree` to `0` to meet the first condition for both the `"Propagation"` and `"Merging"` rules.

Afterwards execution, the desired probability value can be read in the `prob` field of the `Prob` object that decorates the destination node once all the rules have triggered. Figure B.3 on the facing page contains an example that shows how the declarative algorithm given by the rules above solves a simple graph. Notice that this graph is an instance of the class of problems that was not covered by the previous approach.

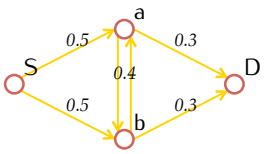
Practical example and its limitations

As said, initially we decorate the S node with a `Prob` whose probability is 1.0 and its `counter` and `degree` are 0 (see Figure B.3a).

At this point, the `"Propagation"` rule triggers on S and generates `Prob` objects for all the nodes that it is possible to reach from S (see FigureB.3b). As a result, the probability value in the `Prob` of both A



(a) Initial step **(b)** Propagation **(c)** Merging **(d)** Final step



(e) Unfeasible instance

Figure B.3: Progressive coverage of the graph with the flow approach and an instance of the problem that is unfeasible for the approach.

and B is 0.5 because both edges have probability 0.5 and S 1.0. Notice that we have represented `Prob` objects whose `counter == degree` with a cyan solid dot around the node, and those for whom it holds that `counter < degree` are hollow.

The node A, in fact, is already ready to propagate while B is not, as it is waiting for the contribution form A. Since a `Prob` object has been already introduced for B, A propagates towards B thanks to the `"Merge"` rule: instead of a simple product, the probability of B is computed with the formula of the NoisyOR (see Figure B.3c), returning

$$0.5 \times 0.4 + 0.5 - 0.5 \times 0.4 \times 0.5 = 0.2 + 0.5 - 0.1 = 0.6.$$

Notice that the `counter` of B is contextually incremented and meets the value of the `degree` (which is 2), making B a good candidate for further propagation. Also notice that during this step, A propagates towards D by means of the `"Propagation"` rule. Therefore the probability of D contained in its `Prob` is $0.5 \times 0.2 = 0.1$.

At this point, only B is ready to propagate as D is still missing the probability contribution from it. In the last step (Figure B.3d), the `"Merge"` rule brings the probability contribution from A to D which is now marked as ready to be propagated. The value in `Prob` of D becomes

$$0.1 + 0.6 \times 0.5 - 0.1 \times 0.6 \times 0.5 = 0.1 + 0.3 - 0.03 = 0.37.$$

D has no edges to propagate probability too, so the procedure ends. Since D is our destination node, it suffices to read the probability value in its `Prob` to get the desired probability that a path exists between S and D.

In addition, notice that the simple instance of the problem depicted in Figure B.3e can not be solved with this approach. This instance aims to model a partially undirected graph by introducing two directed edges of opposite verses for each undirected edge [2]. Unfortunately, this algorithm stalls on this simple instance because the `Prob` of both A and B wait for a contribution from each other, thus never propagating towards D.

Description of the declarative algorithm

CLASSICAL APPROACH The name of this approach stems from the fact that it more closely follows the typical procedure adopted in other formalisms such as the LPADs. In practice, the solution is obtained by performing two steps. During the first one we analyse the theory to find all the paths between the source and destination node, keeping track of all the visited edges and their probability values. Then, during the second phase we opportunely combine the probabilities of those paths to get the overall probability that a route exists between the given nodes.

In the original case of LPADs, the first step was accomplished by a LP resembling the following one:

$$
\begin{aligned}
&path(S, D, P_{out}) \leftarrow path(S, D, [S], P_{out}).\\
&path(S, D, P_{in}, [D|P_{in}]) \leftarrow arc(X, Y).\\
&path(S, D, P_{in}, P_{out}) \leftarrow\\
&\quad S \neq D, arc(S, N), \neg member(N, P_{in}), path(N, D, [N|P_{in}], P_{out}).\\
&arc(X, Y) \leftarrow edge(X, Y).\\
&arc(X, Y) \leftarrow edge(Y, X).
\end{aligned}
$$

This program tries to build and return a connected path P_{out} between the nodes S and D by using the available edges (that are typically annotated with probability values). If the nodes S and D are not directly connected, the program looks for a node N that can be reached from S that is not yet in P_{in} [3] to increase the length of the current path. Then it tries to find a path from N to D to be appended to the current one. Notice that this procedure is recursive and, if it is permitted by the topology of the graph – if the graph is partitioned, S and D pertain to the same island – it eventually reach D. Also notice that the edges in this formulation are indirected, therefore the arc predicate is needed to properly consider them in both directions.

The procedure that we propose stores the visited edges instead of the nodes, and uses `Destination` and `Shuttle` objects in addition to the `Edge` objects that define the graph to properly work. The choice

2 As we have suggested before, it is possible to handle this more general case by marking both opposite edges as used any time one of them is processed, but then other paths flowing in the opposite way could be lost; this more general case would require a deeper though and it was not included in this discussion for both lack of time and sake of simplicity as this is not the main topic of the dissertation.

3 Notice that this condition rules out any loop within graphs.

```
1  declare Destination
2    node: String
3  end
4
5  declare Shuttle
6    node: String
7    counter: int
8    degree: int
9    edges: List<Edge> = new ArrayList<Edge>();
10 end
```

Listing B.7: Classical declarative algorithm for probabilistic graph problems.

of storing edges instead of nodes was suggested by the fact that the edges were already available and they were containing all the needed information. The `Destination` is used to identify the destination node of the paths. Its purpose will be clear in the following paragraphs. The `Shuttle` is very similar to the `Node` object of the previous approach. The declaration of both `Destination` and `Shuttle` objects is found in Listing B.7: it contains a string identifier `node` for a node, a `counter` to keep track of the number of outgoing edges that were considered so far to find the paths, a `degree` which represents the number of outgoing edges from the node which is often colloquially referred as *outdegree* and a list `edges` of the edges visited so far that is automatically initiated upon instantiation of the `Shuttle` object.

As for two previous approaches, this one consists of only two rules as well. This time, however, the rules have a completely different meaning.

The first one is introduced in the following Listing B.8 on the next page. As the reader can see, the premise of the `"Navigate"` rule is quite complex: it aims to identify `Shuttle` objects on nodes with outgoing unvisited `Edge` objects. It also determines the *indegree* of the node that is going to be reached by following each `Edge` for a later use in the consequent of the rule.

In particular, line 13 looks for a `Shuttle` instance `$s`, extracting its `node` `$t`, its `counter` `$c` and its `edges` `$l`. Line 14 identifies any edge `$e` departing from the node `$t` extracting its head node `$h` and line 15 assures that it is included into the list of edges `$l` [4]. And finally lines 16-18 count the number `$d` of outgoing edges from head node `$h` to be used in the consequence of the rule.

The actions performed by the rule are the following. A new `Shuttle` object is generated for the head node `$h` with the `counter` initially set to `0` and the `degree` to `$d`. Remember from the declaration of `Shuttle` that the list `edges` is automatically initialised upon instanti-

4 This condition rules out any loop, if potentially present in the original graph.

```
11 rule "Navigate"
12 when
13   $s: Shuttle( $t: node, $c: counter, $l: edges )
14   $e: Edge( tail == $t, $h: head )
15   not eval($l.contains($e))
16   accumulate( $ee: Edge( tail == $h );
17     $d: count($ee)
18   )
19 then
20   Shuttle shuttle = new Shuttle($h, 0, $d.intValue());
21   shuttle.getEdges().addAll($l);
22   shuttle.getEdges().add($e);
23   insert(shuttle);
24   modify($s) {
25     setCounter($c + 1);
26   }
27 end
28
29 rule "Support"
30 when
31   Shuttle( $t: node, $c: counter, $l: edges )
32   $e: Edge( tail == $t )
33   eval($l.contains($e))
34 then
35   modify($s) {
36     setCounter($c + 1);
37   }
38 end
```

Listing B.8: Classical declarative algorithm for probabilistic graph problems.

```
39 rule "Cleanup"
40 when
41   $s: Shuttle( $n: node, $d: degree, counter == $d )
42   not Destination( node == $n)
43 then
44   retract($s);
45 end
```

Listing B.9: Classical declarative algorithm for probabilistic graph problems: cleapn-up.

ation (line 20). Then we add all the `edges` stored in the `Shuttle` `$s` of the tail node to the temporarily empty list of the new `Shuttle` (line 21). We also add the current `Edge` `$e` to the `edges` of the head

`Shuttle` as we are trying to use it to build the paths (line 22). The `Shuttle` instance is now fully configured and it is added to the WM (line 23). Finally we modify the `Shuttle` *`$s`* object (line 24) by incrementing the value of its `counter` (line 25).

Notice that when all the edges of a node have been explored, the `counter` will have exactly the same value of the `degree`. Also notice that there is a supporting rule which increments the `counter` of a `Shuttle` for the outgoing already visited edges (lines 29-38). The motivation for this rule will be clear in a moment.

The other rule called `"Cleanup"` is much simpler (see Listing B.9 on the facing page). It is used to get rid of the intermediate `Shuttle` objects that are not needed once the propagation of a node is over. The premise of the rule, in fact, looks for any `Shuttle` instance *`$s`* (line 41) not associated with the `Destination` node *`$n`* (line 42) whose `counter` value is exactly like its `degree` (line 41). The consequent of the rule simply retracts the *`$s`* object (line 44).

As the reader may guess, the only purpose of any `Destination` object is to preserve the `Shuttle` objects associates with it since they contain valid paths. Also notice that the `"Support"` node is needed to properly increase the `counter` of a `Shuttle` in presence of an already visited edge, otherwise the `"Cleanup"` could not trigger on that `Shuttle` instance. This rule is actually not mandatory but helps to keep under control the memory footprint of this approach. This statement does not mean that the usage of memory will never grow (it actually depends on the branching factor of the graph), but only that the objects that are not strictly necessary are released as soon as possible.

Practical example and its limitations

The search is triggered by asserting a `Destination` instance for the destination node and, as for the flow approach, by also asserting a `Shuttle` object for the source node. Notice that it is possible to declare several source and destination nodes and the procedure is robust enough to determine all the paths that start from any source node to any destination node, while discarding all the unnecessary partial paths. The Figure B.4 on the next page shows how the instance of the problem that was unfeasible for the previous approach is solved.

First of all we assert a `Destination` object for the end point of the path: `session.insert(new Destination("D"));`. Then we run a quick query to determine the number of outgoing edges from the source node of the path. Notice that we have not provided the code for such a query for sake of simplicity, but the reader should have no problem to create it by himself. In this example, the number of outgoing edges from the source node is 2 therefore we issue the following statement: `session.insert(new Shuttle("S", 0, 2));` (remember that the list is automatically initialised).

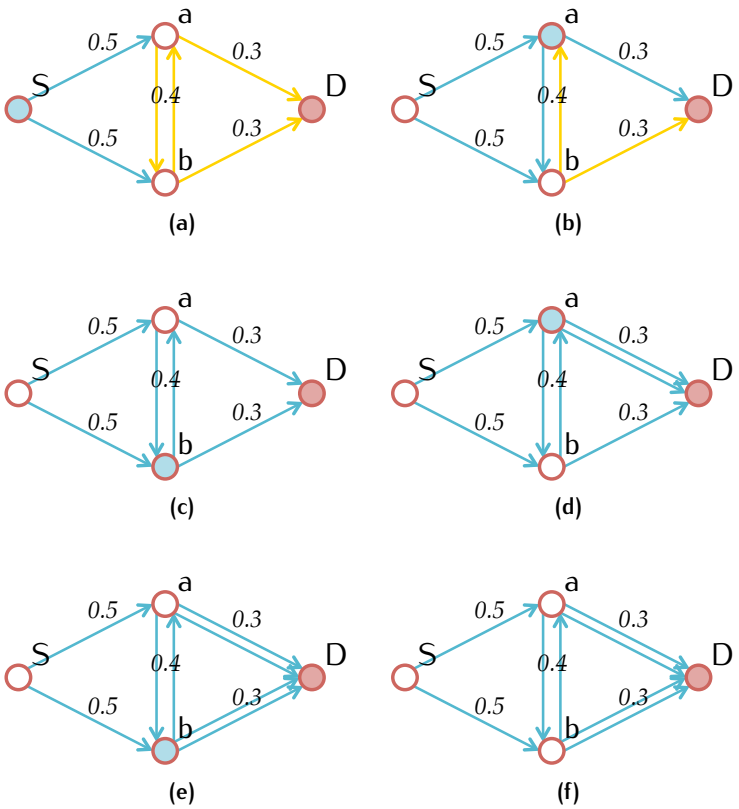


Figure B.4: Traversing the graph by following the edges to find all the paths from the source node to the destination, as in the more classical approach.

This initial configuration is shown in Figure B.4a. Notice that we have highlighted in cyan the node that is being processed and in red the destination node. The edges that are going to be used are highlighted in cyan as well. At the end of this step, the `"Navigate "` rule has triggered twice, both times on S. These activations have created a `Shuttle` object for both A and B, containing a partial path with one of these edges. In addition, the `counter` of S has reached the *outdegree*, therefore the `"Cleanup"` rule has triggered as well disposing the `Shuttle` object for S.

Then the `"Navigate"` rule triggers again, this time on the `Shuttle` object of the node A (see Figure B.4b). Similarly to what happened in the previous step, two new `Shuttle` objects have been created, both with an updated partial path. These are located in D and B (where another instance of `Shuttle` is waiting). Again, the `Shuttle` object in A has been disposed.

Then, one of the `Shuttle` objects in B (the first instance, for example) trigger the `"Navigate"` rule again (Figure B.4c). Two additional `Shuttle` objects are created: one in A and the other in D. Both contain an updated path, so that it is different from what it was stored in the

just disposed `Shuttle` instance in A and the one that is stored in the other `Shuttle` in D. The first of the two `Shuttle` objects is removed from the WM as it has already accomplished its task.

In the following step, the `"Navigate"` rule triggers again on the `Shuttle` object in A (Figure B.4d). This time, the `Shuttle` object triggers the `"Navigate"` rule only with the edge towards D (adding another `Shuttle` object with an updated path). The edge towards B was already visited by this `Shuttle` object so it triggers the `"Support"` rule instead. In either cases, the `counter` of the `Shuttle` object in α meets the value of the *outdegree* and the `"Cleanup"` rule disposes the `Shuttle` object in A.

Now, the last `Shuttle` object in B is activated (Figure B.4e). Both `"Navigate"` and `"Support"` rules are triggered as in the previous case. The `Shuttle` object in B already contains the edge towards A, so the increase of the value of its `counter` is the only action that is performed. It does not contain instead the edge towards D therefore in addition to the increase of the `counter`, a new `Shuttle` is added in D with an updated path.

The only `Shuttle` objects that are currently available are all located in D (Figure B.4f). Their `counter` is set to 0 as well as the `degree`, nevertheless these objects are not retracted because they are preserved by the presence of a `Destination` object in D. With an additional simple query (not included to keep the discussion simple) it is possible to retrieve in D all the valid path found.

From paths to probability

These paths are like the explanations that were found for the LPADs: by comparing the edges that compose each path it is possible to compute the probability that a path exists between S and D. The parts that are shared between paths should be accounted only once and alternative routes should be handled with the NoisyOR formula. This process, however, may become very complex as the complexity of the underlying graph increases. It certainly is very inefficient if the edges that are part of each path are compared one by one. To this end, it is possible to use some *Java* BDD package that computes first a diagram that summarises the paths and then traverses it efficiently to compute the resulting probability value [5].

In conclusion, this approach is capable to solve the largest class of probability graph problems, however it does not represent a fully satisfying solution. It was our hope that the efficient algorithm to select object of PRSs would have lead to improvements in performance to justify the choice.

Unfortunately this approach is unsuccessful because, instead of exploiting such algorithm, it simply readjusts the procedure suitable for

5 We have tested, for example, JEDD (`http://www.sable.mcgill.ca/jedd/`), SableJBDD (`http://www.sable.mcgill.ca/~fqian/SableJBDD/`), JDD (`http://javaddlib.sourceforge.net/jdd/`), JavaBDD (`http://javabdd.sourceforge.net`). All these tools are quite similar and no one has features that make it preferable with respect to the others.

backward-chaining systems by finding first a path at a time and then combining their probability together. In this case, in fact, the consolidated solution does not seem to be appropriate for the characteristics of the system under consideration.

Improving the approaches

OPTIMISATIONS The complexity of all the approaches discussed so far depends, in one way or another, on the number of edges that are included in the graph. With the topological approach, for instance, a graph of n edges can be reduced to a single equivalent edge in $n-1$ activations of the rules. The flow approach exactly triggers a rule for each of the n edges of the graph. The classical approach, instead, triggers at least a rule for each edge of the graph, but each undirected edge of the graph leads to several additional activations whose number depends on the distance of the undirected edge from the destination node that we can estimate between n and n^2.

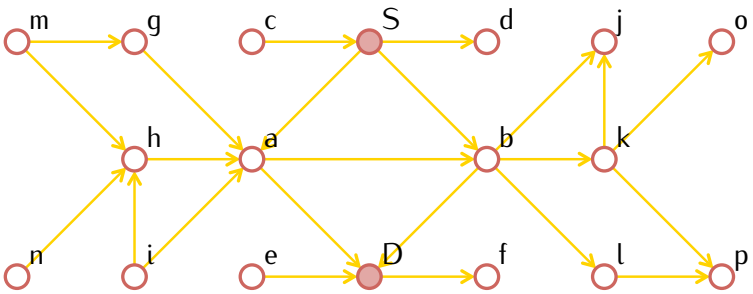
Regardless of the approach chosen, it is evident that the more one reduces the number of edges (discarding unneeded ones), the more he gets better performance. The expected result is to remove all the edges that are not part of any path between the source and destination nodes. Consider that any node of the graph may be chosen as source or destination: in practice it means that these nodes, if not already so, become nodes with null *indegree* or *outdegree* respectively. Any other node on a valid path will have *indegree* and *outdegree* strictly greater than zero as it has to be a link of the chain that binds the source with the destination node. Thanks to this consideration, we can state that any source (*indegree* equals to 0) or sink (*outdegree* equals to 0) nodes different from the terminal nodes will not contribute in any way to the creation of valid paths. Therefore we can remove these nodes and all the outgoing (for sources) and incoming (for sinks) edges that refer to them. In this fashion, we expose new nodes that were intermediate as sources or sinks therefore we can further apply the pruning process. When the procedure halts, the remaining portion of the graph is in *minimal form* as it only includes the edges that are meaningful to build the path among the extremes.

Description of the declarative algorithm

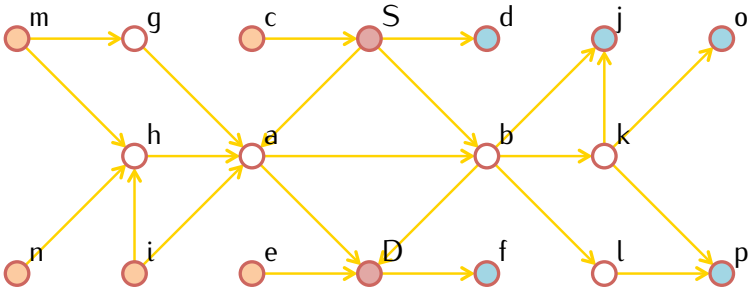
This procedure is implemented by two specular rules `"Sources"` and `"Sinks"`, and a special object `Extreme` (with a string identifier for a node) to mark the source and destination nodes, as reported in Listing B.10 on page 202. Each rule identifies an edge `$e` and its tail or head node respectively (lines 7 or 16). This node must not be an extreme (lines 8 or 17). Finally, no edge can enter or respectively exit from this node (lines 9 or 18). If so, the edge `$e` is retracted from the WM (lines 11 or 20). See the following Listing for the details.

Practical example

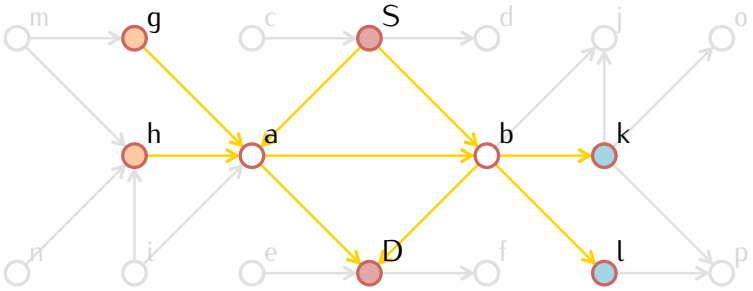
Figure B.5 on the next page shows a practical example of this process at work. Either S and D are marked as `Extreme` nodes by issuing the following commands: `session.insert(new Extreme("S"));`, `session.insert(new Extreme("D"));` (see Figure B.5a, where the `Ex-`



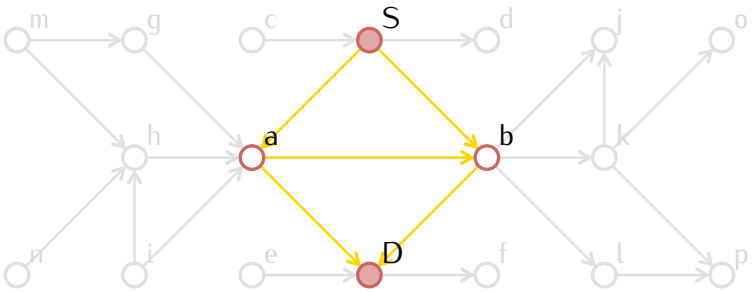
(a) Defining the terminal nodes



(b) Identifying and removing source and sink nodes and their edges



(c) Identifying and removing new source and sink nodes and their edges



(d) The resulting graph in minimal form

Figure B.5: Progressive reduction of a graph to optimise the following computation process by removing the irrelevant parts.

```
declare Extreme
  node: String
end

rule "Sources"
when
  $e: Edge( $t: tail )
      not Extreme( node == $t )
      not Edge( head == $t )
then
  retract($e);
end

rule "Sinks"
when
  $e: Edge( $h: tail )
      not Extreme( node == $h )
      not Edge( tail == $h )
then
  retract($e);
end
```

Listing B.10: Declarative algorithm for optimising the former approaches.

treme nodes are coloured in red). Then the procedure is started by calling `session.fireAllRules();`. Irrelevant source and sink nodes are individuated and the edges directly connected with them removed from memory (see Figure B.5b and B.5c). Detected source and sink nodes are respectively coloured in orange and cyan. When `"Sources"` and `"Sinks"` stop to trigger, the remaining part of the graph is the core problem that needs to be solved by the previous approaches (see Figure B.5d).

B.4 SUMMARY

This Appendix contains a brief introduction to PILP and the formalism for LPADs is presented as an example. We also show that probabilistic problems can often be represented as graph problems where nodes and edges are annotated in some way with probability values. The solution of these problems typically consists in finding the probability that two (or more) given nodes of the knowledge base are connected.

In the second part of the Appendix, we present three possible approaches to address this class of problems within a PRS. Some approaches are interesting because their complexity (intended as the

number of pattern matching phases and rule activations) is linear in the number of edges of the problem, however they only cope with sub-classes of the original problem class. We also introduce an approach that solves the more general problem, but unfortunately it does not take advantage of the benefits deriving from the adoption of a PRS.

A possible solution is to introduce a new approach that combines the most complete exploration strategy of the general approach with the procedure for computing the probability proposed with the former approaches. This will be however matter for future work. It should be noted that, although these methods are not completely satisfactory, they can be used to enhance the expressiveness of the mechanism that manages expectations that is the subject of this dissertation.

BIBLIOGRAPHY

[1] Adams, Michael and ter Hofstede, Arthur H.M. and Edmond, David and van der Aalst, Wil M.P. *Worklets: A service-oriented implementation of dynamic flexibility in workflows*. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 291–308, 2006.
(Cited on pages 8 and 9)

[2] Adriansyah, Arya and van Dongen, Boudewijn F. and van der Aalst, Wil M.P. *Cost-Based Conformance Checking using the A* Algorithm*. *BPM Center Report BPM-11-11*, 2011. URL: http://bpmcenter.org.
(Cited on page 5)

[3] Agarwal, Sudhir and Lamparter, Steffen. *Smart-A semantic matchmaking portal for electronic markets*. In *Seventh IEEE International Conference on E-Commerce Technology, 2005*, pages 405–408. IEEE, 2005.
(Cited on page 171)

[4] Alberti, Marco and Chesani, Federico and Gavanelli, Marco and Lamma, Evelina and Mello, Paola and Montali, Marco and Torroni, Paolo. *Expressing and verifying business contracts with abductive logic programming*. *International Journal of Electronic Commerce*, 12(4):9–38, 2008.
(Cited on pages 8, 10, and 11)

[5] Alberti, Marco and Chesani, Federico and Gavanelli, Marco and Lamma, Evelina and Mello, Paola and Torroni, Paolo. *Verifiable agent interaction in abductive logic programming: the SCIFF framework*. *Association for Computing Machinery (ACM) Transactions on Computational Logic (TOCL)*, 9(4):29, 2008.
(Cited on page 93)

[6] Alberti, Marco and Gavanelli, Marco and Lamma, Evelina and Mello, Paola and Torroni, Paolo and Sartor, Giovanni. *Mapping deontic operators to abductive expectations*. *Computational & Mathematical Organization Theory*, 12(2):205–225, 2006.
(Cited on page 93)

[7] Allen, James F. *An interval-based representation of temporal knowledge*. In *Proceedings of the 7th international joint conference on Artificial intelligence-Volume 1*, pages 221–226. Morgan Kaufmann Publishers Inc., 1981.
(Cited on pages 104 and 168)

[8] Allen, James F. *Maintaining Knowledge about Temporal Intervals*. *Commununications of the Association for Computing Machinery (ACM)*, 26(11):832–843, 1983.
(Cited on page 40)

[9] Allen, James F. *Maintaining knowledge about temporal intervals*. *Communications of the Association for Computing Machinery (ACM)*, 26(11):832–843, 1983.
(Cited on pages 104 and 168)

[10] Allen, James F. *Maintaining Knowledge about Temporal Intervals*. *Expert systems: a software methodology for modern applications*, 26: 248, 1990.
(Cited on page 104)

[11] Allweyer, Thomas. *BPMN 2.0 - Introduction to the Standard for Business Process Modeling*. BoD - Books on Demand, 2010.
(Cited on page 8)

[12] Alonso, Gustavo and Casati, Fabio and Kuno, Harumi A. and Machiraju, Vijay. *Web services: concepts, architectures and applications*. Springer, 2003.
(Cited on page 8)

[13] Amador, Lucas. *Drools Developer's Cookbook*. Packt Publishing, 2012. ISBN: 1849511969.
(Cited on page 158)

[14] Antoniou, Grigoris and Damásio, Carlos Viegas and Grosof, Benjamin N. and Horrocks, Ian and Kifer, Michael and Maluszynski, Jan and Patel-Schneider, Peter F. *Combining rules and ontologies: A survey*. *Reasoning on the Web with Rules and Semantics*, 2005.
(Cited on page 77)

[15] Artikis, Alexander and Sergot, Marek and Paliouras, Georgios. *Run-time composite event recognition*. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 69–80. ACM, 2012.
(Cited on page 31)

[16] Awad, Ahmed and Decker, Gero and Weske, Mathias. *Efficient compliance checking using BPMN-Q and temporal logic*. *Business Process Management*, pages 326–341, 2008.
(Cited on pages 8 and 10)

[17] Awad, Ahmed and Weidlich, Matthias and Weske, Mathias. *Specification, verification and explanation of violation for data aware compliance rules*. *Service-Oriented Computing*, pages 500–515, 2009.
(Cited on pages 8 and 10)

[18] Baader, Franz and Horrocks, Ian and Sattler, Ulrike. *Description logics*. *Foundations of Artificial Intelligence*, 3:135–179, 2008.
(Cited on page 82)

[19] Baldwin, James F. and Martin, Trevor Patrick and Pilsworth, B.W. *FRIL-fuzzy and evidential reasoning in artificial intelligence*. John Wiley & Sons, Inc. New York, NY, USA, 1995.
(Cited on page 79)

[20] Bali, Michal. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009. ISBN: 1847195644, 9781847195647.
(Cited on page 158)

[21] Baral, Chitta and Gelfond, Michael and Rushton, J. Nelson. *Probabilistic reasoning with answer sets*. *Logic Programming and Nonmonotonic Reasoning*, pages 21–33, 2004.
(Cited on page 184)

[22] Barros, Alistair and Dumas, Marlon and ter Hofstede, Arthur H.M. *Service interaction patterns*. *Business Process Management*, pages 302–318, 2005.
(Cited on page 8)

[23] Batory, Donald S. *The LEAPS algorithms*. *Relation*, 10(1.16):8539, 1994.
(Cited on page 153)

[24] Bechhofer, Sean and Goble, Carole A. *Description Logics and Multimedia-Applying Lessons Learnt from the GALEN Project*, 1996.
(Cited on page 171)

[25] Behrends, Erik and Fritzen, Oliver and May, Wolfgang and Schenk, Franz. *Embedding Event Algebras and Process for ECA Rules for the Semantic Web*. *Fundamenta Informaticae*, 82(3):237–263, 2008.
(Cited on page 79)

[26] Bennett, Brandon and Galton, Antony P. *A unifying semantics for time and events*. *Artificial Intelligence*, 153(1):13–48, 2004.
(Cited on page 168)

[27] Berry, Andrew and Milosevic, Zoran. *Extending choreography with business contract constraints*. *International Journal of Cooperative Information Systems*, 14(2-3):131–179, 2005.
(Cited on pages 8 and 11)

[28] Bobillo, Fernando and Straccia, Umberto. *Towards a Crisp Representation of Fuzzy Description Logics under Łukasiewicz Semantics*.

In A. An, S. Matwin, Z. W. Raś, and D. Ślęzak, editors, *Proceedings of the 17th International Symposium on Methodologies for Intelligent Systems (ISMIS 2008)*, volume 4994 of *Lecture Notes in Computer Science*, pages 309–318. Springer Verlag, May 2008.
(Cited on page 83)

[29] Bragaglia, Stefano. *Reasoning on Logic Programs with Annotated Disjunctions*. *Intelligenza Artificiale*, 6(1):77–96, 2012. ISSN: 2211-0097. URL: http://dx.doi.org/10.3233/IA-2012-0029.
(Cited on page 186)

[30] Bragaglia, Stefano and Luccarini, Luca and Mello, Paola and Pulcini, Dalila and Sottara, Davide. *Monitoring the performance of Soft Sensors used in WWTPs by means of Formal Verification*. In *Proceedings of 6th International Congress on Environmental Modelling and Software*. iEMSs, 2012. URL: http://www.iemss.org/sites/iemss2012//proceedings/G1_2_0924_Luccarini_et_al.pdf.
(Cited on page 135)

[31] Bragaglia, Stefano and Luccarini, Luca and Mello, Paola and Pulcini, Dalila and Sottara, Davide. *Ontologies, Rules, Workflow and Predictive Models: Knowledge Assets for an EDSS*. In *Proceedings of 6th International Congress on Environmental Modelling and Software*. iEMSs, 2012. URL: http://www.iemss.org/sites/iemss2012//proceedings/A3_0926_Sottara_et_al.pdf.
(Cited on page 134)

[32] Bragaglia, Stefano and Riguzzi, Fabrizio. *Approximate Inference for Logic Programs with Annotated Disjunctions*. In P. Frasconi and F. Lisi, editors, *Inductive Logic Programming*, volume 6489 of *Lecture Notes in Computer Science*, pages 30–37. Springer Berlin / Heidelberg, 2011. ISBN: 978-3-642-21294-9. URL: http://dx.doi.org/10.1007/978-3-642-21295-6_7.
(Cited on page 184)

[33] Browne, Paul. *JBoss Drools Business Rules*. Packt Publishing, 2009. ISBN: 1847196063, 9781847196064.
(Cited on page 158)

[34] Carnap, Rudolf. *Logical Foundations of Probability*. University of Chicago Press, 1950.
(Cited on page 184)

[35] Casati, Fabio and Ceri, Stefano and Pernici, Barbara and Pozzi, Giuseppe. *Workflow evolution*. *Data & Knowledge Engineering*, 24 (3):211–238, 1998.
(Cited on pages 8 and 9)

[36] Cervesato, Iliano and Franceschet, Massimo and Montanari, Angelo. *A guided tour through some extensions of the event calculus*.

Computational Intelligence, 16(2):307–347, 2000.
(Cited on page 26)

[37] Chen, Harry and Fellah, Stephane and Bishr, Yaser A. *Rules for geospatial semantic web applications*. In *W3C Workshop on Rule Languages for Interoperability*, pages 27–28, 2005.
(Cited on page 171)

[38] Chen, Li and Reichert, Manfred U. and Wombacher, Andreas. *The MinAdept clustering approach for discovering reference process models out of process variants*. *International Journal of Cooperative Information Systems*, 19(03n04):159–203, 2010.
(Cited on pages 8 and 9)

[39] Chesani, Federico and Mello, Paola and Montali, Marco and Torroni, Paolo. *A logic-based, reactive calculus of events*. *Fundamenta Informaticae*, 105(1):135–161, 2010.
(Cited on page 27)

[40] Chittaro, Luca and Montanari, Angelo. *Efficient Handling of Context-Dependency in the Cached Event Calculus*. In *Proceedings of TIME'94 - International Workshop on Temporal Representation and Reasoning*, pages 103–112, 1994.
(Cited on page 27)

[41] Cholak, Peter and Blair, Howard A. *The complexity of local stratification*. *Fundamenta Informaticae*, 21(4):333–344, 1994.
(Cited on page 36)

[42] Cicekli, Niham Kesim and Cicekli, Ilyas. *Formalizing the specification and execution of workflows using the event calculus*. *Information Sciences*, 176(15):2227–2267, 2006. URL: http://dx.doi.org/10.1016/j.ins.2005.10.007.
(Cited on page 22)

[43] Clark, Keith L. *Negation as failure*. *Logic and data bases*, 1:293–322, 1978.
(Cited on page 23)

[44] Damiani, Gianfranco and Pinnarelli, Luigi and Colosimo, Simona C. and Almiento, Roberta and Sicuro, Lorella and Galasso, Rocco and Sommella, Lorenzo and Ricciardi, Walter. *The effectiveness of computerized clinical guidelines in the process of care: a systematic review*. *BMC Health Services Research*, 10(1):2, 2010.
(Cited on page 91)

[45] Dantsin, Evgeny. *Probabilistic Logic Programs and their Semantics*. In *Logic Programming, First Russian Conference on Logic Programming, Irkutsk, Russia, September 14-18, 1990 - Second Russian Conference on Logic Programming, St. Petersburg, Russia, September 11-16,*

1991, Proceedings, volume 592 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 1991.
(Cited on page 184)

[46] DE BRUIJN, JOS. *Semantic Web Language Layering with Ontologies, Rules, and Meta-Modeling*. PhD thesis, University of Innsbruck, 2008.
(Cited on page 79)

[47] DE RAEDT, LUC AND DEMOEN, BART AND FIERENS, DAAN AND GUTMANN, BERND AND JANSSENS, GERDA AND KIMMIG, ANGELIKA AND LANDWEHR, NIELS AND MANTADELIS, THEOFRASTOS AND MEERT, WANNES AND ROCHA, RICARDO AND OTHERS. *Towards digesting the alphabet-soup of statistical relational learning*. In *NIPS Workshop on Probabilistic Programming*, 2008.
(Cited on page 184)

[48] L. DE RAEDT, P. FRASCONI, K. KERSTING, AND S. H. MUGGLETON, EDITORS. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN: 978-3-540-78651-1.
(Cited on page 184)

[49] DE RAEDT, LUC AND KIMMIG, ANGELIKA AND TOIVONEN, HANNU. *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In *International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.
(Cited on page 184)

[50] DECKER, GERO AND WESKE, MATHIAS. *Behavioral consistency for B2B process integration*. In *Advanced Information Systems Engineering*, pages 81–95. Springer, 2007.
(Cited on page 8)

[51] DECKER, GERO AND WESKE, MATHIAS. *Interaction-centric modeling of process choreographies*. *Information Systems*, 36(2):292–312, 2011.
(Cited on page 8)

[52] DEPAIRE, BENOÎT AND SWINNEN, JO AND JANS, MIEKE AND VANHOOF, KOEN. *A Process Deviation Detection and Diagnosis Framework*. In *Proceedings of the 1st Joint International Workshop on Security in Business Processes*, 2012. URL: http://hdl.handle.net/1942/13928.
(Cited on pages 5 and 6)

[53] DESAI, NIRMIT AND CHOPRA, AMIT K. AND SINGH, MUNINDAR P. *Representing and Reasoning about Commitments in Business Processes*. In *AAAI*, pages 1328–1333. AAAI Press, 2007. ISBN: 978-1-57735-323-2.
(Cited on page 93)

[54] DI MONTE, STEFANO. *Sistema di pose-prediction e monitoraggio real-time basato sull'utilizzo dello strumento Kinect e del formalismo logic-based Event Calculus*. Technical report, DISI – University of Bologna, 2012. In Italian.
(Cited on page 114)

[55] DOORENBOS, ROBERT B. *Production matching for large learning systems*. PhD thesis, University of Southern California, 1995.
(Cited on page 153)

[56] DUSTDAR, SCHAHRAM. *Caramba –– a process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams*. *Distributed and Parallel Databases*, 15(1):45–66, 2004.
(Cited on pages 8 and 9)

[57] DWYER, MATTHEW B. AND AVRUNIN, GEORGE S. AND CORBETT, JAMES C. *Property specification patterns for finite-state verification*. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. Association for Computing Machinery (ACM), 1998.
(Cited on pages 8 and 10)

[58] EL KHARBILI, MARWANE AND DE MEDEIROS, ANA KARLA A. AND STEIN, SEBASTIAN AND VAN DER AALST, WIL M.P. *Business process compliance checking: Current state and future challenges*. *Modelling Business Information Systems*, pages 107–113, 2008.
(Cited on pages 8 and 10)

[59] ESPINOSA, ENRIQUE DAVID AND BUENO, ABEL AND MOLÍNA, MARTÍN AND MUÑOZ, JESÚS. *Modeling Business Diagnosis with Dynamic Workflow Construction*. In *Proceedings of SOLI'06 - the IEEE International Conference on Service Operations and Logistics and Informatics*, pages 197–202. IEEE, 2006.
(Cited on page 95)

[60] FARRELL, ANDREW D.H. AND SERGOT, MAREK J. AND SALLÉ, MATHIAS AND BARTOLINI, CLAUDIO. *Using the Event Calculus for Tracking the Normative State of Contracts*. *International Journal of Cooperative Information Systems*, 14(2-3):99–129, 2005.
(Cited on pages 22 and 23)

[61] FERNANDES, ALVARO A.A. AND WILLIAMS, M. HOWARD AND PATON, NORMAN W. *A logic-based integration of active and deductive databases*. *New Generation Computing*, 15(2):205–244, 1997.
(Cited on page 22)

[62] FORGY, CHARLES L. *RETE: A fast algorithm for the many pattern/-many object pattern match problem*. *Artificial Intelligence*, 19(1):17–37, 1982.
(Cited on pages 153 and 155)

[63] Forgy, Charles L. *Rete: A Fast Algorithm for the Many Pattern-s/Many Objects Match Problem*. *Artificial Intelligence*, 19(1):17–37, 1982.
(Cited on pages 16 and 17)

[64] Fornara, Nicoletta and Colombetti, Marco. *A Commitment-Based Approach To Agent Communication*. *Applied Artificial Intelligence*, 18 (9-10):853–866, 2004.
(Cited on page 93)

[65] Fuhr, Norbert. *Probabilistic datalog: Implementing logical information retrieval for advanced applications*. *JASIS*, 51(2):95–110, 2000.
(Cited on page 184)

[66] Gaifman, Haim. *Concerning measures in first order calculi*. *Israel Journal of Mathematics*, 2:1–18, 1964.
(Cited on page 184)

[67] Gamma, Erich and Helm, Richard and Johnson, Ralph E. and Vlissides, John M. *Design patterns: Abstraction and reuse of object-oriented design*. *ECOOP'93—Object-Oriented Programming*, pages 406–431, 1993.
(Cited on pages 84 and 99)

[68] L. Getoor and B. Taskar, editors. *An Introduction to Statistical Relational Learning*. MIT Press, 2007.
(Cited on page 184)

[69] Ghose, Aditya K. and Koliadis, George. *Auditing business process compliance*. *Service-Oriented Computing–ICSOC 2007*, pages 169–180, 2007.
(Cited on pages 8 and 10)

[70] Giblin, Christopher and Müller, Samuel and Pfitzmann, Birgit. *From regulatory policies to event monitoring rules: Towards model-driven compliance automation*. *IBM Research Zurich, Report RZ*, 3662, 2006.
(Cited on pages 8 and 11)

[71] Goedertier, Stijn and Vanthienen, Jan. *Designing compliant business processes with obligations and permissions*. In *Business Process Management Workshops*, pages 5–14. Springer, 2006.
(Cited on pages 8, 10, and 11)

[72] Golbreich, Christine and Bierlaire, Olivier and Dameron, Olivier and Gibaud, Bernard. *Use case: Ontology with rules for identifying brain anatomical structures*. In *W3C Workshop*, pages 1–5, 2005.
(Cited on page 171)

[73] Governatori, Guido. *Representing business contracts in RuleML*. *International Journal of Cooperative Information Systems*, 14(02n03):

181–216, 2005.
(Cited on page 93)

[74] Governatori, Guido and Hulstijn, Joris and Riveret, Régis and Rotolo, Antonino. *Characterising deadlines in temporal modal defeasible logic*. *AI 2007: Advances in Artificial Intelligence*, pages 486–496, 2007.
(Cited on page 93)

[75] Governatori, Guido and Milosevic, Zoran and Sadiq, Shazia Wasim. *Compliance checking between business processes and business contracts*. In *Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International*, pages 221–232. IEEE, 2006.
(Cited on pages 8, 9, and 10)

[76] Governatori, Guido and Rotolo, Antonino. *Norm Compliance in Business Process Modeling*. In *RuleML*, pages 194–209, 2010.
(Cited on page 93)

[77] Governatori, Guido and Rotolo, Antonino and Sartor, Giovanni. *Temporalised normative positions in defeasible logic*. In *Proceedings of the 10th international conference on Artificial intelligence and law*, pages 25–34. Association for Computing Machinery (ACM), 2005.
(Cited on page 93)

[78] Grosof, Benjamin N. and Horrocks, Ian and Volz, Raphael and Decker, Stefan. *Description logic programs: Combining logic programs with description logic*. In *Proceedings of the 12th international conference on World Wide Web*, pages 48–57. Association for Computing Machinery (ACM) New York, NY, USA, 2003.
(Cited on pages 77 and 79)

[79] Günther, Christian W. and Rinderle-Ma, Stefanie B. and Reichert, Manfred U. and van der Aalst, Wil M.P. *Change mining in adaptive process management systems*. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 309–326, 2006.
(Cited on pages 8 and 9)

[80] Gutmann, Bernd and Kimmig, Angelika and Kersting, Kristian and De Raedt, Luc. *Parameter learning in probabilistic databases: A least squares approach*. *Machine Learning and Knowledge Discovery in Databases*, pages 473–488, 2008.
(Cited on page 186)

[81] Herchenröder, Thomas. *Lightweight semantic web oriented reasoning in Prolog: tableaux inference for description logics*. Master's thesis, Master of science, School of Informatics, University of Edin-

burgh, 2006.
(Cited on page 16)

[82] Horn, Alfred. *On sentences which are true of direct unions of algebras*. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.
(Cited on page 23)

[83] Huhns, Michael N. and Singh, Munindar P. *Service-oriented computing: Key concepts and principles*. *Internet Computing, IEEE*, 9(1): 75–81, 2005.
(Cited on page 22)

[84] Hustadt, Ullrich and Motik, Boris and Sattler, Ulrike. *Reducing SHIQ- description logic to disjunctive Datalog programs*. *Proceedings of KR 2004 on Knowledge Representation*, pages 152–162, 2004.
(Cited on pages 77 and 79)

[85] Kandel, Abraham. *Fuzzy techniques in pattern recognition*. Cambridge University Press, 1982.
(Cited on page 171)

[86] Kersting, Kristian and De Raedt, Luc. *Towards combining Inductive Logic Programming with Bayesian networks*. *Inductive Logic Programming*, pages 118–131, 2001.
(Cited on page 184)

[87] Kifer, Michael. *Requirements for an expressive rule language on the semantic web*. In *W3C Workshop on Rule Languages for Interoperability*. Citeseer, 2005.
(Cited on page 171)

[88] Kimmig, Angelika and Costa, Vítor Santos and Rocha, Ricardo and Demoen, Bart and De Raedt, Luc. *On the Efficient Execution of ProbLog Programs*. In *Proceedings of ICLP 2008 - International Conference on Logic Programming*, pages 175–189, 2008.
(Cited on page 184)

[89] Kimmig, Angelika and Demoen, Bart and De Raedt, Luc and Costa, Vítor Santos and Rocha, Ricardo. *On the implementation of the probabilistic logic programming language ProbLog*. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.
(Cited on page 184)

[90] Knuplesch, David and Ly, Linh-Thao and Rinderle-Ma, Stefanie B. and Pfeifer, Holger and Dadam, Peter. *On enabling data-aware compliance checking of business process models*. *Conceptual Modeling–ER 2010*, pages 332–346, 2010.
(Cited on pages 8 and 10)

[91] Knuplesch, David and Reichert, Manfred U. *Ensuring Business Process Compliance Along the Process Life Cycle*. Technical Report

UIB-2011-06, University of Ulm, Ulm, July 2011. URL: http://dbis.eprints.uni-ulm.de/749/.
(Cited on pages 8 and 10)

[92] Knuplesch, David and Reichert, Manfred U. and Mangler, Jürgen and Rinderle-Ma, Stefanie B. and Fdhila, Walid. *Towards Compliance of Cross-Organizational Processes and their Changes*. In *1st Int Workshop on Security in Business Processes (SBP'12), BPM'12 Workshops*, LNBIP. Springer, September 2012. URL: http://dbis.eprints.uni-ulm.de/848/.
(Cited on pages 4, 7, and 8)

[93] Kochukuttan, H. and Chandrasekaran, A. *Development of a Fuzzy Expert System for Power Quality Applications*. In *Proceedings of the XXIX Southeastern Symposium on System Theory*, pages 239–243, 1997.
(Cited on page 77)

[94] Kokash, Natallia and Krause, Christian and de Vink, Erik P. *Time and data-aware analysis of graphical service models in REO*. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 125–134. IEEE, 2010.
(Cited on pages 8 and 10)

[95] Kowalski, Robert A. *Database updates in the event calculus*. *The Journal of Logic Programming*, 12(1-2):121–146, 1992.
(Cited on page 26)

[96] Kowalski, Robert A. *A logic-based approach to conflict resolution*. Technical report, Department of Computing, Imperial College London, 2003. URL: http://www.doc.ic.ac.uk/~rak/papers/conflictresolution.pdf.
(Cited on page 156)

[97] Kowalski, Robert A. and Sadri, Fariba. *Towards a Logic-based Production System Language*. Technical report, Department of Computing, Imperial College London, 2010. URL: http://www.doc.ic.ac.uk/~fs/Papers/LPS%20technical%20submission.pdf.
(Cited on page 154)

[98] Kowalski, Robert A. and Sergot, Marek J. *A logic-based calculus of events*. *New generation computing*, 4(1):67–95, 1986.
(Cited on pages 22 and 26)

[99] Krishnapuram, Raghu and Keller, James M. *Fuzzy set theoretic approach to computer vision: An overview*. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 135–142. IEEE, 1992.
(Cited on page 171)

[100] Larsen, Henrik Legind and Yager, Ronald R. *The use of fuzzy relational thesauri for classificatory problem solving in information retrieval and expert systems*. *IEEE Transactions on Systems, Man and Cybernetics*, 23(1):31–41, 1993.
(Cited on page 171)

[101] Lee, Chuen-Chien Chien. *Fuzzy logic in control systems: fuzzy logic controller*. *IEEE Transactions on Systems, Man and Cybernetics*, 20 (2):404–418, 1990.
(Cited on page 85)

[102] Lefèvre, Claire and Nicolas, Pascal. *A first order forward-chaining approach for Answer Set Computing*. *Logic Programming and Nonmonotonic Reasoning*, pages 196–208, 2009.
(Cited on page 149)

[103] Lefèvre, Claire and Nicolas, Pascal. *The first version of a new ASP solver: ASPeRiX*. *Logic Programming and Nonmonotonic Reasoning*, pages 522–527, 2009.
(Cited on page 149)

[104] Liu, Duen-Ren and Shen, Minxin. *Business-to-business workflow interoperation based on process-views*. *Decision Support Systems*, 38 (3):399–419, 2004.
(Cited on page 8)

[105] Liu, Ying and Müller, Samuel and Xu, Ke. *A static compliance-checking framework for business process models*. *IBM Systems Journal*, 46(2):335–361, 2007.
(Cited on pages 8 and 10)

[106] Lloyd, John W. *Foundations of logic programming*. Springer-Verlag, Berlin New York, 1984. ISBN: 3540132996.
(Cited on page 154)

[107] Lloyd, John W. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987. ISBN: 3-540-18199-7.
(Cited on page 185)

[108] Lloyd, John W. *Practical advantages of declarative programming*. In *Joint Conference on Declarative Programming, GULP-PRODE*, volume 94, page 94, 1994.
(Cited on page 154)

[109] Luckham, David C. *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley, Boston, 2002. ISBN: 0201727897.
(Cited on page 164)

[110] Luckham, David C. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, Hoboken, N.J, 2012. ISBN:

0470534850.
(Cited on page 164)

[111] Lukácsy, Gargeli and Szeredi, Péter and Kádár, Balázs. *Prolog based description logic reasoning*. In M. J. G. de la Banda and E. Pontelli, editors, *Proceedings of ICLP 2008 - International Conference on Logic Programming*, volume 5366 of *LNCS*, pages 455–469. Springer, 2008.
(Cited on pages 77 and 79)

[112] Lukasiewicz, Thomas and Straccia, Umberto. *Managing Uncertainty and Vagueness in Description Logics for the Semantic Web*. *Journal of Web Semantics*, 6:291–308, 2008.
(Cited on page 83)

[113] Ly, Linh-Thao and Rinderle-Ma, Stefanie B. and Dadam, Peter. *Integration and verification of semantic constraints in adaptive process management systems*. *Data & Knowledge Engineering*, 64(1):3–23, 2008.
(Cited on pages 8 and 10)

[114] Ly, Linh-Thao and Rinderle-Ma, Stefanie B. and Dadam, Peter. *Design and verification of instantiable compliance rule graphs in process-aware information systems*. In *Advanced Information Systems Engineering*, pages 9–23. Springer, 2010.
(Cited on pages 8 and 10)

[115] Ly, Linh-Thao and Rinderle-Ma, Stefanie B. and Göser, Kevin and Dadam, Peter. *On enabling integrated process compliance with semantic constraints in process management systems*. *Information Systems Frontiers*, 14(2):195–219, 2012.
(Cited on pages 8 and 10)

[116] Ly, Linh-Thao and Rinderle-Ma, Stefanie B. and Knuplesch, David and Dadam, Peter. *Monitoring business process compliance using compliance rule graphs*. *On the Move to Meaningful Internet Systems: OTM 2011*, pages 82–99, 2011.
(Cited on pages 8 and 11)

[117] Maamar, Zakaria and Benslimane, Djamal and Ghedira, Chirine and Mrissa, Michaël. *Views in composite web services*. *Internet Computing, IEEE*, 9(4):79–84, 2005.
(Cited on page 8)

[118] Maggi, Fabrizio Maria and Montali, Marco and Westergaard, Michael and van der Aalst, Wil M.P. *Monitoring business constraints with linear temporal logic: an approach based on colored automata*. *Business Process Management*, pages 132–147, 2011.
(Cited on pages 8 and 11)

[119] Mahbub, Khaled and Spanoudakis, George. *Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience*. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 257–265. IEEE, 2005.
(Cited on page 22)

[120] Martens, Axel. *Consistency between executable and abstract processes*. In *e-Technology, e-Commerce and e-Service, 2005. EEE'05. Proceedings. The 2005 IEEE International Conference on*, pages 60–67. IEEE, 2005.
(Cited on page 8)

[121] Matheus, Christopher J. *Using ontology-based rules for situation awareness and information fusion*. *W3C Work. on Rule Languages for Interoperability*, 2005.
(Cited on page 171)

[122] McCarthy, John. *Applications of circumscription to formalizing common-sense knowledge*. *Artificial Intelligence*, 28(1):89–116, 1986.
(Cited on page 23)

[123] McCarthy, John and Hayes, Patrick J. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.
(Cited on page 22)

[124] Mello, Paola and Proctor, Mark and Sottara, Davide. *A Configurable RETE-OO Engine for Reasoning with different types of Imperfect Information*. *IEEE TKDE - Special Issue on Rule Representation, Interchange and Reasoning in Distributed, Heterogeneous Environments*, 2010.
(Cited on page 78)

[125] Miller, Robert Simon and Shanahan, Murray. *The event calculus in classical logic-alternative axiomatizations*. *Electronic Transactions on Artificial Intelligence*, 4, 1999.
(Cited on page 26)

[126] Motik, Boris. *Reasoning in description logics using resolution and deductive databases*. *PhD theis, University Karlsruhe, Germany*, 2006.
(Cited on pages 77 and 79)

[127] Motik, Boris and Vrandečić, Danny and Hitzler, Pascal and Studer, Rudi. *DLPconvert – converting OWL-DLP statements to logic programs*. In *European Semantic Web Conference 2005 Demos and Posters*. Citeseer, 2005.
(Cited on pages 77 and 79)

[128] Mueller, Erik T. *Commonsense reasoning*. Elsevier Morgan Kaufmann, Amsterdam Boston, 2006. ISBN: 9780123693884.
(Cited on page 27)

[129] Mueller, Erik T. *Event calculus*. *Foundations of Artificial Intelligence*, 3:671–708, 2008.
(Cited on page 26)

[130] Namiri, Kioumars and Stojanovic, Nenad. *Pattern-based design and validation of business process compliance*. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 59–76, 2007.
(Cited on pages 8 and 11)

[131] Novák, Vilém. *Mathematical Fuzzy Logic in Narrow and Broader Sense — a Unified Concept*. In *BISCSE 2005. The Berkeley Initiative in Soft Computing*. University of California, Berkley, 2005.
(Cited on pages 78 and 85)

[132] Özgür, Nezihe Burcu and Koyuncu, Murat and Yazici, Adnan. *An intelligent fuzzy object-oriented database framework for video database applications*. *Fuzzy Sets and Systems*, 160(15):2253–2274, 2009.
(Cited on page 77)

[133] Papazoglou, Michael P. *The challenges of service evolution*. In *Advanced Information Systems Engineering*, pages 1–15. Springer, 2008.
(Cited on pages 8 and 10)

[134] Pesic, Maja and Schonenberg, Helen and van der Aalst, Wil M.P. *Declare: Full support for loosely-structured processes*. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 287–287. IEEE, 2007.
(Cited on pages 8, 9, and 11)

[135] Poole, David. *The Independent Choice Logic for modelling multiple agents under uncertainty*. *Artificial Intelligence*, 94(1):7–56, 1997.
(Cited on page 184)

[136] Reichert, Manfred U. and Bauer, Thomas. *Supporting ad-hoc changes in distributed workflow management systems*. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 150–168, 2007.
(Cited on pages 8 and 9)

[137] Reichert, Manfred U. and Dadam, Peter. *ADEPT$_{flex}$ – supporting dynamic changes of workflows without losing control*. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
(Cited on pages 8 and 9)

[138] Reichert, Manfred U. and Dadam, Peter and Bauer, Thomas. *Dealing with forward and backward jumps in workflow management systems*. *Software and Systems Modeling*, 2(1):37–58, 2003.
(Cited on pages 8 and 9)

[139] Reichert, Manfred U. and Rinderle-Ma, Stefanie B. and Kreher, Ulrich and Dadam, Peter. *Adaptive Process Management with ADEPT2 (Tool Demo)*. In *Proceedings 21st International Conference on Data Engineering (ICDE 2005)*, pages 1113–1114, Los Alamitos, CA, USA, 2005. IEEE Computer Society. URL: http://doc.utwente.nl/53738/.
(Cited on pages 8 and 9)

[140] Reichert, Manfred U. and Weber, Barbara. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. 2012.
(Cited on page 8)

[141] Reiter, Raymond. *The frame problem in the Situation Calculus: A simple solution (sometimes) and a completeness result for goal regression*. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, 27:359–380, 1991.
(Cited on page 23)

[142] Riguzzi, Fabrizio. *ALLPAD: Approximate Learning of Logic Programs with Annotated Disjunctions*. Technical Report CS-2006-01, University of Ferrara, 2006. URL: http://www.ing.unife.it/aree_ricerca/informazione/cs/technical_reports/CS-2006-01.pdf.
(Cited on page 184)

[143] Riguzzi, Fabrizio. *ALLPAD: Approximate Learning of Logic Programs with Annotated Disjunctions*. In *Proceedings of the 16th International Conference on Inductive Logic Programming*, number 4455 in LNAI. Springer, 2007.
(Cited on page 184)

[144] Riguzzi, Fabrizio. *A Top Down Interpreter for LPAD and CP-logic*. In *Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNAI*, pages 109–120. Springer, 2007. URL: http://dx.medra.org/10.1007/978-3-540-74782-6_11.
(Cited on page 186)

[145] Riguzzi, Fabrizio. *Inference with Logic Programs with Annotated Disjunctions under the well founded semantics*. *Logic Programming*, pages 667–671, 2008.
(Cited on page 186)

[146] Riguzzi, Fabrizio. *SLGAD resolution for inference on Logic Programs with Annotated Disjunctions*. *Fundamenta Informaticae*, 102

(3):429–466, 2010.
(Cited on page 186)

[147] Riguzzi, Fabrizio. *MCINTYRE: A Monte Carlo Algorithm for Probabilistic Logic Programming*. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC2011), Pescara, Italy, 31 August-2 September, 2011*, pages 25–39, 2011. URL: http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/Rig-CILC11.pdf.
(Cited on page 186)

[148] Riguzzi, Fabrizio and Swift, Terrance. *Tabling and answer subsumption for reasoning on Logic Programs with Annotated Disjunctions*. In *Technical Communications of the 26th International Conference on Logic Programming*, volume 7, pages 162–171. Citeseer, 2010.
(Cited on page 186)

[149] Rinderle-Ma, Stefanie B. *Schema evolution in process management systems*. PhD thesis, University of Ulm, 2004.
(Cited on pages 8 and 9)

[150] Rinderle-Ma, Stefanie B. and Reichert, Manfred U. and Dadam, Peter. *Correctness criteria for dynamic changes in workflow systems—-a survey*. *Data & Knowledge Engineering*, 50(1):9–34, 2004.
(Cited on pages 8 and 9)

[151] Rinderle-Ma, Stefanie B. and Reichert, Manfred U. and Dadam, Peter. *Flexible support of team processes by adaptive workflow systems*. *Distributed and Parallel Databases*, 16(1):91–116, 2004.
(Cited on pages 8 and 9)

[152] Rouached, Mohsen and Fdhila, Walid and Godart, Claude. *A semantical framework to engineering WSBPEL processes*. *Information Systems and E-Business Management*, 7(2):223–250, 2009.
(Cited on page 22)

[153] Rozinat, Anne and de Jong, Ivo S.M. and Günther, Christian W. and van der Aalst, Wil M.P. *Conformance Analysis of ASML's Test Process*. In *Proceedings of the Second International Workshop on Governance, Risk and Compliance (GRCIS'09)*, volume 459, pages 1–15, 2009.
(Cited on page 5)

[154] Rozinat, Anne and van der Aalst, Wil M.P. *Conformance checking of processes based on monitoring real behavior*. *Information Systems*, 33(1):64–95, 2008.
(Cited on page 5)

[155] Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 2nd

edition, 2003.
(Cited on page 154)

[156] Sadiq, Shazia Wasim and Governatori, Guido and Namiri, Kioumars. *Modeling control objectives for business process compliance*. *Business process management*, pages 149–164, 2007.
(Cited on page 93)

[157] Sadiq, Shazia Wasim and Sadiq, Wasim and Orlowska, Maria Elzbieta. *A framework for constraint specification and validation in flexible workflows*. *Information Systems*, 30(5):349–378, 2005.
(Cited on pages 8 and 9)

[158] Sadri, Fariba and Kowalski, Robert A. *Variants of the event calculus*. In *Proceedings of International Conference on Logic Programming, ICLP 1995*, volume 95, pages 67–82, 1995.
(Cited on pages 25 and 26)

[159] Sato, Taisuke and Kameya, Yoshitaka. *PRISM: A Language for Symbolic-Statistical Modeling*. In *International Joint Conference on Artificial Intelligence*, pages 1330–1339. Morgan Kaufmann, 1997.
(Cited on page 184)

[160] Schmidt, Kay-Uwe and Stühmer, Roland and Stojanovic, Ljiljana. *Blending complex event processing with the RETE algorithm*. In *iCEP2008: 1st International workshop on Complex Event Processing for the Future Internet colocated with the Future Internet Symposium (FIS2008)*. CEUR Workshop Proceedings (CEUR-WS.org), 2008.
(Cited on page 95)

[161] Scott, David S. and Krauss, Peter H. *Assigning probabilities to logical formulas*. *Aspects of inductive logic*, 43:219–264, 1966.
(Cited on page 184)

[162] Shanahan, Murray. *Robotics and the Common Sense Informatic Situation*. In *ECAI 96. 12th European Conference on Artificial Intelligence*, pages 684–688. Wiley, 1996.
(Cited on page 22)

[163] Shanahan, Murray. *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. The MIT Press, 1997.
(Cited on page 23)

[164] Shanahan, Murray. *The Event Calculus explained*. *Artificial intelligence today*, pages 409–430, 1999.
(Cited on pages 25, 26, and 31)

[165] Shanahan, Murray. *An abductive event calculus planner*. *The Journal of Logic Programming*, 44(1-3):207–240, 2000.
(Cited on page 22)

[166] Simon, Herbert A. *The MIT Encyclopedia of the Cognitive Sciences*, chapter Production Systems. The MIT Press, 1999. Wilson, Robert A. and Keil, Frank C.
(Cited on page 154)

[167] Singh, Munindar P. and Chopra, Amit K. and Desai, Nirmit. *Commitment-Based Service-Oriented Architecture*. *IEEE Computer*, 42(11):72–79, 2009.
(Cited on page 93)

[168] Sottara, Davide. *Integration of symbolic and connectionist AI techniques in the development of Decision Support Systems applied to biochemical processes*. PhD thesis, PhD School in Electronics, Computer Science and Telecommunications, University of Bologna, 2010.
(Cited on pages 131 and 157)

[169] Sottara, Davide and Mello, Paola and Sartori, Claudio and Fry, Emory. *Enhancing a production rule engine with predictive models using PMML*. In *Proceedings of the 2011 workshop on Predictive markup language modeling*, pages 39–47. Association for Computing Machinery (ACM), 2011.
(Cited on pages 112 and 117)

[170] Stamou, Giorgios B. and Tzouvaras, Vassilis and Pan, Jeff Z. and Horrocks, Ian. *A fuzzy extension of SWRL*. In *W3C Workshop on Rule Languages for Interoperability*. Citeseer, 2005.
(Cited on page 171)

[171] Stoilos, Giorgios and Stamou, Giorgios B. and Tzouvaras, Vassilis and Pan, Jeff Z.and Horrocks, Ian. *A fuzzy description logic for multimedia knowledge representation*. In *Proceedings of the International Workshop on Multimedia and the Semantic Web*, pages 12–19, 2005.
(Cited on page 171)

[172] Straccia, Umberto. *Reasoning within Fuzzy Description Logics*. *JAIR*, 14:137–166, 2001.
(Cited on page 83)

[173] Straccia, Umberto and Bobillo, Fernando. *fuzzyDL: An expressive fuzzy description logic reasoner*. In *Proceedings of the International Conference on Fuzzy Systems (FUZZ-08)*, 2008.
(Cited on pages 77, 78, and 83)

[174] Tata, Samir and Klai, Kais and Ould Ahmed M'Bareck, Nomane. *CoopFlow: a bottom-up approach to workflow cooperation for short-term virtual enterprises*. *Services Computing, IEEE Transactions on*, 1(4):214–228, 2008.
(Cited on page 8)

[175] TEN TEIJE, ANNETTE AND MIKSCH, SILVIA AND LUCAS, PETER J.F. *Computer-based medical guidelines and protocols: a primer and current trends*, volume 139. IOS Press, 2008.
(Cited on page 22)

[176] THAGARD, PAUL. *Mind: Introduction to cognitive science*. The MIT Press, 2nd edition, 2005.
(Cited on page 154)

[177] TORRONI, PAOLO AND CHESANI, FEDERICO AND MELLO, PAOLA AND MONTALI, MARCO. *Social Commitments in Time: Satisfied or Compensated*. In M. Baldoni, J. Bentahar, M. B. van Riemsdijk, and J. W. Lloyd, editors, *DALT*, volume 5948 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2009. ISBN: 978-3-642-11354-3.
(Cited on page 105)

[178] TORRONI, PAOLO AND YOLUM, PINAR AND SINGH, MUNINDAR P. AND ALBERTI, MARCO AND CHESANI, FEDERICO AND GAVANELLI, MARCO AND LAMMA, EVELINA AND MELLO, PAOLA. *Modelling interactions via commitments and expectations*. *Handbook of research on multi-agent systems: Semantics and dynamics of organizational models*, pages 263–284, 2009.
(Cited on page 93)

[179] TURING, ALAN MATHISON. *Computing machinery and intelligence*. *Mind*, 59(236):433–460, 1950.
(Cited on page 1)

[180] UROVI, VISARA AND STATHIS, KOSTAS. *Playing with agent coordination patterns in MAGE*. In *Proceedings of the 5th international conference on Coordination, organizations, institutions, and norms in agent systems*, COIN'09, pages 86–101, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN: 3-642-14961-8, 978-3-642-14961-0. URL: http://dl.acm.org/citation.cfm?id=1886594.1886601.
(Cited on page 3)

[181] VALIANT, LESLIE G. *The complexity of computing the permanent*. *Theoretical computer science*, 8(2):189–201, 1979.
(Cited on page 184)

[182] VAN DER AALST, WIL M.P. *Inheritance of interorganizational workflows to enable business-to-business e-commerce*. *Electronic commerce research*, 2(3):195–231, 2002.
(Cited on page 8)

[183] VAN DER AALST, WIL M.P. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
(Cited on page 5)

[184] VAN DER AALST, WIL M.P. AND DE BEER, HUUB T. AND VAN DONGEN, BOUDEWIJN F. *Process mining and verification of properties: An approach based on temporal logic*. *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 130–147, 2005.
(Cited on pages 8 and 11)

[185] VAN DER AALST, WIL M.P. AND LOHMANN, NIELS AND MASSUTHE, PETER AND STAHL, CHRISTIAN AND WOLF, KARSTEN. *Multiparty contracts: Agreeing and implementing interorganizational processes*. *The Computer Journal*, 53(1):90–106, 2010.
(Cited on page 8)

[186] VAN DER AALST, WIL M.P. AND PESIC, MAJA AND MONTALI, MARCO AND CHESANI, FEDERICO AND MELLO, PAOLA AND STORARI, SERGIO. *Declarative specification and verification of service choreographiess*. *TWEB*, 4(1), 2010.
(Cited on page 93)

[187] VENNEKENS, JOOST AND VERBAETEN, SOFIE. *Logic Programs with Annotated Disjunctions*. CW Reports CW368, Department of Computer Science, K.U. Leuven, Leuven, Belgium, Sep 2003. URL: https://lirias.kuleuven.be/handle/123456789/131950.
(Cited on pages 184 and 185)

[188] VENNEKENS, JOOST AND VERBAETEN, SOFIE AND BRUYNOOGHE, MAURICE. *Logic Programs with Annotated Disjunctions*. *Lecture Notes in Computer Science*, 3132:431–445, 2004. URL: https://lirias.kuleuven.be/handle/123456789/124869.
(Cited on pages 184 and 185)

[189] WEBER, BARBARA AND REICHERT, MANFRED U. AND RINDERLE-MA, STEFANIE B. *Change patterns and change support features–enhancing flexibility in process-aware information systems*. *Data & knowledge engineering*, 66(3):438–466, 2008.
(Cited on pages 8 and 9)

[190] WEBER, BARBARA AND REICHERT, MANFRED U. AND RINDERLE-MA, STEFANIE B. AND WILD, WERNER. *Providing integrated life cycle support in process-aware information systems*. *International Journal of Cooperative Information Systems*, 18(01):115–165, 2009.
(Cited on pages 8 and 9)

[191] WEBER, BARBARA AND REICHERT, MANFRED U. AND WILD, WERNER AND RINDERLE-MA, STEFANIE B. *Balancing flexibility and security in adaptive process management systems*. *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 59–76, 2005.
(Cited on pages 8 and 9)

[192] Weber, Barbara and Sadiq, Shazia Wasim and Reichert, Manfred U. *Beyond rigidity-dynamic process lifecycle support*. *Computer Science-Research and Development*, 23(2):47–65, 2009.
(Cited on pages 8 and 9)

[193] Weber, Ingo and Hoffmann, Jörg and Mendling, Jan. *Semantic business process validation*. In *Proceedings of International workshop on Semantic Business Process Management*, 2008.
(Cited on pages 8 and 10)

[194] Weidlich, Matthias and Polyvyanyy, Artem and Desai, Nirmit and Mendling, Jan and Weske, Mathias. *Process compliance analysis based on behavioural profiles*. *Information Systems*, 36(7):1009–1025, 2011.
(Cited on page 5)

[195] Weiss, Gerhard. *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT press, 1999.
(Cited on page 22)

[196] Weske, Mathias. *Business process management: concepts, languages, architectures*. Springer, 2010.
(Cited on page 22)

[197] Yoneki, Eiko and Bacon, Jean. *Unified semantics for event correlation over time and space in hybrid network environments*. *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 366–384, 2005.
(Cited on page 168)

[198] Zaha, Johannes Maria and Barros, Alistair and Dumas, Marlon and ter Hofstede, Arthur H.M. *Let's dance: A language for service behavior modeling*. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 145–162, 2006.
(Cited on page 8)

[199] Zhang, Lei and Yu, Yong and Zhou, Jian and Lin, Chen-Xi and Yang, Yin. *An enhanced model for searching in semantic portals*. In *Proceedings of the 14th international conference on World Wide Web*, pages 453–462. Association for Computing Machinery (ACM), 2005.
(Cited on page 171)

[200] Zimmermann, Hans-Jürgen. *Fuzzy sets, decision making and expert systems*, volume 10. Springer, 1987.
(Cited on page 171)

COLOPHON

This document was typeset using the typographical look-and-feel `arsclassica` developed by Lorenzo Pantieri. This style changes some typographical features of `classicthesis` developed by André Miede which was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". It includes semi-transparent headlines separated from the page number by a small rule and itemise lists with semi-transparent bullets.

It uses Herman Zapf's *Palatino* and *Euler* type faces as default font and math font respectively. Figures from the *Euler* font are also used for chapter numbers. The *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera", is used as typewriter font and for listings. The *Iwona* font by Janusz M. Nowacki for the bold face, the titles of the sectioning units of the document, the short labels of the acronyms, the labels of description lists, the headlines and the bold label of the captions.

Final Version as of March 14, 2013 (version 1.0.3)