

Constraint-based workflow management systems : shifting control to users

Citation for published version (APA):

Pesic, M. (2008). *Constraint-based workflow management systems : shifting control to users*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Industrial Engineering and Innovation Sciences]. Technische Universiteit Eindhoven. <https://doi.org/10.6100/IR638413>

DOI:

[10.6100/IR638413](https://doi.org/10.6100/IR638413)

Document status and date:

Published: 01/01/2008

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

**Constraint-Based
Workflow Management Systems:
Shifting Control to Users**

Copyright © 2008 by Maja Pešić. All Rights Reserved.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Pešić, Maja

Constraint-Based Workflow Management Systems: Shifting Control to Users / by Maja Pešić.

- Eindhoven: Technische Universiteit Eindhoven, 2008. - Proefschrift. -

ISBN 978-90-386-1319-2

NUR 982

Keywords: Workflow Management Systems / Business Process Management / Flexibility / Declarative Process Models / Constraint-Based Systems / Socio-Technical Systems

The work in this thesis has been carried out under the auspices of Beta Research School for Operations Management and Logistics.

Beta Dissertation Series D106

Printed by University Press Facilities, Eindhoven

Constraint-Based Workflow Management Systems: Shifting Control to Users

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 8 oktober 2008 om 16.00 uur

door

Maja Pešić

geboren te Belgrado, Servië

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. W.M.P. van der Aalst

Copromotor:

dr. F.M. van Eijnatten

To Boris.

Contents

1	Introduction	1
1.1	Business Processes Management	2
1.2	Characterization of Business Processes	4
1.3	Characterization of Decision Making	6
1.3.1	Adjusting to the Environment	8
1.3.2	Combining Social and Technical Aspects	9
1.4	The Tradeoff between Flexibility and Support	11
1.5	Problem Definition and Research Goal	11
1.6	Contributions	13
1.6.1	Constraint-Based Process Models	13
1.6.2	Generic Constraint-Based Process Modeling Language	15
1.6.3	A Prototype of a Constraint-Based Workflow Management System	16
1.6.4	Constraint-Based Approach in the BPM Life Cycle	16
1.6.5	Combining Traditional and Constraint-Based Approach	17
1.7	Road Map	18
2	Related Work	19
2.1	Workflow Flexibility	19
2.1.1	Taxonomy of Flexibility by Heinel et al.	21
2.1.2	Taxonomy of Flexibility by Schonenberg et al.	21
2.1.3	Flexibility by Design	23
2.1.4	Flexibility by Underspecification	26
2.1.5	Flexibility by Change	28
2.1.6	Flexibility by Deviation	30
2.2	Workflow Management Systems	32
2.2.1	Staffware	33
2.2.2	FLOWer	35
2.2.3	YAWL	37
2.2.4	ADEPT	38
2.2.5	Other Systems	40
2.3	Workflow Management Systems and the Organization of Human Work	41
2.3.1	Two Contrasting Regimes for the Organization of Work	41
2.3.2	Socio-Technical Systems	42
2.3.3	Workflow Management Systems and the Structural Parameters	42
2.3.4	Summary	45
2.4	Outlook	45

3	Flexibility of Workflow Management Systems	47
3.1	Contemporary Workflow Management Systems	47
3.1.1	The Control-Flow Perspective	51
3.1.2	The Resource Perspective	56
3.1.3	The Data Perspective	66
3.1.4	Summary	68
3.2	Taxonomy of Flexibility	69
3.2.1	Flexibility by Design	69
3.2.2	Flexibility by Underspecification	72
3.2.3	Flexibility by Change	74
3.2.4	Flexibility by Deviation	76
3.2.5	Summary	77
3.3	A New Approach for Full Flexibility	79
4	Constraint-Based Approach	83
4.1	Activities, Events, Traces and Constraints	84
4.2	Constraint Models	89
4.3	Illustrative Example: The Fractures Treatment Process	94
4.4	Execution of Constraint Model Instances	98
4.4.1	Instance State	99
4.4.2	Enabled Events	101
4.4.3	States of Constraints	103
4.5	Ad-hoc Instance Change	106
4.6	Verification of Constraint Models	109
4.6.1	Dead Events	110
4.6.2	Conflicts	112
4.6.3	Compatibility of Models	114
4.7	Summary	117
5	Constraint Specification with Linear Temporal Logic	119
5.1	LTL for Business Process Models	119
5.2	ConDec: An Example of an LTL-Based Constraint Language	123
5.2.1	Existence Templates	125
5.2.2	Relation Templates	126
5.2.3	Negation Templates	129
5.2.4	Choice Templates	130
5.2.5	Branching of Templates	133
5.3	ConDec Constraints	134
5.3.1	Adjusting to Properties of Business Processes	136
5.3.2	Dealing with the Non-Determinism	137
5.3.3	Retrieving the Set of Satisfying Traces	139
5.4	ConDec Models	141
5.5	ConDec Model: Fractures Treatment Process	144
5.6	Execution of ConDec Instances	146
5.6.1	Instance State	146
5.6.2	Enabled Events	149
5.6.3	States of Constraints	150
5.7	Ad-hoc Change of ConDec Instances	150
5.8	Verification of ConDec Models	152
5.9	Activity Life Cycle and ConDec	155
5.9.1	Possible Problems	155
5.9.2	Available Solutions	157
5.10	Summary	160

6	DECLARE: Prototype of a Constraint-Based System	163
6.1	System Architecture	163
6.2	Constraint Templates	165
6.3	Constraint Models	167
6.4	Execution of Instances	168
6.5	Ad-hoc Change of Instances	171
6.6	Verification of Constraint Models	173
6.7	The Resource Perspective	175
6.8	The Data Perspective	178
6.9	Conditional Constraints	180
6.10	Defining Other Languages	184
6.10.1	Languages Based on LTL	184
6.10.2	Languages Based on Other Formalizations	185
6.11	Combining the Constraint-Based and Procedural Approach	187
6.11.1	Decomposition of DECLARE and YAWL Processes	188
6.11.2	Dynamic Decompositions	192
6.11.3	Integration of Even More Approaches	194
6.12	Summary	195
7	Using Process Mining for the Constraint-Based Approach	197
7.1	Process Mining with the ProM Framework	198
7.2	Verification of Event Logs with LTL Checker	201
7.2.1	The Default LTL Checker	203
7.2.2	Combining the LTL Checker and DECLARE	205
7.3	The SCIFF Language	207
7.3.1	Verification of Event Logs with SCIFF Checker	210
7.3.2	Discovering Constraints with DecMiner	211
7.4	Recommendations Based on Past Executions	212
7.5	Summary	216
8	Conclusions	219
8.1	Evaluation of the Research Goal	219
8.2	Contributions	220
8.2.1	Flexibility of the Constraint-Based Approach	220
8.2.2	Support of the Constraint-Based Approach	222
8.2.3	The Constraint-Based Approach and Organization of Human Work	223
8.2.4	Combining the Constraint-Based Approach with Other Approaches	226
8.2.5	Business Process Management with the Constraint-Based Approach	226
8.3	Limitations	227
8.3.1	Complexity of Constraint-Based Models	227
8.3.2	Evaluation of the Approach	229
8.4	Directions for Future Work	230
8.5	Summary	231
	Appendices	232
A	Work Distribution in Staffware, FileNet and FLOWer	233
A.1	Staffware	233
A.1.1	Work Queues	234
A.1.2	Resource Allocation	236
A.1.3	Forward and Suspend	238
A.2	FileNet	241

A.2.1	Queues	241
A.2.2	Resource Allocation	242
A.2.3	Forward and Suspend	244
A.3	FLOWer	244
A.3.1	Case Handling	245
A.3.2	Authorization Rights	245
A.3.3	Distribution Rights	245
A.3.4	Distribution of Instances	246
A.3.5	Distribution within an Instance	249
A.4	Summary	255
B	Evaluation of Workflow Patterns Support	257
B.1	Control-Flow Patterns	258
B.2	Resource Patterns	259
B.3	Data Patterns	260
	Bibliography	261
	Summary	279
	Samenvatting	283
	Acknowledgements	287
	Curriculum Vitae	289

Chapter 1

Introduction

An organization produces value for its customers by executing various business processes. Due to complexity and variety of business processes, contemporary organizations use information technology to support activities and possibly also automate their processes. *Business Process Management* systems (or BPM systems) are software systems used for automation of business processes. Once a BPM system is employed in a company, it has a significant influence on the way business processes are executed in the company. Contemporary BPM systems tend to *determine* the way companies organize work and force companies to adjust their business processes to the system. In other words, a company that uses a BPM system is not likely to be able to implement its business processes in the way that is most appropriate for the company. Instead, business processes must be implemented such that they ‘fit the system’, which can cause various problems. First, due to a mismatch between the preferred way of work and the system’s way of work, companies may be forced to ‘run’ inappropriate business processes. Second, two parallel realities may be created: the actual work is done ‘outside the system’ in one way, and later registered in the system in another way. These problems may prevent a company from using a BPM system.

In this chapter we introduce the research presented in this thesis, which aims at enabling a better alignment of BPM systems with business processes in companies. We start by introducing business processes and BPM systems in Section 1.1. The nature of contemporary business processes is described in Section 1.2. Section 1.3 describes the way today’s organizations manage their work. The tradeoff between flexibility and support in BPM systems is shortly discussed in Section 1.4. Section 1.5 defines the problem addressed by this research and the research goal. Finally, a short overview of research contributions is given in Section 1.6 and the outline of the thesis is provided in Section 1.7.

1.1 Business Processes Management

A business process defines a specific *ordering of activities* that are executed by employees, available input and required output, and the flow of information. Business Process Management (BPM) is a method to continuously improve business processes in order to achieve better results. BPM includes concepts, methods and techniques to support the design, implementation, enactment and diagnosis of business processes [93]. Figure 1.1 shows the BPM life cycle as a continuous cycle consisting of four phases.

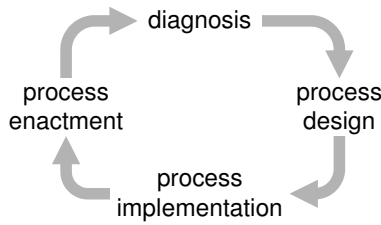


Figure 1.1: BPM life cycle [93]

The BPM life cycle starts with *process design*, where the business processes are identified, reviewed, validated and finally represented as process models [266]. A *process model* describes (a part of) a business process by defining how documents, information, and activities are passed from one participant to another [93, 266]. Process models are developed using a process modeling language, e.g., Business Process Modeling Notation (BPMN), Business Process Execution Language (BPEL), Unified Modeling Language (UML), Event-driven Process Chains (EPCs), etc. In some cases, process models can be *verified* against inconsistencies and errors [89, 254]. Next, the process model is *implemented* in order to align work of the employees with the prescribed process model. In the *process enactment* phase, the business process should be executed within the organization in the way prescribed in the implemented process model. The *process diagnosis* phase uses information about the actual enactment of processes in order to evaluate them. The results from the diagnosis phase are used to close the BPM life cycle in order to continuously improve business processes, i.e., based on the diagnosis, the processes are redesigned, etc.

Business processes can be supported by various types of software products. *BPM systems* support collaboration, coordination and decision making in business processes [93, 110, 266]. Various BPM systems provide for different degrees of automation of ordering and coordination of activities. Figure 1.2 shows two extreme types of BPM systems: *groupware systems* and *workflow management systems*.

Groupware systems focus on supporting human collaboration, and co-decision. Ordering and coordination of activities in these systems *cannot* be

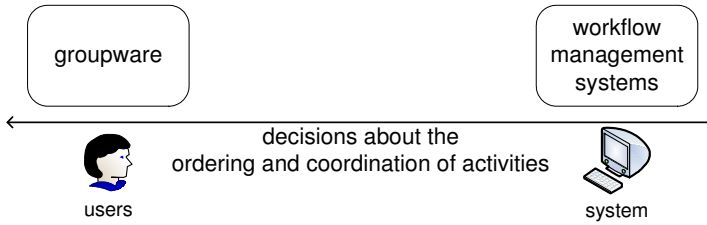


Figure 1.2: BPM systems

automated [110]. Instead, users of groupware control the ordering and coordination of activities while executing the business process (i.e., ‘on the fly’) [110]. Groupware systems range from ‘enhanced’ electronic mail to group conferencing systems.

Workflow management systems focus on the business process by explicitly controlling ordering, coordination and execution of activities with possibly little human intervention [110]. In general, humans merely influence the execution of business processes by entering necessary data. A workflow management system automates a set of business processes by the definition and execution of process models [93, 266]. Moreover, most contemporary systems support three phases of the BPM cycle, as shown Figure 1.3. First, *process design* is conducted by defining process models, which define (1) the execution order of activities, (2) which employees are allowed to execute which activities, and (3) which information will be available during the execution. In addition, in some systems it is possible to verify models against errors. Second, process models are *implemented* thus allowing for the automatic *enactment* of process instances in the system. A process model can be seen as a template that workflow management systems use for execution of concrete process instances. Thus, by executing process models, workflow management systems determine in which order activities can be executed, which employee executes which activity and which information is available.

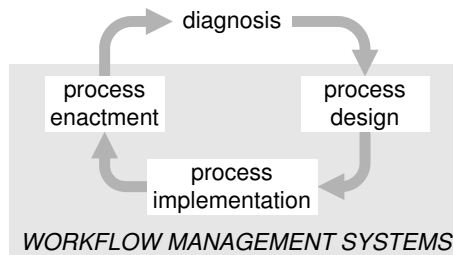


Figure 1.3: Workflow management systems and the BPM life cycle presented in Figure 1.1

1.2 Characterization of Business Processes

Not all business processes are the same. Even within one organization, business processes can be very different in terms of their essential properties. Business processes can be characterized based on various properties [110]. For example, the nature of the business process depends on its *complexity*, *predictability* and *repetitiveness*.

The *complexity* of a business process refers to the complexity of collaboration, coordination, and decision making [110]. The more complex collaboration, coordination, and decision making in the business process are, the higher the degree of complexity of the process is. Figure 1.4 shows examples of several business processes with various degrees of complexity. Simple business processes (e.g., exchanging personal email messages and handling travel requests) require trivial collaboration, coordination and decision making. On the other hand, handling medical treatments is a complex business process because it is non-trivial from the view point of collaboration, coordination and decision making.

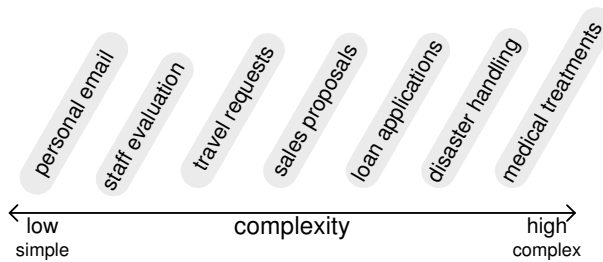


Figure 1.4: Complexity of business processes

The *predictability* of a business process depends on how easy it is to determine in advance the way the process will be executed. The more predictable possible future executions of the business process are, the more predictable the process is. Figure 1.5 shows examples of several business processes with various degrees of predictability. For example, handling travel requests has a high degree of

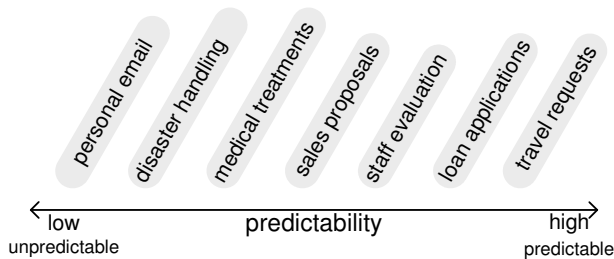


Figure 1.5: Predictability of business processes

predictability because it is quite certain how it will be executed. On the other hand, it is hard to predict how personal email messages can be exchanged, i.e., this business process is unpredictable.

The *repetitiveness* of a business process refers to the frequency of process execution. The more times the business process is executed, the higher degree of the repetitiveness of the process is. For example, a business process that is executed once per year has a lower degree of repetitiveness than a process executed more than a thousand times per year. Figure 1.6 shows examples of several business processes with various degrees of repetitiveness. For example, disaster handling (e.g., floods, earthquakes, etc.) is a business process with a low degree of repetitiveness because it does not happen frequently. On the other hand, exchanging personal email messages is a frequent and, thus, repetitive business process.

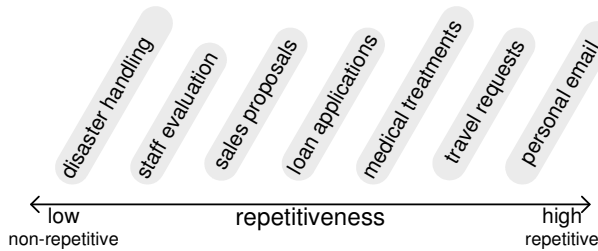


Figure 1.6: Repetitiveness of business processes

Note that one business process can have different degrees of complexity, predictability and repetitiveness. Figure 1.7 shows that the nature of a business process is determined by the degrees of complexity, predictability and repetitiveness.

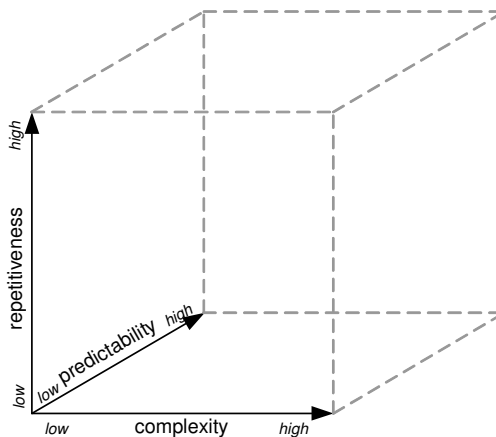


Figure 1.7: Complexity, predictability and repetitiveness determine the nature of business processes

For example, medical treatments are very complex processes with a high degree of repetitiveness and a low degree of predictability (cf. figures 1.4, 1.5 and 1.6).

BPM systems (cf. Section 1.1) *aim* at supporting complex and repetitive processes, as Figure 1.8(a) shows. As described in Section 1.1, there are two extreme types of BPM systems: groupware and workflow management systems. Because in groupware systems users control the ordering and coordination of activities, they are suitable for *unpredictable* processes [110], as shown in Figure 1.8(b). Workflow management systems fully automate the ordering and coordination of activities by executing predefined process models. Therefore, workflow management systems support *highly predictable* business processes [110].

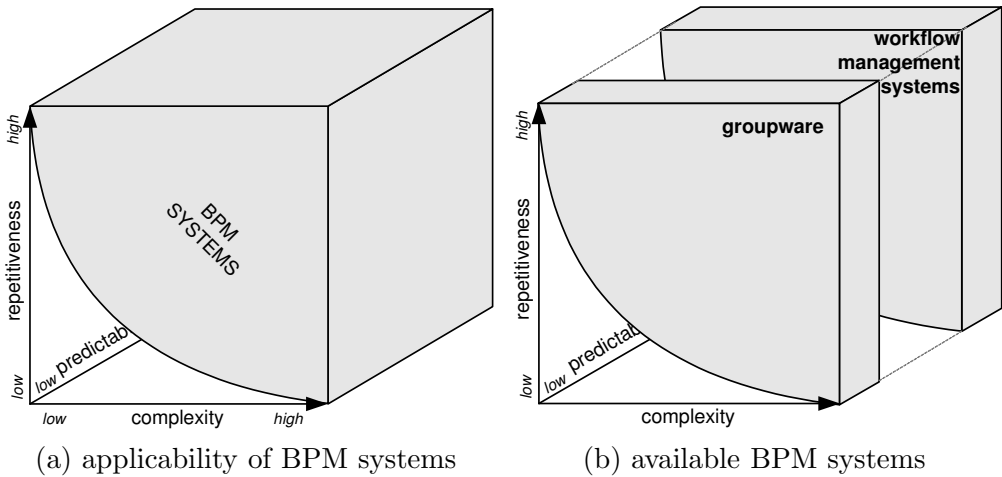


Figure 1.8: Automation of business processes with BPM systems

1.3 Characterization of Decision Making

Decision making determines to a great extent the way people work and influences their productivity. If decisions about how to work are made centrally, then we speak about *centralized* decision making. If the workers who do the work make decisions themselves, then we speak of *local* decision making. At the middle of the twentieth century, schools of organizational science were divided into two groups that propagated two extreme styles of decision making. The so called

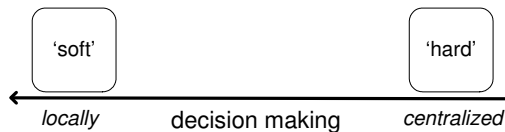


Figure 1.9: Two extreme styles of BPM

‘hard’ approaches propagated centralized decision making, while the so-called ‘soft’ approaches propagated local decision making, as shown in Figure 1.9.

A good illustration of the differences between the two extreme approaches is given by the motivation theory of McGregor: *Theory X* and *Theory Y* [169]. Table 1.1 shows the main principles of the two basic modes.

Table 1.1: Theory X and Theory Y [169]

Theory X (‘hard’ approach)	Theory Y (‘soft’ approach)
Humans inherently dislike working and will try to avoid it if they can.	People view work as being as natural as play and rest. Humans expend the same amount of physical and mental effort in their work as in their private lives.
Because people dislike work they have to be coerced or controlled by management and threatened so they work hard enough.	Provided people are motivated, they will be self-directing to the aims of the organization. Control and punishment are not the only mechanisms to make people work.
Average employees want to be directed.	Job satisfaction is key to engaging employees and ensuring their commitment.
People don’t like responsibility.	People learn to accept and seek responsibility. Average humans, under the proper conditions, will not only accept but even naturally seek responsibility.
Average humans are simple and need security at work.	People are imaginative and creative. Their ingenuity should be used to solve problems at work.

Theory X characterizes authoritarian and repressive ‘hard’ approaches with centralized decision making. This theory takes a pessimistic view on workers, i.e., it is assumed that humans do not like to work, can’t be trusted, and need to be closely supervised and controlled [169]. The result is a limited and depressed culture of work and a constant *decrease* of worker’s motivation and productivity. ‘Hard’ approaches advocate detailed division and specialization of work, and centralized decision making at its extreme [105,114,242,262]. Workers are specialized and prepared for the execution of small and monotonous tasks, and they do not participate in decision making. This way of thinking emerged with the industrialization. It was believed that the automation of business processes increases productivity by minimizing participation of humans and, thus, minimizing human errors and throughput times. A worker was considered to be an extension to the machine, which merely performs tasks that cannot be automatized.

Theory Y can be characterized by liberating and developmental ‘soft’ approaches with local decision making. This theory takes an optimistic view on workers, i.e., it is assumed that humans enjoy working, may be ambitious, self-motivated, anxious to accept greater responsibility, and exercise self-control, self-direction, autonomy and empowerment [169]. ‘Soft’ approaches advocate local-

ized decision making where all relevant decisions about work are made directly by people who actually do the work [168,204]. In this way, workers are in a full control and share responsibility for their work. Satisfaction of doing a good job is a strong motivation and, therefore, will lead to constant *increase* of productivity.

The two extreme approaches to decision making were criticized and mostly abandoned in the second half of the twentieth century. Relying on either a ‘soft’ or ‘hard’ manner seems to represent unrealistic extremes. In reality, companies aim at achieving an *optimal* ratio between local and centralized decision making, depending on the specific situation. Moreover, contemporary companies *commonly* place decision making somewhere between the ‘soft’ and ‘hard’ approach, as Figure 1.10 shows.

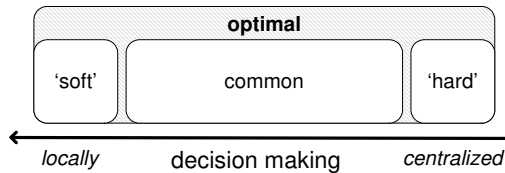


Figure 1.10: Optimal decision making

New approaches consider multiple aspects of business processes. Each organization is seen as a *unique* open system with inputs, transformations, outputs and feedback. Therefore, each company should consider the *influence of the environment* and the *integration of social and technical aspects* of business processes when choosing for the optimal style of decision making [69, 99, 102, 103, 128, 244, 246].

1.3.1 Adjusting to the Environment

Changes in the environment can have a major influence on an organization (e.g., change in customer requirements, appearance of new competitors, etc.). Therefore, business processes must constantly be adjusted to the environment (cf. Section 1.1) [65, 104]. Environments with a low degree of turbulency are *stable* environments, and environments with a high degree of turbulency are *turbulent* environments. The degree of turbulency of the environment influences the predictability of business processes (cf. Figure 1.5 on page 4) and the nature of decision making. The more turbulent the environment is, the more often it will be necessary to adjust business processes to it and the more unpredictable business processes are. For example, medical processes have a low degree of predictability because each treatment must be adjusted to specific environment (e.g., available medications, conditions of the patient, etc.), while handling travel requests has a higher degree of predictability because traveling conditions do not change so frequently.

‘Hard’ and ‘soft’ approaches advocate centralized and local decision making, regardless the nature of the environment, as Figure 1.11(a) shows. However, the need for localized decision making rises with the turbulence of the environment and, thus, the *unpredictability* of business processes, as Figure 1.11(b) shows [65, 104]. In other words, unpredictable business processes require localized decision making because decisions about how to adjust the process to new requirements must be frequently made ‘on the fly’. For example, decisions about how to adjust the medical treatment to the specific patient must be frequently made. Therefore, these decisions should be made by the involved medical staff ‘on the spot’. Because handling travel requests is a predictable process, decisions about how to handle the process can be made ‘outside’ the process, i.e., in a centralized manner.

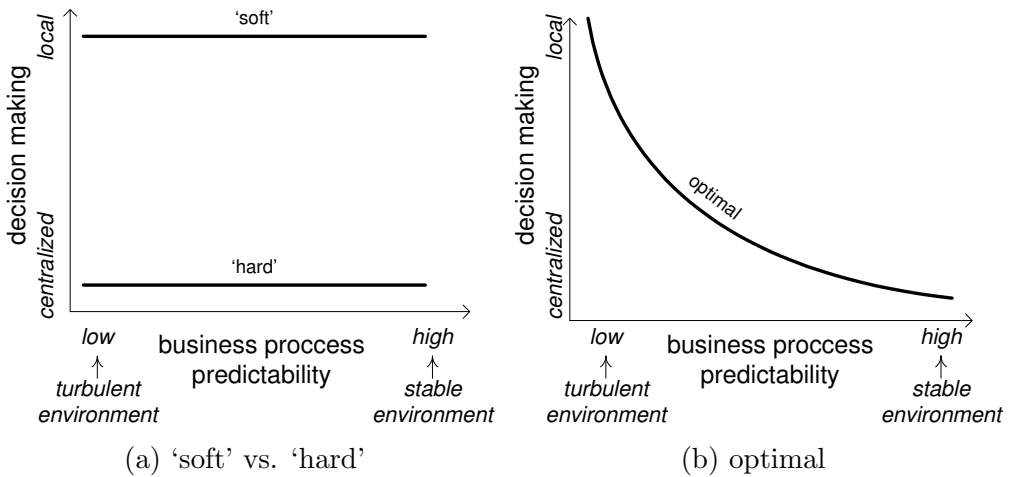


Figure 1.11: Influence of environment on decision making

1.3.2 Combining Social and Technical Aspects

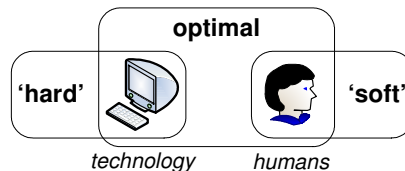
Technology used for the automation of business processes influences the way of work. While ‘hard’ approaches praise the automation for taking over the decision making from workers [105, 114, 242, 262], ‘soft’ approaches see automation as a means for suppressing the motivation and capabilities of people by imposing a centralized decision making [58, 168, 204]. However, organizations can significantly benefit from using the best that both humans and technology have to offer [246]. Table 1.2 shows a list of things that humans can do better than machines and vice versa [96, 134].

Instead of being replaceable, humans and machines should complement one another [142, 224, 229]. Moreover, both technical and social aspects of an organization must be optimized in order to achieve the best results [246]. For example,

Table 1.2: People versus machines [134]

people are better in:	machines are better in:
Detection of certain forms of very low energy levels.	Monitoring (both men and machines).
Sensitivity to an extremely wide variety of stimuli.	Performing routine, repetitive, or very precise operations.
Perceiving patterns and making generalizations about them.	Responding very quickly to control signals.
Ability to store important information for long periods and recalling relevant facts at appropriate moments.	Storing and recalling large amounts of information in long time periods.
Ability to exercise judgment where events cannot be completely defined.	Performing complex and rapid computation with high accuracy.
Improving and adopting flexible procedures.	Sensitivity to stimuli beyond the range of human sensitivity (infrared, radio waves, etc.),
Ability to react to unexpected low-probability events.	Doing many different things at one time.
Applying originality in closing problems (i.e., alternative solutions).	Exerting large amounts of force smoothly and precisely.
Ability to profit from experience and alter course of action.	Insensitivity to extraneous factors.
Ability to perform fine manipulation, especially where misalignment appears unexpectedly.	Ability to repeat operations very rapidly, continuously, and precisely the same way over a long period.
Ability to continue to perform when overloaded.	Operating in environments that are hostile to man or beyond human tolerance.
Inductive reasoning.	Deductive reasoning.

technology can be used for decision making involving complex and rapid computation using large amounts of data, while humans can make decisions regarding unpredicted and exceptional situations. Therefore, instead of replacing humans and technology with each other, modern organizations strive to optimally benefit from both aspects, as Figure 1.12 shows.

**Figure 1.12:** Technology and humans in decision making

1.4 The Tradeoff between Flexibility and Support

The flexibility that users have and the support that users get while working with BPM systems (cf. Section 1.1) have a major influence on both satisfaction and productivity. Figure 1.13 shows flexibility and support as two ‘opposed’ properties of business processes. Flexibility refers to the degree to which *users* can make *local* decisions about how to execute business processes. Support refers to the degree to which a *system* makes *centralized* decisions about how to execute business processes. As discussed in Section 1.1, groupware and workflow management systems are the two (extreme) types of BPM systems. The main difference between the two types of systems is decision making, as shown in Figure 1.2 on page 3. While users make decisions locally in groupware systems, the system makes decisions centrally in workflow management systems. Thus, groupware systems provide a high degree of flexibility and a low degree of support, while workflow management systems provide a high degree of support and a low degree of flexibility, as shown in Figure 1.13. In order to be able to align decision making with the predictability of business processes (cf. Section 1.3.1), companies use groupware systems to automate highly unpredictable business processes, and workflow management systems to automate highly predictable business processes, as shown in Figure 1.8(b) on page 6.

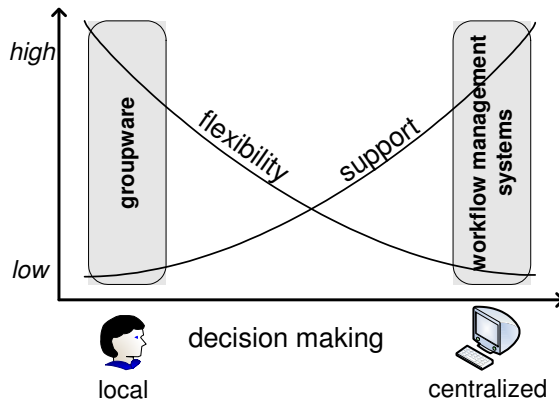


Figure 1.13: Tradeoff: flexibility *or* support in BPM systems [90]

1.5 Problem Definition and Research Goal

Companies rarely choose for extreme centralized or localized decision making, as ‘hard’ and ‘soft’ approaches. Instead, a modern company constantly strives towards an *optimal balance* between the two styles decision making (cf. Section 1.3). The balance between centralized and local decision making must be aligned with the specific situation, i.e., namely with the degree of the environ-

ment turbulency and the predictability of the business process, as described in Section 1.3.1. The more unpredictable the business process is, the more localized decision making should be, as shown in Figure 1.11(b).

The complexity of contemporary business processes raises the need for organizations to use technology for automation of supporting people in decision making while executing business processes. Technology should not be seen as a means that can and should replace humans completely (i.e., ‘hard’ approaches), nor as an ultimate ‘evil’ which should be completely exterminated from business processes (i.e., ‘soft’ approaches). Instead, the best results are achieved by combining the expertise of both humans and technology (cf. Section 1.3.2).

BPM systems are software systems that aim at automating business processes by supporting collaboration, coordination and decision making (cf. Section 1.1). These systems can offer different degrees of flexibility and support in business processes (cf. Section 1.4). Figure 1.13 shows two extreme types of BPM systems that offer *either* flexibility *or* support. First, groupware systems offer a high degree of flexibility and a low degree of support by allowing users to make all decisions about how to execute business processes. Second, workflow management systems make decisions about how to execute business processes, i.e., they offer a high degree of support but not enough flexibility ¹.

Due to the typical complexity of contemporary business processes, companies need BPM systems to support workers in difficult decision making. For example, a workflow management system can provide support by centrally making decisions involving complex manipulation of large amounts of data. However, a workflow management system typically does not allow for flexibility, which disables users to make local decisions about exceptional situations in unpredictable business processes. Thus, a workflow management system forces a company to stick to centralized decision making and work according to the ‘hard’ approach. A groupware system, on the other hand, provides for flexibility by allowing users to make local decisions necessary to handle unpredictable business processes. However, a groupware system does not provide for necessary support while handling complex business processes. Therefore, a company that uses a groupware system is forced to stick to local decision making, which is advocated by the ‘soft’ approaches. Because BPM systems do not offer an optimal ratio between flexibility and support, companies that use these systems are not able to choose an *optimal* balance between centralized and local decision making, as Figure 1.14 shows.

The research presented in this thesis is concerned with the following problem: *BPM systems force companies to implement either centralized or local decision making, instead of allowing for an optimal balance between the two.*

The *goal of the research* is to enable companies that use BPM systems to

¹Note that in this context we have in mind mainstream commercial workflow management systems.

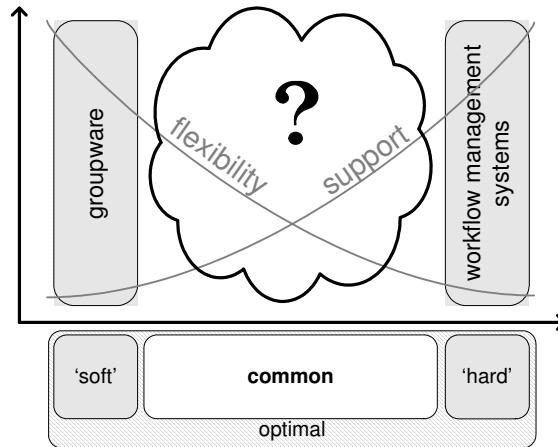


Figure 1.14: Problem definition

achieve an optimal balance between local and centralized decision making. We hope to achieve this (i.e., the *goal in the research*) (1) by proposing a new approach towards process support and (2) by developing a prototype of a workflow management system that can offer an optimal ratio between flexibility and support, as described in Section 1.6.

1.6 Contributions

In this section we briefly describe contributions of this thesis. The three main contributions are:

- The definition of a constraint-based approach to process modeling (cf. Section 1.6.1).
- The definition of a modeling language for the development of constraint-based process models (cf. Section 1.6.2).
- The development of a prototype of a constraint-based workflow management system (cf. Figure 1.6.3).

Two additional contributions are:

- The application of the constraint-based approach to the whole BPM life cycle (cf. Section 1.6.4).
- Showing that a combination of traditional and constraint-based approaches is possible (cf. Section 1.6.5).

1.6.1 Constraint-Based Process Models

Starting point for our constraint-based approach is the observation that only three types of ‘scenarios’ can exist in a business process : (1) *forbidden* scenarios

should never occur in practice, (2) optional scenarios are allowed, but should be avoided in most of the cases, and (3) allowed scenarios can be executed without any concerns. This is illustrated in Figure 1.15(a). As described in Section 1.1, workflow management systems enable definition and execution of models of business processes, which specify the ordering of activities in business processes. In traditional workflow management systems process models *explicitly* specify the ordering of activities, i.e., the *control-flow* of a business process. In other words, during the execution of the model it will be *possible* to execute business process only as explicitly specified in the control-flow, as Figure 1.15(b) shows. Due to the high level of unpredictability of business processes, many allowed and optional executions often cannot be anticipated and explicitly included in the control-flow. Therefore, in traditional systems it is *not possible* to execute substantial subsets of the *allowed* scenarios.

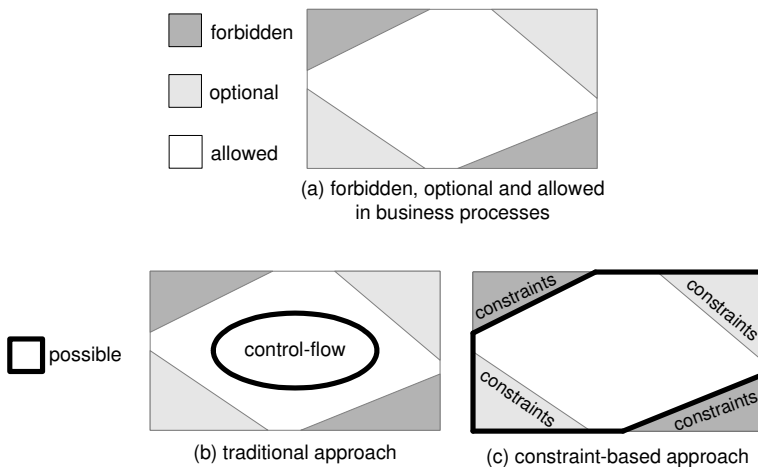


Figure 1.15: New constraint-based approach

We propose a *constraint-based approach* to process models, which makes it *possible* to execute both *allowed* and *optional* scenarios in business processes. Instead of explicitly specifying what is *possible* in business processes, constraint-based process models specify what is *forbidden*, as shown in Figure 1.15(c). The *possible* ordering of activities is *implicitly* specified with *constraints*, i.e., rules that should be followed during execution. Moreover, there are two types of constraints: (1) *mandatory* constraints focus on the forbidden scenarios, and (2) *optional* constraints specify the optional ones. Anything that does not violate mandatory constraints is *possible* during execution. In addition to execution, our constraint-based process models also allow for verification against errors and change during execution (i.e., the so called ad-hoc change).

Our constraint-based approach to process modeling enables flexibility without sacrificing support. On the one hand, constraint-based models tend to offer

more possibilities for execution than the traditional models. This allows users to make local decisions about how to execute business process. On the other hand, a constraint-based process model supports users by being able to keep track of multiple constraints in multiple business processes and preventing users from violating these constraints. In addition, it is also possible to distinguish between the constraints that must be followed (i.e., mandatory) and constraints that should be followed (i.e., optional). In the first case, users will be prevented from violating the constraints. In the second case, users can violate the constraints, but they will be warned in advance about the ‘soft violation’. Moreover, our constraint-based approach enables achieving a ratio between flexibility and support that is optimal for the situation at hand: more constraints in a model mean less flexibility and more support, while less constraints mean more flexibility and less support.

1.6.2 Generic Constraint-Based Process Modeling Language

Constraint-based process models are composed of constraints, which specify rules that should be followed during execution of business processes. A process modeling language used by a workflow management system must fulfill two important criteria. First, the process models developed in the language must be *understandable for end-users*. Second, process models developed in the language must have *formal semantics* in order to be executable in a workflow management system. We propose a new constraint-based process modeling language *ConDec*, which fulfils both criteria. ConDec is based on constraint templates, i.e., types of constraints. Each template has (1) a graphical representation that will be presented to users, and (2) Linear Temporal Logic (LTL) formula specifying the semantics. Our approach and implementation are *generic*, i.e., templates can be easily changed, removed from, or added to the language. Templates are used to create constraints in ConDec process models. Each constraint inherits the graphical representation and semantics (i.e., LTL formula) from its template. Figure 1.16 shows an example of a ConDec constraint, which specifies that activities A and B should not be executed both in one instance of the business process. Users see this constraint as a line with special symbols between two activities, while the LTL semantics remains hidden. While the LTL semantics enable execution of ConDec models, graphical representation makes models understandable by non-experts.

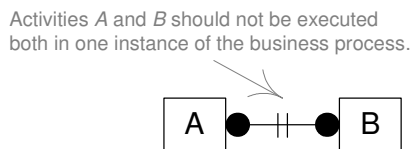


Figure 1.16: A constraint

1.6.3 A Prototype of a Constraint-Based Workflow Management System

We developed the DECLARE system as a prototype of a constraint-based workflow management system. This prototype can be downloaded from <http://declare.sf.net>. DECLARE can support different constraint-based modeling languages and is grounded on our constraint-based approach, as Figure 1.17 shows. Although the default version of the prototype includes the ConDec language, any other constraint-based language based on LTL can easily be added. In addition, constraint-based languages that use formalizations other than LTL can be added by simple extensions of the prototype. Further, DECLARE allows for definition, verification, execution of constraint-based process models, and ad-hoc change of running instances.

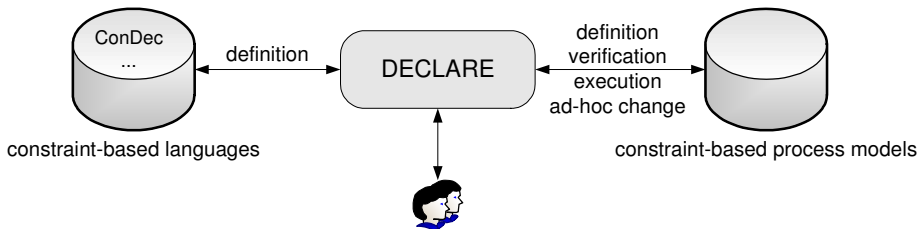


Figure 1.17: The DECLARE prototype

1.6.4 Constraint-Based Approach in the BPM Life Cycle

Workflow management systems can be used together with *process mining* tools for support of all phases of the BPM life cycle shown in Figure 1.1 on page 2. Figure 1.3 on page 3 shows that workflow management systems support design, implementation, and enactment of business processes. Process mining tools support the diagnosis phase by using various process mining techniques for analysis of executed business processes [28]. For example, *ProM* is a process mining tool that can be used for many kinds of analysis of business processes executed in various workflow management systems [28, 91].

Our constraint-based approach can be applied to all phases of the BPM life cycle. On the one hand, DECLARE is a prototype of a workflow management system and, thus, supports design, implementation, and enactment of constraint-based process models, as shown in Figure 1.18. On the other hand, DECLARE languages and models can be re-used in the diagnosis phase by the ProM tool for the analysis of business processes already executed in DECLARE. The results of this analysis can be used for two purposes. First, the results can indicate that process models should be changed, i.e., the cycle is re-entered. Second, DECLARE can use the analysis results during execution of constraint-based models,

as history-based recommendations. The recommendations generated from past executions are presented to users executing DECLARE models as additional information that can help them deal with uncertain situations, i.e., recommendations provide support for DECLARE users without sacrificing available flexibility.

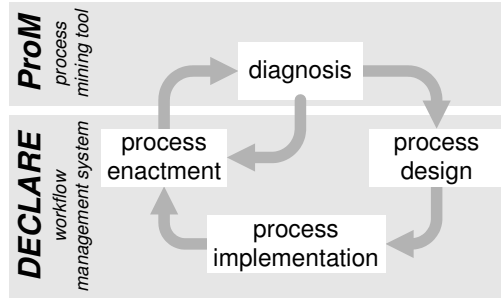


Figure 1.18: Constraint-based approach in the BPM life cycle

1.6.5 Combining Traditional and Constraint-Based Approach

As described in sections 1.3 and 1.4, the level of flexibility and support that users should get in workflow management systems depends on the nature of the business process at hand. Contemporary BPM systems exclusively focus on one type of business processes and offer either support or flexibility (cf. Figure 1.8(b) on page 6 and Figure 1.13 on page 11). However, business processes of different types are typically interleaved, even within the same organization. Consider for example, business processes in the medical domain. Unpredictable medical processes, like, e.g., treating urgent severe injuries, require a high degree of flexibility in order for the staff involved to be able to make local decisions based on each particular patient. This process is very complex and it consists of several other business processes. For example, while treating the injury it might be necessary to perform a blood analysis in the laboratory, which is another business process. Laboratory tests are critical processes and, in order to guarantee reliability of results, they must be executed exactly according to predefined procedures. In other words, instead of flexibility, the blood analysis process requires a high degree of support. The medical domain is one of many examples where a mixture of processes requiring either a lot of support or a lot of flexibility is needed. Therefore, it is important to support the full spectrum.

The DECLARE prototype can be combined with the YAWL system [11, 23, 32, 210, 212] for defining arbitrary decompositions of constraint-based and traditional process models. YAWL is a traditional workflow management system developed at both Queensland University of Technology and Eindhoven University of Technology. The service-oriented architecture of YAWL allows for arbitrary decompositions of various process models. Figure 1.19 shows that the connection

between DECLARE and YAWL models is twofold: (1) a YAWL model can be a sub-model of a DECLARE model and (2) a DECLARE model can be a sub-model of a YAWL model. Decomposition of YAWL and DECLARE models allows for combining different degrees of flexibility and support within one business process. In this way, different parts of one business process can offer different degrees of flexibility and support. Note that YAWL and DECLARE are just two examples, i.e., using a service oriented architecture different styles of modeling can be combined.

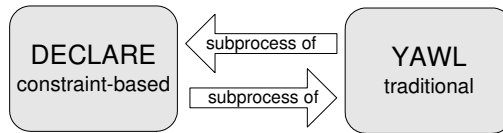


Figure 1.19: Combining different approaches

1.7 Road Map

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview of the related work in the area of flexibility of workflow management systems.

Chapter 3 explains in detail workflow management systems and factors that determine the flexibility of these systems.

Chapter 4 formalizes our constraint-based approach to process modeling.

Chapter 5 presents the ConDec language as one example of a constraint-based process modeling language, which uses Linear Temporal Logic for the formal specification of constraints. The principles described in this chapter can be applied to any other LTL-based language.

Chapter 6 describes the DECLARE system as a prototype workflow management system that supports the constraint-based approach. DECLARE provides full support the constraint-based approach and the ConDec language presented in Chapters 4 and 5, respectively.

Chapter 7 describes how process mining techniques can be applied to the constraint-based approach.

Chapter 8 concludes this thesis, discusses existing problems and proposed future work.

In addition, two appendices are provided. **Appendix A** analyses work distribution in three widely-used commercial workflow management systems, while **Appendix B** presents evaluation results of the workflow pattern [10, 35] support in these three systems. These appendices provide details related to the discussion of flexibility of workflow management systems in Chapter 3.

Chapter 2

Related Work

This chapter provides an overview of related work. Section 2.1 discusses the various proposals to deal with flexibility described in literature. Section 2.2 introduces several workflow management systems that are interesting from the viewpoint of flexibility. Section 2.3 discusses related work on the organization of human work. Finally, Section 2.4 concludes the chapter with an outlook.

2.1 Workflow Flexibility

The importance of flexibility of workflow management systems has been acknowledged by many researchers [66, 109, 125, 196]. The main problem regarding the flexibility of workflow technology remains the requirement to specify business processes in detail, although these processes cannot be predicted with a high certainty [77, 109, 125, 143, 153, 166, 188, 233], and need to be constantly adapted to changing environments [77, 188, 233].

Based on experiences from practice, Reijers provides a brief discussion about the fact that workflow technology failed to bring the intended flexibility by extracting the notion of the business process coordination logic from applications [196]. The paradigm of workflow management systems is based on extracting the business process logic from applications, which should provide for flexibility by making it easier to change the model of the underlying business process [159, 196]. In [196] Reijers argues that, instead of flexibility, workflow management systems improved the logistical aspects of work: managers benefit from decreased through-put times and workers from the fact that the system provides automatically all relevant data and steers the business process.

In [64], Bowers et al. report on a case study conducted in a print industry office that started using a workflow management system. This study revealed that, instead of improving the work in the print office, workflow technology causes serious interruptions in the work of employees because the system completely took over the work and workers were no longer able to handle many unpredictable

situations.

In [125], Heidl et al. addressed the issue of flexibility in the context of a case study conducted in a large market research company that uses workflow technology for support of more than 400 processes. The case study showed that inflexible workflow technology caused problems because: (1) it is almost impossible to identify all steps in the business process in advance, (2) even if a step is identified, it is not obvious whether it should be included in the process model or not, (3) it is not always possible to predict the order of identified steps in advance, and (4) mapping of business processes to process models is prone to errors [125]. Moreover, the authors suggest concrete measures that can improve the flexibility of systems. Namely, it is advocated that flexible systems should allow users to select from multiple execution alternatives and change process models at run-time [125].

Besides the above mentioned theoretical and practical approaches to the problem of flexibility of workflow technology, there have been several attempts to classify flexibility.

Snowdon et al. identify three factors that motivate the need for different types of flexibility [233]. First, the need for *type flexibility* arises from the variety of different information systems. Second, *volume flexibility* is needed to deal with the amount of information types. Third, *structural flexibility* is necessary because of the need to work in different ways.

Soffer uses concepts from the Generic Process Model (GPM) and the theory of coordination to classify flexibility into *short-term flexibility* and *long-term flexibility* [234]. Short-term flexibility implies the ability to deviate temporarily from a standard way of working, while the long-term flexibility allows for changing the standard way of working.

In [232], Carlsen et al. propose a quality evaluation framework, which they use to evaluate five workflow management products (including commercial systems and prototypes), and identify desirable flexibility features. The framework is based on the quality of a process model and the quality of a modeling language. Evaluation of workflow products identified a large set of desirable flexibility features for workflow management systems (e.g., flexible error handling support, quick turnaround for model changes, etc.). In addition, evaluation showed that none of the five workflow products were flexible along all identified features, and some of features were not covered by any product.

The first comprehensive taxonomy of concrete features that enhance flexibility of workflow management systems was given in 1999 by Heidl et al. [36, 125]. In 2007 Schonenberg et al. conducted a follow-up study and adjusted the original taxonomy to recent developments in workflow technology [226–228]. The remainder of this section is organized as follows. First, we present taxonomies of Heidl et al. and Schonenberg et al. in sections 2.1.1 and 2.1.2, respectively. Second, we present related work classified by flexibility types of Schonenberg et al. in sections 2.1.3, 2.1.4, 2.1.5, and 2.1.6.

2.1.1 Taxonomy of Flexibility by Heinel et al.

In [125], Heinel et al. use a case study conducted in a large marketing company as an indication of the need for flexibility in workflow technology. This study showed that serious problems arise due to the fact that it is hard to predict all alternatives in business process execution when specifying a process model, and that flexibility in the context of execution of instances of process models is needed to cope with these problems. Flexibility of a workflow management system is seen as a degree to which users can choose between various alternatives while executing process models. *Flexibility by selection* and *flexibility by adaptation* are identified as two concepts that should be supported by a flexible workflow management system, as Figure 2.1 shows.

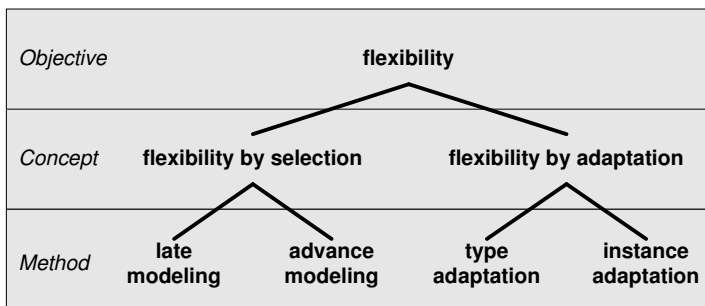


Figure 2.1: Classification scheme for flexibility of workflow management systems by Heinel et al. [125]

Flexibility by selection gives a user a certain degree of freedom by offering multiple execution alternatives. This type of flexibility can be achieved by *advance modeling* and *late modeling*. Advance modeling means that multiple execution alternatives are implicitly or explicitly specified in the process model. When it comes to late modeling, parts of a process model are not modeled before execution, i.e., they are left as ‘black boxes’, and the actual execution of these parts is selected only at the execution time.

The limitation of flexibility by selection is that it has to be anticipated and included in the process model. Flexibility by adaptation considers adding one or more unforeseen execution alternatives to a process model while the model is being executed. This can be achieved via *type adaptation* or *instance adaptation*. In the case of type adaptation, a process model is changed while running instances of that model are not affected by the change. In case of instance adaptation, the change is applied to running instances.

2.1.2 Taxonomy of Flexibility by Schonenberg et al.

In [226–228], Schonenberg et al. revisited the issue of flexibility and extended the original taxonomy by Heinel et al. [125]: the terminology changed, one flexibility

type is abandoned, and one flexibility type is added, as Table 2.1 shows. These changes reflect the recent innovations in the area of workflow technology. In addition, Schonenberg et al. evaluated several state-of-the-art workflow management systems with respect to flexibility types support.

Table 2.1: Two taxonomies of flexibility

Heinl et al. [125]		Schonenberg et al. [226–228]
flexibility by selection	advance modeling	flexibility by design
	late modeling	flexibility by underspecification
flexibility by adaptation	type adaptation	flexibility by change
	instance adaptation	
×		flexibility by deviation

In [226–228], Schonenberg et al. propose four types of flexibility:

1. *Flexibility by design* is the ability to specify alternative execution alternatives in the process model, such that users can select the most appropriate alternative at run-time for each process instance. This type of flexibility refers to *advance modeling* of Heinl et al.
2. *Flexibility by underspecification* is the ability to leave parts of a process model unspecified. These parts are later specified during execution of process instances. In this way, parts of the execution alternatives are left unspecified in the process model, and are specified later during the execution. This type of flexibility refers to *late modeling* of Heinl et al.
3. *Flexibility by change* is the ability to modify a process model at run-time, such that one or several of the currently running process instances are migrated to the new model. Change enables adding one or more execution alternatives during execution of process instances. This type of flexibility refers to *instance adaptation* of Heinl et al.
4. *Flexibility by deviation* is the ability to deviate at run-time from the execution alternatives specified in the process model, without changing the process model. Deviation enables users to ‘ignore’ execution alternatives prescribed by the process model by executing an alternative not prescribed in the model. This is a new type of flexibility introduced by Schonenberg et al. and is inspired by new approaches (cf. FLOWer [39,180] and DECLARE [183])

Further in this section we present relevant research conducted in the area of each of the four types of flexibility proposed by Schonenberg et al.: flexibility by design in Section 2.1.3, flexibility by underspecification in Section 2.1.4, flexibility by change in Section 2.1.5, and flexibility by deviation in Section 2.1.6.

2.1.3 Flexibility by Design

Flexibility by design, as the ability to include multiple execution scenarios in process models, has drawn much research attention in the area of workflow technology. While some approaches advocate that flexibility by design can be increased by adjusting the way existing technology is used [22, 109, 137, 149, 194, 198], other approaches propose radical innovations in the area [55, 56, 92, 115, 186, 187, 256].

‘Softening’ Traditional Approaches

Some researchers propose concrete methods for developing process models using existing modeling languages in a way that models offer as many execution alternatives as possible [22, 137, 194, 198]. For example, in [194, 198], Reijers et al. propose a set of heuristics (the so-called ‘best practices’) that can improve flexibility by design. One of the proposed heuristics advocates that parallel activities in process models imply more execution alternatives than sequential activities. In [22], van der Aalst goes a step further and describes the applicability of measures that can increase the number of available execution alternatives. For example, this author suggests that “putting subsequent tasks in parallel can only have a considerable positive effect if the following conditions are satisfied: resources from different classes execute the tasks, the flow times of the parallel subprocesses are of the same order of magnitude, ...” [22].

Other researchers propose ‘relaxing’ a process model by introducing *optional* areas. In [149], Klingemann propose splitting up a process model into two parts. First, the *mandatory part* consists of activities that must be executed in a predefined order, i.e., this is the traditional notion of a process model. Second, the *flexible part* consists of activities that can be selected depending on requirements at run-time. Similar concepts are proposed by Georgakopoulos in [109], who claims that flexible processes “specify prescribed and optional activities...”. During execution, prescribed activities are always required, while users decide themselves whether and when to execute optional activities. Thus, optional activities allow users to impose the process structure when it is necessary, i.e., they allow for multiple execution alternatives.

Data-Driven Approaches

There are several approaches that focus on the data availability in order to improve flexibility. In [117], Grigori et al. propose *anticipation* as a means for more flexible execution of traditional process models. Anticipation allows an activity to start its execution when all input data parameters are available, which may be earlier than specified in the control-flow of the process model.

The idea to focus on the product data instead of the control-flow when deciding the order of activities was introduced by van der Aalst in [15, 18]. Here the author proposes the automatic generation of a process model (represented

as a Petri Net [29, 72, 93]) from a given Bill Of Materials (BOM). This method was worked out in more detail by van der Aalst et al. in [197], where authors propose a method called Product-Driven Workflow Design (PDWD) for deriving a process model. PDWD takes a product specification in the form of BOM and three design criteria (i.e., quality, costs and time) as a starting point for deriving a favorable new design of the process model. In addition, the authors demonstrate how the ExSpect tool [3] can support PDWD. The possibility to support PDWD by case-handling systems [26, 39] was presented by Vanderfeesten et al. in [249, 250].

While the approaches mentioned in the previous paragraph focus on deriving a process model from a BOM, more advanced approaches advocate the direct execution of the BOM. In [251, 252], Vanderfeesten et al. present an implementation of a system for direct execution of Product Data Models (PDMs)¹. Similarly, in [257], Wang et al. present an execution framework for a document-driven workflow management system that does not require an explicit control-flow. Instead, the execution of a process is driven by input documents.

Proposals for New Process Modeling Languages

In [66, 135], Jablonski et al. propose meta-modeling of workflows in the system called MOBILE. In [135] authors distinguish between *prescriptive* and *descriptive* workflows. In prescriptive workflows eligible instances are known a priori, while in descriptive workflows instances are not known beforehand but are determined during processing. MOBILE supports meta-modeling of both prescriptive and descriptive process models by means of control predicates [66], which are internally presented by Petri Nets [29, 72, 93]. Moreover, this approach allows for combining prescriptive and descriptive processes in the same framework by decomposing the two types of models [135].

Several approaches propose using *intertask dependencies* for specification of the process models. In [55, 56], Attie et al. propose using Computational Tree Logic (CTL) [74] for the specification of intertask dependencies amongst different unique events (e.g., commit dependency, abort dependency, conditional existence dependency, etc.). Dependencies are transformed into automata, which are used by a central scheduler to decide if particular events are accepted, delayed or rejected. In [186, 187], Raposo et al. propose a larger set of basic interdependencies and propose modeling their coordination using Petri Nets [29, 72, 93].

Another popular stream of research is applying rule-based or constraint-based process modeling languages [92, 115, 256] that are able to offer multiple execution alternatives and, therefore, can enhance flexibility by design.

In [115], Glance et al. use process grammars for definition of rules involving activities and documents. Process models are executed via execution of rules

¹Product Data Models are a special kind of BOM where the building blocks are data elements, instead of physical parts.

that trigger each other.

The Freeflow prototype presented in [92] uses constraints for building declarative process models. Freeflow constraints represent dependencies between states (e.g., inactive, active, disabled, enabled, etc.) of different activities, i.e., an activity can enter a specific state only if another activity is in a certain state.

Plasmeijer et al. apply the paradigm of functional programming languages embedded in the iTask system to workflow management systems [185]. On the one hand, the iTask system supports all workflow patterns [10, 35, 208, 211, 213]. On the other hand, it offers additional features like suspending activities, passing activities to other users and continuing with a suspended activity. Another interesting property of this approach is the possibility to automatically generate a multi-user interactive web-based workflow management system.

Some approaches consider process models based on dependencies between events involving activities [80, 256]. For example, the constraint-based language presented in [256] uses rules involving (1) preconditions that must hold before an activity can be executed, (2) postconditions that must hold after an activity is executed and (3) “parconditions” that must hold in general before or after an activity is executed. In addition, the Tucupi server is implemented in Prolog [239], which is a prototype of a system supporting this approach. A similar idea is presented in [141] by Joeris, who proposes flexible workflow enactment based on event-condition-action (ECA) rules. In [162, 163], a temporal constraint network is proposed for business process execution. The authors use thirteen temporal intervals defined by Allen [50] (e.g., *before*, *meets*, *during*, *overlaps*, *starts*, *finishes*, *after*, etc.) to define selection constraints (which define activities in a process) and scheduling constraints (which define when these activities should be executed). Moreover, using the notion of Business Process Constraint Network (BPCN) ensures execution of process models that conforms to specified constraints and detection of (possible) conflicting constraints. After a knowledge worker invokes a special build function to dynamically adapt the instance template (instance templates define total order of task execution), translation of Interval Algebra (IA) network generated from constraints to Point Algebra (PA) network [60] is used to validate whether the given instance template conforms to given constraints. If this validation is satisfactory, the execution continues according to the instance template.

New languages for specification of process models that offer flexibility by design have also been proposed in the areas of web services and contracting. In [78], $\mathcal{CTR}\text{-S}$, an extended version Concurrent Transaction Logic [62], is proposed for process modeling in the context of contracts in web services. The authors propose using $\mathcal{CTR}\text{-S}$ for specifying contracts as formulas that represent various choices that are available for the parties in the contract. This language allows stating the desired outcomes of the contract execution and to verify that the outcome can be achieved as long as all parties obey to the rules of the contract.

The declarative SCIFF language is developed for the specification, monitoring

and verification of interaction protocol of web services [48, 49]. SCIFF envisages a powerful logic-based language with a clear declarative semantics. The SCIFF language is intended for specifying social interaction, and is equipped with a proof procedure capable to check at run-time or a-posteriori whether a set of interacting entities is behaving in a conforming manner with respect to a given specification. Due to its high abstraction level, SCIFF can be used for dependency specifications in various domains. For example, in the area of business processes, SCIFF specifications can include activities (i.e., the control-flow), temporal constraints (i.e., deadlines) and data dependencies. The possibility to learn SCIFF specifications from past executions is presented in [154, 155]. The SCIFF language is described in more detail in Section 7.3 of this thesis.

In [269], Zaha et al. propose a language called *Let's Dance* for modeling interactions of web services. This language focuses on flexible modeling of message exchange between services. A straight-forward graphical notation is used to represent patterns in message exchange, while π -calculus [174] captures the execution semantics [81].

Our approach is more comprehensive than the approaches discussed above: it includes (1) a formal definition of a constraint-based approach on an abstract (i.e., language-independent) level, (2) a concrete, formal constraint-based language that enables deadlock-free execution, ad-hoc change and verification, (3) a working ‘proof of concept’ prototype, (4) application of the constraint-based approach to the whole BPM cycle, and (5) combining procedural and constraint-based process models (cf. Section 1.6). To our knowledge, none of the new languages discussed above include these five aspects together.

2.1.4 Flexibility by Underspecification

Underspecification in process models has been addressed by several researchers. In [129, 130], Herrmann et al. advocate *vagueness* in models of socio-technical systems. This approach proposes the semi-structured modeling language SeeMe [129], which allows *uncertain*, *questionable* and *unknown* knowledge to be included in models, as well as *checked* and *committed*. For example, SeeMe allows for definitions about ordering of activities, process decomposition, role allocation, etc. to be specified as uncertain, or even omitted from the model. The vagueness allows knowledge workers to decide at later stages about the actual process.

Van der Aalst proposes enhancing flexibility of process models with *generic processes* [16, 21]. Besides elementary activities and routing elements, processes models can contain of *non-atomic concrete processes* and *generic processes*. While activities are directly executed by users, non-atomic concrete and generic processes decompose to process models. In the case of a non-atomic concrete process, the process to be executed is already specified in the original model. In the case of a generic process, the model to be executed must be selected at the execution time, i.e., generic processes refer to unspecified placeholders that are

specified only at the execution time.

In [167], Mangan and Sadiq propose building instances from partially defined process models in order to deal with the fact that it is often not possible to completely predefine business processes. The idea is that a partially defined process model is only fully specified at run-time and may be unique for each instance. Instances can be built of activities and subprocesses (i.e., modeling *fragments*), ordered in sequences, parallel branches and as multiple executions (i.e., modeling *constructs*). In addition, the building of instances is supported by three groups of previously defined domain-specific constraints, as rules under which valid instances can be built. First, *selection constraints* define which fragments are available for the instance. Second, due to the lack of an explicit termination activity, *termination constraints* are needed to define when an instance is completed. Third, additional restrictions in an instance are imposed by *build constraints*. An instance is dynamically built in a valid manner for as long as all constraints are satisfied. In addition, Sadiq et al. propose using this approach to build *pockets of flexibility*, which are (together with predefined activities) components of process models [220].

Trajcevski et al. propose process model specification based on the known effects of process activities [243]. In addition to the ‘known’, process models can also contain the ‘ignorance’ (unknown values allow specifying situations of dealing with incomplete information). In addition, the authors define an entailment relation which enables verifying the correctness of process models (in terms of achieving a desired goal).

The OPENflow system is an example of a system that directly supports flexibility by underspecification [121]. This system allows for incorporating *genesis activities* in process models. A genesis activity represents a placeholder for an undefined subprocess. The actual structure of a genesis activity is determined at run-time.

In [41, 44, 45] Adams et al. describe the Worklet Service as a means to dynamically build process instances based on the specific context. The idea is to dynamically substitute an activity with a new instance of a contextually selected process, i.e., a *worklet*. The decision about which worklet to select for a given activity depends on the activity data and existing *ripple down rules*. In this manner, worklet activities in process models represent unspecified parts of the model, which are to be determined by the Worklet Service at run-time².

Staffware [12, 237] is a popular commercial workflow management system that supports flexibility by underspecification via dynamic process selection [116]. Staffware process models consist of activities and subprocesses. Besides static process selection, where subprocesses are already specified in the model, dynamic process selection allows for selection of an appropriate sub-process at execution time based on the instance data [116]. The idea is comparable to the worklets

²The Worklet Service is described in more detail in sections 2.2.3 and 6.11.3 of this thesis.

approach [41, 44, 45], i.e., subprocesses in Staffware instances are dynamically selected based on the instance data (e.g., a specific data element may carry the name of the subprocess to be launched)³.

2.1.5 Flexibility by Change

Flexibility by change is achieved when instances can be changed at run-time. This topic has driven much research attention in two areas. First, we will describe the research conducted in the area of *adaptive systems* (i.e., systems that enable run-time change of instances). Second, we describe some examples of the research in the area of *ad-hoc systems* (i.e., systems that enable run-time construction of instances).

Adaptive Approaches

Run-time (i.e., dynamic) change of instances of process models has drawn much attention amongst researchers [24, 36, 46, 68, 71, 100, 101, 109, 140, 145, 148, 151, 189, 219, 230, 265]. A comprehensive overview of the existing approaches to dynamic change in the context of workflow technology is given by Rinderle et al. in [201]. The authors present a good classification of existing approaches and evaluate the approaches against identified correctness criteria.

At the most advanced level, it might be necessary to change an instance in an ad-hoc manner (i.e., ad-hoc change), which implies that the change is unpredictable and often applied as a response to unforeseen situations [201]. Systems like Breeze [219], WASA₂ [265] and ADEPT [164, 189, 202] use advanced compliance checks (i.e., to check whether the current execution can be applied to the changed instance), correctness properties of the process model (e.g., regarding data flow), etc., to support ad-hoc change [201]. Unfortunately, today's commercial systems do not provide sufficient support for ad-hoc change. Systems like InConcert, SER Workflow and FileNet Ensemble are rare examples of commercial systems that, to some extent, enable ad-hoc change of running instances [201].

Another important functionality when it comes to dynamic change is the so-called *migration*, where a dynamic change is applied to multiple running instances (e.g., due to a change in law regulations, all running instances must be migrated from the old to the new process) [201]. Besides for the basic problems that accompany change of a single instance, migration of multiple instances introduces some additional difficulties. Namely, additional challenges emerge in systems that aim at concurrently supporting both types of change, e.g., in cases when conflicting changes must be resolved at different levels [203]. Unfortunately, not many systems support this type of change. The ADEPT system has been

³The dynamic process selection in Staffware is explained in more detail in Section 2.2.1 of this thesis.

extended to support change of multiple instances of a process model, resulting in the second version of the system, i.e., ADEPT2 [192, 193]⁴.

Some systems support *pre-planned* and *automated* instance changes, where necessary changes and their scope are already known at the design phase (i.e., while the process model is developed) [201]. This type of change is supported by ADEPT [164, 189, 191, 202], WASA₂ [265], InConcert [133], etc. [201]. In order to support pre-planned instance changes, a system must be able to (1) detect failures that cause the change, (2) determine necessary changes, (3) identify instances that must be changed, (4) correctly introduce the change to those instances, and (5) notify the users about the conducted change(s) [29, 201].

A problem that often arises during dynamic change is pointed out by Ellis et al. as the “dynamic change bug” [101]. A dynamic change is preformed on running instance, i.e., the instance already has a history that puts the instance in a certain *state* when the change takes place. The complexity of the dynamic change stems from the fact that for the current state of an instance, an appropriate state in the new model has to be found, and this is not always possible. In [20], van der Aalst proposes an approach for dealing with this problem by calculating the *safe change region*. A dynamic change is only allowed if an instance is in this region.

As a means of comparing various approaches to process control-flow change, Weber et al. [260] propose a set of seventeen *change patterns* and six *change support features*. First, change patterns are classified into *adaptation* patterns and patterns for *predefined change*. While adaptation patterns cover unpredictable changes, predefined patterns consider only the changes that are predefined in the process model at the design time. Second, change support features of workflow management systems that are identified are, e.g., version control, change correctness, change traceability, etc.

In [36], van der Aalst and Jablonski propose a scheme for classifying workflow changes in detail based on six criteria: (1) the *reason for change* can be a development outside or inside the system, (2) the *effect of change* can be momentary or evolutionary, (3) the *effected perspectives* can be process, organization, information, operation or integration perspective, (4) the *kind of change* can be extending, reducing, replacing or re-linking, (5) the moment at which *change is allowed* can be at entry time or on-the-fly, and (6) the choice *what to do with running process instances* can be to abort old instances, proceeded according to the old model, etc.

Ellis and Keddara propose a Modeling Language to support Dynamic Evolution within Workflow Systems (*ML-DEWS*) as a means for modeling the *process of dynamic change* [100]. The language supports a variety of predefined change schemes, e.g., the *abort scheme* as a disruptive change strategy where the instance is simply aborted, the *defer scheme* that allows the instance to proceed

⁴A more detailed description of ADEPT is given in Section 2.2.4 of this thesis.

according to the old process model, the *ad-hoc scheme* that supports changes whose components are not fully specified at design time, etc.

In [118], Günther et al. apply process mining techniques [28] to change logs created by adaptive systems. The authors propose using process mining to provide an aggregated view of all changes that happened so far in process instances. The mining results can trigger various process improvement actions, e.g., a result may indicate that, due to frequent changes, a process redesign is necessary.

Weber et al. introduce a framework for the agile mining of business processes that supports the whole process life cycle by using *Conversational Case-based Reasoning* (CCBR), *adaptive business process management* and *process mining* [259]. Process mining techniques are used to extract and analyze information about realized process adaptations. Integration of the ADEPT [164,189,191,202] and CBRFlow [261] prototypes enables using CCBR to perform ad-hoc changes of single process instances, to memorize these changes, and to support their reuse in similar future situations. Collected information can be used by process engineers to adapt process models and to migrate related process instances to the new model.

Ad-Hoc Approaches

Ad-hoc approaches to workflow management systems provide a powerful mechanism to increase flexibility by change by allowing users to build the process model for each instance while executing the instance.

In [94], Dustdar investigates the relevant criteria for process-aware collaboration and proposes an ad-hoc approach to workflow management systems implemented in the system Caramba. Besides the traditional execution of process models, this approach allows users to execute ad-hoc instances that are not based on a predefined process model. Instead of the system, users coordinate activities in ad-hoc instances.

InConcert is an example of a commercial ad-hoc workflow management system, which allows users to design or modify process models [29,133]. InConcert allows for the creation of a new instance in four manners [29]. First, a new instance can be created based on an existing process model. Second, it is possible to create a new instance based on a previously changed existing process model. Third, an ad-hoc instance can be initiated by specifying a sequence of activities. Fourth, a new instance can be initiated as a ‘free routing process’, i.e., the instance is created based on an empty process, and the actual process model is created on the fly.

2.1.6 Flexibility by Deviation

Deviation from predefined process models is recognized as a means for increasing flexibility in the workflow area [76,264]. While some approaches propose

methods for deviations from traditional process models [39, 76, 180, 264, 265], other approaches focus on specialized mechanisms to handle unexpected situations [42, 98, 218, 241].

Deviation in Traditional Approaches

In [76], Cugola proposes deviating from process models when it comes to situations unforeseen in the design phase, i.e., not incorporated in the model. The author describes two types of inconsistencies that may occur at unexpected situations. First, a *domain-level inconsistency* occurs when an actual instance does not follow the process model. Second, an *environment-level inconsistency* occurs when a business process is executed outside of the system, and the system has no knowledge about how the process is executed. These inconsistencies are caused by *domain-level deviations* and *environment-level deviations*, i.e., as actions that system users undertake in order to deal with unforeseen situations. The author describes the PROSYT system, which is able to tolerate *domain-level deviations*, and, thus, minimize *environment-level deviations*. The PROSYT system deals with unforeseen situations by allowing a *deviation policy* and a *consistency handling policy* to be specified for a process model. A deviation policy identifies which forms of deviation are tolerated, while a consistency handling policy ensures any allowed deviations do not impact the overall correctness of the system.

In the context of the WASA prototype [264, 265], Weske nominates three user-initiated operations. These operations are: *skip activity* that has not been started yet, *stop activity* that is currently being executed, and *repeat activity* that has already been executed [264]. These three operations allow for deviations from normal workflow execution, but do not change the original process model.

FLOWer [180] is an example of a commercial system that allows deviations from process models. FLOWer is a case-handling system [39] that allows users to *open* activities that are not supposed to be executed yet, *skip* activities that should be executed, and *redo* an activity that has been executed before. FLOWer allows for these deviations by applying a powerful mechanism that ensures consistency of running instances (e.g., data elements are taken into account). A more detailed description of FLOWer is given in Section 2.2.2.

Exception Handling

Another area of research that is concerned with deviations from what is specified in process models is *exception handling*. Exception handling provides a means for handling errors without explicitly including them in the process model. Exceptions are seen as errors/failures that can occur during execution of process models [42, 43, 98, 218, 241]. Although we consider exception handling as a technique to handle (technical) problems rather than supporting flexibility, it is related to the above approaches. Therefore, we mention some work on exception handling.

Strong et al. investigated exception handling of an operational process in one organization. They suggested points for further research on the roles of people in computerized systems and design of computer-based systems that can handle multiple conflicting goals [241].

In [98], Eder et al. discuss advanced concepts concerning recovery from system failures and semantic failures in the context of workflow transactions. The authors identify different failure sources (i.e., workflow engine failures, activity failures, and communication failures) and failure classes (i.e., system failures and semantic failures) in process-oriented and document-oriented workflows.

Saastamoinen et al. propose using a set of (1) formal organizational rules, (2) informal group rules, and (3) informal individual rules for handling exceptions and assuring that the goal of the process is achieved after the change [218].

The work presented in [41–43, 208, 209] is a comprehensive attempt to provide a concrete framework for exception handling in workflow management systems. In [43], Adams et al. propose using the *Worklet Service* [41, 44, 45] for exception handling of events that occur while executing process models. When such an event occurs, a repository of rules is used to select the procedure that should be used to handle the exception. In [208, 209], Russell et al. define a rigorous classification framework for workflow exception handling independent of modeling approaches and technologies, i.e., a set of *workflow exception patterns* is identified. Based on these exception patterns, in [42] Adams et al. present their implementation of a *Exception Service*, which provides a fully featured exception handling paradigm for detecting, handling and incorporating exceptions as they occur. Moreover, this implementation allows for handling both predicted and unpredicted exceptions.

Various approaches to exception handling have been implemented in a number of systems, e.g., WAMO [97], ConTracts [200], Exotica [51], OPERA [119, 120], TREX [240], WIDE [67], etc.

2.2 Workflow Management Systems

Numerous workflow management systems are available on the market today in addition to the open source products and academic prototypes [29]. In this section we shortly present several workflow management systems that are able to offer one or more types of flexibility. We start by presenting two popular commercial systems. In Section 2.2.1 we present the traditional system Staffware [238], which provides support for flexibility by underspecification and a limited support for flexibility by change. In Section 2.2.2 we present the case-handling system FLOWer [180], which provides flexibility by deviation. Then, we present two academic systems (i.e., these systems are developed under the supervision of academic workflow researchers). In Section 2.2.3 we present YAWL [23, 210, 212], which supports flexibility by underspecification and exception handling. Finally,

in Section 2.2.4, we present ADEPT [164, 189, 191, 202], a workflow management systems focusing on supporting flexibility by change.

2.2.1 Staffware

Staffware is one of the most used workflow management systems in the world. For example, in 1998, it was estimated that Staffware had 25% of the world market. Staffware consists of several components that are, in general, used to define process models, users and their roles, and to execute instances of process models. For example, Figure 2.2 shows one process model in the Staffware component for model definition, i.e., the Graphical Workflow Definer, and Figure 2.3 shows the Work Queue Manager - a client tool which is used by users to execute activities of running instances.

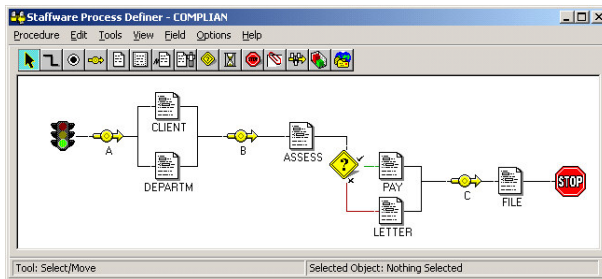


Figure 2.2: A process model in Staffware

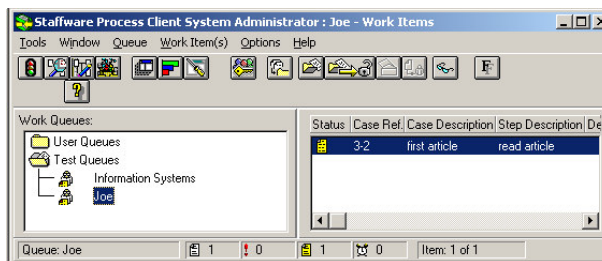


Figure 2.3: A work queue with a work item in Staffware

Despite its reputation of being an inflexible system, in recent years considerable efforts have been undertaken to enrich Staffware with features that enhance its flexibility. Since recently, Staffware supports flexibility by underspecification and, to a limited extent, flexibility by change.

Dynamic Process Orchestration in Staffware

The new paradigm of Dynamic Process Orchestration [116] introduces flexibility by underspecification in Staffware. Staffware process models can be composed of atomic activities and subprocesses. There are four ways in which subprocesses can be invoked from their parent processes [116]: (1) static process selection, (2) dynamic process selection, (3) multiple process selection, and (4) goal-driven process selection.

Static process selection. In a conventional way, subprocesses are invoked in a static manner. This means that the subprocess is known in advance and explicitly specified in the process model. The result is that the same subprocess will be invoked for all instances of the parent process model.

Dynamic process selection. Often it is the case that a range of subprocesses can be invoked, the choice of which depends on specific circumstances. The names of such subprocesses and the conditions of their usage is known in advance, and specified in the parent process model. However, the circumstances are only known at the execution time, and then it is decided which subprocess should be invoked. The dynamic process selection is achieved by specifying in a process model a range of possible subprocesses and conditions (involving data elements) for their invocation. The result is that, each instance will decide, based on current values of its data elements, which one of the available subprocesses to invoke.

Multiple process selection. This is an extension of the dynamic process selection in a way that, instead of invoking only one subprocess, *multiple* subprocesses can be invoked in a dynamic manner. The parent process will proceed to the next activity only when all invoked subprocesses have completed successfully.

Goal-driven process selection. In some circumstances, the name(s) of subprocesses might not be known in advance and, thus, cannot be specified in the process model. Instead, only a *goal* (e.g., ‘examine patient’) that the subprocess should achieve is known and specified in the parent process. Further, on the system level each subprocess is tagged with the goal it achieves (e.g., ‘examine patient’, ‘perform tests’) and the entry conditions (e.g., ‘*age* > 65’). The result is that, for each instance, the system will automatically invoke a subprocesses that achieves the given goal and satisfies the entry condition based on the current instance data.

Change on the Instance Level in Staffware

Staffware does not support dynamic change in the ad-hoc manner described in Section 2.1.5. This means that it is not possible to apply a dynamic change

directly on a running instance. According to [201], a change of a process model can trigger change of its running instances. However, there are several problems arising when it comes to this kind of dynamic change in Staffware, as pointed out by Rinderle et al. in [201]. First, it might happen that activities in running instances are automatically deleted, without informing the users who are working on these instances. Second, if a deleted activity has already been executed, the results of this activity are lost. Third, Staffware suffers from the so-called ‘changing the past’ problem, i.e., some changes might influence the past of the instance, which may lead to missing data values or even program failures. Finally, dynamic changes in Staffware are too restrictive (e.g., if an activity is activated, insertions before it are no longer possible).

2.2.2 FLOWer

FLOWer is a case-handling system [26, 39], i.e., there are two major differences when compared with traditional workflow management systems. First, the execution order of activities is heavily influenced by data elements in case-handling systems. Second, while traditional systems offer atomic activities from running instances to users for execution, case-handling systems offer whole cases to users.

In traditional workflow management systems, the execution order of activities in instances is explicitly defined by the control-flow definition of the underlying process model. In FLOWer, users can follow the execution order of activities defined in the control-flow of the instance. Figure 2.4 shows the control-flow specification of one process model in FLOWer. However, in addition, the control-flow ordering may as well be violated. On the one hand, FLOWer considers an activity to be successfully executed as soon as all its mandatory data elements become available. This means that, if all mandatory data elements of an activity become available before the manual execution of this activity, the activity is considered to be successfully completed, and a manual execution is no longer necessary. On the other hand, an activity can be executed or skipped even if it contradicts to the control-flow specification. In this way, FLOWer is one of the rare commercial systems that offers *flexibility by deviation*.

Besides executing activities according to the control-flow specification (i.e., executing currently *enabled* activities), users of FLOWer can also [39, 180]:

- *open an activity* that is not enabled yet (i.e., disabled activity),
- *skip an activity* that has not been executed yet, i.e., not execute an activity that should be executed according to the control-flow, and
- *re-do an activity* that has already been executed before, i.e., choose to execute again an activity that has already been executed.

A major difference between traditional workflow management systems and case-handling systems is how the work available in running instances is offered to users. On the one hand, traditional systems typically offer work to users as

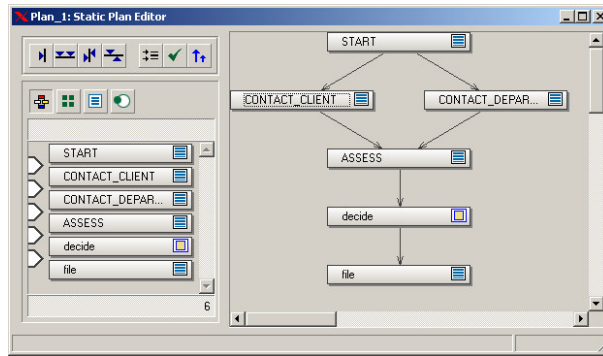


Figure 2.4: A process model in FLOWER

atomic activities, i.e., a user has access to a list containing all currently enabled activities for all running instances (e.g., Figure 2.3 shows such a list in Staffware). On the other hand, case-handling systems try to avoid the narrow view provided by activities, and therefore present the whole instance to the user. Note that, although a whole case is presented to a user, he(she) can only work with activities for which he(she) is authorized. In more detail, for each activity in FLOWER, users can get authorizations to execute, skip and redo the activity. Figure 2.5 shows how FLOWER presents a whole instance to one user. In the instance shown in this figure, activities *Claim Start* and *Register Claim* have already been successfully executed (as indicated by the ‘check’ symbol). After that, activity *Get Medical Report* was skipped (as indicated by the ‘skip arrow’ symbol). Currently, activities *Get Police Report*, *Assign Loss Adjuster* and *Witness Statements* are currently enabled, i.e., available for execution. The last two activities (i.e., *Policy Holder Liable* and *Close Case*) are not enabled yet.

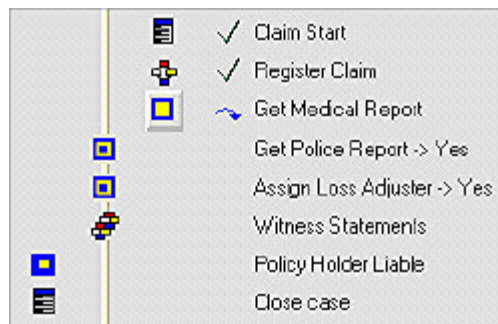


Figure 2.5: FLOWER Wave Front

Note that skipping, opening and redoing an activity in FLOWER may also affect other activities. First, when a disabled activity is opened, all preceding

not yet executed activities will automatically become skipped. For example, if activity *Close Case* would be opened or skipped at the moment presented in Figure 2.5, then activities *Get Police Report*, *Assign Loss Adjuster*, *Witness Statements*, and *Policy Holder Liable* would automatically become skipped. If activity *Close Case* was opened, then it can be directly executed. If activity *Close Case* was skipped, then the case execution continues after this activity, e.g., the instance presented in Figure 2.5 is completed. Second, if an activity is re-done, all succeeding executed activities must also be re-done. For example, if activity *Claim Start* would be re-done at the moment presented in Figure 2.5, then activities *Register Claim* and *Get Medical Report* would also need to be re-done after activity *Claim Start*. A drawback of the described side-effects is that the deviation becomes more extensive than intended, e.g., while attempting to re-do only one activity, a FLOWer user might end up being forced to also re-do many preceding activities.

2.2.3 YAWL

YAWL is a workflow management system developed in a collaboration between the Eindhoven University of Technology and the University of Queensland [11, 23, 32, 210, 212]. YAWL is developed in the context of the workflow patterns initiative [10, 32, 35, 208] and aims at supporting all workflow patterns, i.e., it aims at supporting various features offered by existing workflow management systems. In simple words, YAWL is driven by the ambition to be able to provide a comprehensive support for most patterns while using a relatively simple language.

In its essence, YAWL is built as a traditional workflow management system, i.e., ordering of activities is defined in the traditional ‘control-flow’ manner. While executing activities in running instances in YAWL, users must follow the order strictly specified in the control-flow of the underlying process model. Figure 2.6 shows a process model in YAWL.

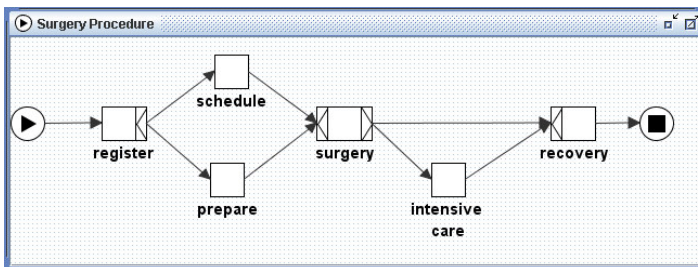


Figure 2.6: A process model in YAWL

YAWL’s architecture is based on the so-called service-oriented architecture, and the systems can be easily extended by various functionalities⁵. Thanks to

⁵The architecture of the YAWL system is described in more detail in Section 6.11.

its service-oriented architecture, the YAWL system nowadays offers two important features that enhance its flexibility to a great extent. These features are YAWL Worklet Service [41, 44, 45], which directly provides for flexibility by underspecification, and YAWL Exception Service [41–43, 208, 209], which represents a powerful mechanism for exception-handling. Moreover, the approach presented in these papers is also implemented into YAWL.

The Worklet Service

The main idea behind the Worklet Service [41, 44, 45] is to dynamically select subprocesses (i.e., *worklets*) that should be invoked in YAWL instances. There are two types of activities in YAWL process models. First, ‘atomic activities’ are activities that should be executed by users. Second, instead of being executed by a user as an atomic activity, an activity can refer to a subprocess (i.e., a ‘worklet activity’). At the execution time, ‘worklet activities’ and relevant instance data are delegated to the Worklet Service. The service then uses a predefined set of *ripple down rules* and the received data to select the most appropriate YAWL process model and automatically invoke it as the selected worklet. Ripple down rules specify which YAWL process model should be invoked as a worklet, given the actual data of the parent instance⁶. In other words, each worklet activity is dynamically decomposed into a YAWL process, which enriches the YAWL system with flexibility by underspecification.

The Exception Service

The worklet paradigm is reused in YAWL to enable a powerful exception handling mechanism realized via the Exception Service [41–43, 208, 209]. The Exception Service provides a fully featured exception handling paradigm for detecting, handling and incorporating exceptions as they occur. This service operates similarly like the Worklet Service, i.e., when an exception occurs in an instance, the Exception Service uses ripple down rules and instance data to select and automatically invoke a YAWL process (i.e., *exlet*) that will be executed in order to handle the exception. Moreover, this implementation allows for handling both predicted and unpredicted exceptions. Unpredicted exceptions are especially interesting: a YAWL user can, at any point during the execution of an instance, report the occurrence of an exception and let the Exception Service invoke a suitable *exlet*.

2.2.4 ADEPT

ADEPT is a workflow management system that focuses on dynamic change. ADEPT was developed at the University of Ulm [189, 191–193, 202]. This system uses powerful mechanisms that allow users to change running instances of process

⁶The Worklet Service is described in more detail in Section 6.11.3.

models by, e.g., adding, deleting or replacing activities or jumping forward in the process [189]. Besides the dynamic change of instances, ADEPT also enables definition of the control-flow, the data-flow, temporal constraints (i.e., minimal and maximal duration of activities, deadlines, etc.) and preplanned exceptions (e.g., forward and backward jumps [190]) in process models, etc. Moreover, the system guarantees static and dynamic correctness properties (e.g., prevents missing input data, deadlocks, etc.) [191]. Therefore, ADEPT offers a comprehensive support for flexibility by change.

Dynamic change is supported in two ways in ADEPT. First, the so-called *ad-hoc* change relates to changing a single running instance [189,191]. Figure 2.7 shows a screen of ADEPT handling a dynamic change. The system offers a complete set of operations for defining dynamic changes at a semantic level and ensures correctness via pre- and post-conditions for changes. Complexity associated with the change (e.g., missing data due to activity deletions) is hidden from users. The second type of run-time change provided by ADEPT is the so-called *propagation* of model changes to its running instances [191–193]. In case of the model change propagation, the change will be applied only to its instances for which the model change does not conflict with the current instance state or previous ad-hoc changes.

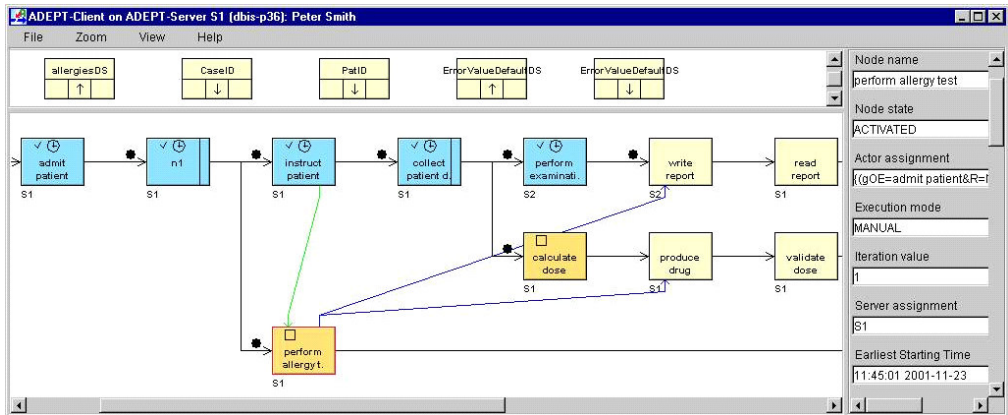


Figure 2.7: Visualizing a dynamic change in ADEPT

Besides the support for dynamic change, ADEPT incorporates other useful features. For example, it considers inter-workflow dependencies and semantical correctness of dynamic change, as described in the following paragraphs.

Most of the workflow management systems do not consider *inter-workflow dependencies* and allow instances to execute independently from each other. However, different instances are often semantically inter-related in some way [126]. ADEPT uses interaction expressions and interaction graphs to enable the specification and implementation of such dependencies [191].

Another useful feature of ADEPT is the semantic check of dynamic changes [164, 165]. For example, in the medical domain it is often the case that certain medications should not be combined. Semantical constraints can be used to define undesired combinations of medications, e.g., activities *administer Aspirin* and *administer Marcumar* should not be executed both because these two medicines are not compatible. Verification of semantic constraints (1) detects situations where these two medications are used together and (2) alerts the user performing the change about this problem. Note that semantic constraints are not necessarily enforced. Instead, an authorized user can commit a change event if it violates a constraint. In this case, the user needs to document the reason for violating the constraint. This approach enables for more flexibility and traceability in situations when problems indeed occur after the change.

2.2.5 Other Systems

There is a variety of workflow management systems available on the market. Although all systems have the same aim, i.e., automating business processes, each system has some unique features. For the purpose of illustration, in this section we will briefly describe three more systems: FileNet, InConcert, and COSA.

FileNet [107] is a conventional workflow management system. Despite its traditional approach to process modeling and execution, FileNet offers the possibility of voting to its users, i.e., it is possible to specify in the model that users should vote during the execution in order to decide the routing of the process. FileNet Ensemble allows on-the-fly adaptations of running instances [201]. Another interesting feature of FileNet is the possibility to monitor states of running instances and perform extensive statistical analysis of past executions. In addition, FileNet offers the possibility to evaluate process models by means of simulation.

InConcert was a workflow management system⁷ built on the ‘workflow design by discovery’ paradigm, which allows for the creation of templates based on the actual execution of instances. The motivation behind InConcert was to tempt the users to design the instance on-the-fly, i.e., while executing it. As an extreme, InConcert allowed for ad-hoc building of instances via ‘free routing processes’, where the created instance is initially empty, and its actual routing is created on-the-fly (cf. Section 2.1.5).

COSA [235, 236] allows for definition of subprocesses and events that trigger them. Process models can be modified at run-time, but the change is applied only for future instances. COSA supports deadlines in a way that on a deadline expiry a compensating activity can be launched. In addition, a compensating activity can also be invoked manually. COSA also allows for run-time deviations by reordering, skipping, re-doing, postponing or terminating activities.

⁷InConcert is not available anymore on the market.

2.3 Workflow Management Systems and the Organization of Human Work

The related work discussed so far originates from the information systems field, i.e., information technology is used to support business processes without much consideration for the role of humans in these processes and the organizational context. In this section we try to provide an overview of related work on the organization of human work. In Section 2.3.1 we describe two contrasting regimes for the organization of work: Autocratic Work Regime (AWR) and Democratic Work Regime (DWR). In Section 2.3.2 we describe structural parameters for AWR and DWR defined by the organizational theory of Socio-Technical Systems (STS) [246]. In Section 2.3.3 we evaluate workflow management systems against the structural parameters, and in Section 2.3.4 we summarize this overview.

2.3.1 Two Contrasting Regimes for the Organization of Work

An ‘autocracy’ is a form of government in which unlimited power is held by a single individual. An Autocratic Work Regime is a practice of management in which there is a strict division of labor by allocating control and execution to separate individuals, i.e., managers and workers. An AWR is further characterized by a hierarchical organization with formal authority; an emphasis on formal, standardized rules; fixed specialized tasks per position; an absolute split into management and technical-support tasks (‘staff’ and ‘line’), and a fragmentation of the executive work into multiple, short-cycled, tasks. In other words, decision making is centralized (cf. Section 1.3) in an AWR. By far the best-known AWRs are the approach to ‘Scientific Management’ [114, 242] and the Classical Organization or ‘Ideal Bureaucracy’ [105, 262]. Although their dominance is fading, these two AWRs still serve as organizational archetypes for both industrial and service organizations.

A ‘democracy’ is a form of government that aspires to serve under ‘the people’ rather than ruling over them. A Democratic Work Regime is defined as a management practice in which people actively take part in the actual decision-making process. In other words, decision making is local (cf. Section 1.3) in a DWR. Two ideal patterns can be distinguished: *representative democracy* and *participative democracy*. In [102], Emery defines *representative democracy* as “choosing by voting from among people who offer themselves as candidates to be our representatives” (page 1). In [103], Emery and Emery define *participative democracy* as “locating responsibility for coordination clearly and firmly with those whose efforts require coordination” (page 100). In a *representative democracy* the influence of people on decision-making is rather indirect. This form, called ‘political participation’, is defined by Abrahamsson [40] as “participation involving the right to control organization’s executive (...) /involvement in high-level goal setting and long-term planning” (pages 186-189) . In a *participative democracy* the

influence of people on decision making is direct. This form, called ‘socio-technical participation’, is defined by Abrahamsson as “participation in the organization’s production, i.e., in the implementation of decisions taken on higher levels” [40]. By far the best-known DWR is the *team-based organization* [99, 231, 244–246].

2.3.2 Socio-Technical Systems

Socio-Technical Systems (STS) [246] is an organizational theory that promotes a Self-Managed Work Team (SMWT) as the prime organizational unit of analysis and design. In STS, *participative democracy* is practiced by giving any potential member of an SMWT the opportunity - as well as the authority - to perform every single task, no matter whether it is executive, managerial, or supportive in character [99, 244–246]. As its name says, STS advocate optimizing benefits from both social and technical aspects of work. We selected STS as a representative organizational DWR theory because it, like workflow technology, extensively considers the operational aspect of flexible work. Moreover, as its name says, STS advocates benefiting from both social and technical aspects in human work [246]. Therefore, we will shortly describe STS and its relation with workflow technology.

Within the STS school, De Sitter et al. identified a set of *structural parameters* that can be used as a typology for the characterization of AWRs and DWRs [231], as Table 2.2 shows. The semantics of each of the parameters will be explained in Section 2.3.3. The structural parameters refer to the basic organization of production (e.g., number of parallel processes), and the various aspects concerning the division of labor (e.g., performance and control). An AWR is characterized by functional concentration, separation of performance and control, performance specialization, performance differentiation, division of control functions, control specialization, and control differentiation [231]. This is best typified by bureaucratic office work in which each employee is doing one single, simple performance task only, for all sorts of different project assignments. Supervisors allocate individual tasks on a daily basis, while specialized technical staff members care for the planning of work and for the administering of quality procedures. A DWR is characterized by functional deconcentration, integration of performance and control, multiple performance integrations, and multiple control integrations. A typical representative of DWRs is self-managed office work, in which teams of employees carry out whole projects by allocating, planning, and controlling full project assignments without additional help of a supervisor or technical specialist.

2.3.3 Workflow Management Systems and the Structural Parameters

To characterize the style of work imposed by workflow management systems, we evaluate these systems against the structural parameters of De Sitter et al.

Table 2.2: Evaluation of AWRs and DWRs with respect to the STS structural requirements of De Sitter [231]

	Socio-Technical requirements	AWR	DWR
1	<i>functional deconcentration</i> (multiple parallel processes)	NO	YES
2	<i>integration of performance and control</i>	NO	YES
3	<i>performance integration A</i> (whole tasks)	NO	YES
4	<i>performance integration B</i> (prepare + produce + support)	NO	YES
5	<i>control integration A</i> (sensing + judging +selecting + acting)	NO	YES
6	<i>control integration B</i> (quality + maintenance + logistics + personnel, etc.)	NO	YES
7	<i>control integration C</i> (operational + tactical + strategic)	NO	YES

[231], as shown in Table 2.3. The evaluation of workflow technology against the structural parameters shows that workflow technology enforces an AWR (cf. Table 2.2), and thus, prevents the functioning of DWRs (e.g., SMWTs).

Table 2.3: Evaluation of workflow management systems with respect to the STS structural requirements of De Sitter [231]

	Socio-Technical requirements	workflow management systems
1	<i>functional deconcentration</i> (multiple parallel processes)	NO: work is repeatedly executed in the same manner.
2	<i>integration of performance and control</i>	NO: people perform and the system controls the work.
3	<i>performance integration A</i> (whole tasks)	NO: people execute specialized, small activities.
4	<i>performance integration B</i> (prepare + produce + support)	NOT APPLICABLE
5	<i>control integration A</i> (sensing + judging +selecting + acting)	NO: people cannot execute a selected control action.
6	<i>control integration B</i> (quality + maintenance + logistics + personnel, etc.)	NOT APPLICABLE
7	<i>control integration C</i> (operational + tactical + strategic)	NO: the system is in charge of the operational control.

Functional deconcentration. This parameter refers to grouping and coupling of performance functions (process models) with respect to work orders [231]. If all orders undergo the same procedure, then we talk about *function concentra-*

tion [231]. If, due to their variety, orders undergo different procedures, then we talk about *functional deconcentration* [231]. Traditional workflow management systems lack flexibility and force people to repeatedly execute their work in the same manner [77, 109, 125, 143, 153, 166, 188, 233]. Therefore, business processes are constantly executed in the same way, regardless of the nature of work orders.

Integration of performance and control. In a DWR, the same people who perform the work are also authorized and responsible for control [231]. This so-called integration of performance and control is not possible in a conventional workflow management systems because, due to lack of flexibility, system users cannot influence the way they work. Instead, process models that prescribe the way people execute their work is developed by external experts [77, 109, 125, 143, 153, 166, 188, 233].

Integration into whole tasks. Instead of specialized, short-cycled tasks, STS advocates whole tasks that form a meaningful unit of work [231]. When working in DWRs, people deal with more variety in their work. However, in workflow management systems, a business process is represented by a large process model consisting of individual activities [29, 66, 93, 109, 110, 125, 266]. To this end, bigger, meaningful, units of work are divided into separate, short-cycled tasks that should be executed by authorized individuals. Working with conventional workflow management systems implies performance specialization. So, any form of performance integration is lacking.

Integration of preparation, production, and support. Preparation, production and support functions must be integrated at the workplace level [231]. Workflow technology is used to support only the production function [29, 93, 266]. Preparation and support functions are allocated elsewhere within the company, and workflow technology does not influence this integration. Therefore, this parameter is considered to be not applicable in the evaluation of workflow technology.

Integration of control functions: sensing, judging, selecting, and acting. The functions of a control cycle are: (1) sensing the process states, (2) judging about the need for a corrective action, (3) selecting the appropriate correction action, and (4) acting with the selected control action [231]. In a DWR, the four control functions should be integrated [231]. Although, when working with a workflow management system, people can sense the need for control and are able to successfully judge any controls needed, they do not have the authorizations and/or possibilities to select and execute the appropriate control activities [109, 125]. Due to lack of flexibility of workflow management systems, it

is not possible to successfully integrate the control functions when working with such a system.

Integration of the control of quality, maintenance, logistics, personnel, etc. Control of quality, maintenance, logistics, personnel, etc. should be conducted at the workplace level [231]. Workflow technology is used to support only the production function [29, 93, 266]. Control of quality, maintenance, logistics, and personnel are allocated elsewhere within the company, and workflow technology does not influence this integration. Therefore, this parameter is considered to be not applicable in the evaluation of workflow technology.

Integration of operational, tactical and strategic controls. Operational, tactical and strategic controls should be integrated at the workplace level [231]. Independently of the workflow technology, an organization can integrate (or not) operational, tactical, and strategic controls. Although the use of a workflow management system does not explicitly influence tactical and strategic control, it prevents this control integration at the workplace level because workers are not made responsible for the operational control [77, 109, 125, 143, 153, 166, 188, 233]. In this case, operational control is external - i.e., managers and business-process modelers control the operational design of work. Therefore, this integration cannot be established at the workplace level and this parameter is not supported by conventional workflow management systems.

2.3.4 Summary

This section provided an overview of the work related to the requirements for flexible style of human work in the organizational context. It showed that, despite the fact that there is not much work that combines the fields of IT and organizational science, the lack of flexibility of workflow technology indeed disables DWRs advocated by many organizational theories, like, e.g., STS and SMWTs [246].

2.4 Outlook

In this chapter we presented the research conducted in the area of flexibility of workflow management systems. As indicated by many researchers, workflow management systems lack flexibility due to the fact that users cannot adjust the execution of processes to requirements imposed by specific situations. We also described why inflexible systems prevent implementation of democratic regimes of work in practice. We used a taxonomy of features that enhance flexibility of workflow management systems to classify the relevant work in this field: (1) design of flexible process models, (2) underspecification in process models, (3) change of running processes, and (4) deviation from prescribed process models.

Although much research has been done in each of these areas, a unique approach that unifies all relevant features is still lacking. In this thesis, we propose a new, constraint-based, approach to workflow management systems which primarily aims at enhancing flexibility by design, but also enables all other types of flexibility (i.e., flexibility by underspecification, change, and deviation). First we define the class of constraint-based process modeling languages on an abstract level and one concrete formal language for constraint specification. Then we present a ‘proof of concept’ prototype that shows how the proposed constraint-based approach can be applied to workflow management systems.

Chapter 3

Flexibility of Workflow Management Systems

Workflow management systems influence to a great extent the way employees execute their work. As discussed in sections 1.3 and 2.3., modern organizational theories advocate democratic work regimes where people can control their work. In order to be able to support this kind of work, workflow management systems must offer a high degree of *flexibility*. Flexibility of workflow management systems represents the degree to which users can choose how to do their work, instead of having a workflow management system decide how to work [125, 226–228]. In this chapter we describe contemporary workflow management systems and the features that enhance flexibility of these systems.

The remainder of the chapter is structured as follows. First, in Section 3.1 we describe the functionality of contemporary workflow management systems based on the three dominant workflow perspectives: the control-flow, the resource and the data perspective. Second, in Section 3.2 we illustrate the flexibility of contemporary workflow management systems using simple, system and language-independent examples. Finally, in Section 3.3 we conclude this chapter by proposing a new approach to process modeling that is able to offer all types of flexibility.

3.1 Contemporary Workflow Management Systems

Despite the complexity of workflow management systems and the high impact they have on business processes, a good standardization is still lacking in the area of workflow technology. Vendors of workflow management systems tend to offer different functionalities in their systems. A standardization is also lacking with respect to the terminology used in workflow technology. On the one hand, the same concepts often have different names in various systems, which creates an illusion that a particular functionality is different in systems. On the other hand,

it might happen that systems use the same name for different concepts or functionalities. In this section we describe in the main concepts and functionalities of contemporary workflow management systems.

The Workflow Management Coalition (WFMC) [9] aims at standardizing workflow technology. One of the efforts of the WFMC in the direction of standardization was proposing the reference model for general architecture of workflow management systems [75]. The WFMC's reference model shows many possible components of workflow management systems. However, three of those components are the core of every system: a *process definition tool*, a *workflow engine*, and a *workflow client application*. These three components are shown in Figure 3.1.

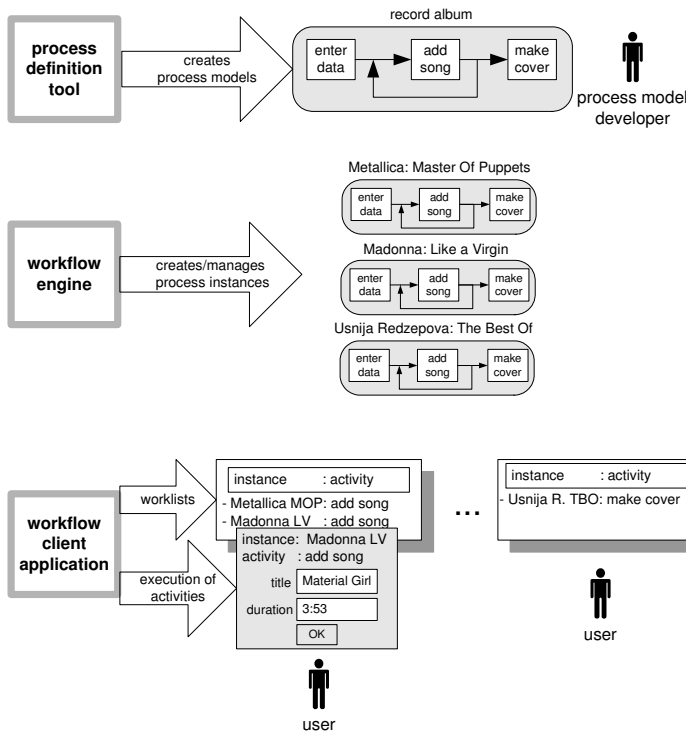


Figure 3.1: The three main components of a workflow management system

First, a *process definition tool* is used by process model developers to create process models. For example, Figure 3.1 shows that a model for the *record album* process can be developed using such a tool. Second, a *workflow engine* is needed to manage the execution of instances of process models. In the example presented in Figure 3.1, this ensures that each artist can record an album in the way it is prescribed in the *record album* model. Based on the definition of a process model, the *workflow engine* decides which activity(-ies) can be executed by which users and at what point in time. Third, a *workflow client application*

presents the so-called worklist (i.e., a list of all activities that can be executed in running instances) to each user. Using this tool each user can execute activities available in the worklist. In Figure 3.1 we see two users: (1) the worklist of the first user contains two activities *add song* (i.e., for two running instances), (2) the worklist of the second user contains activity *make cover* for the third running instance, and (3) the first user is currently executing activity *add song* for one of the instances. Each time a user executes an activity in an instance, the *workflow engine* decides, based on the process model and the current state of the instance, which activities can be executed next, and updates the worklists of all users with this information. For example, people working on any of the three instances presented in Figure 3.1 will be able to execute activity *add song* only after executing activity *enter data*.

Figure 3.2 shows the three main perspectives of process models: the control-flow perspective, the resource perspective, and the data perspective. These three perspectives determine the order in which activities will be executed, which users can execute which activities, and which information will be available during execution. The *control-flow perspective* of a process model defines in which order activities can be executed [29,35,208,213]. For example, Figure 3.2(a) shows that the control-flow perspective of the process model *record album* specifies that (1) the process starts with activity *enter data*, (2) followed by an arbitrary number of executions of activity *add song*, and (3) the process ends by executing activity *make cover*.

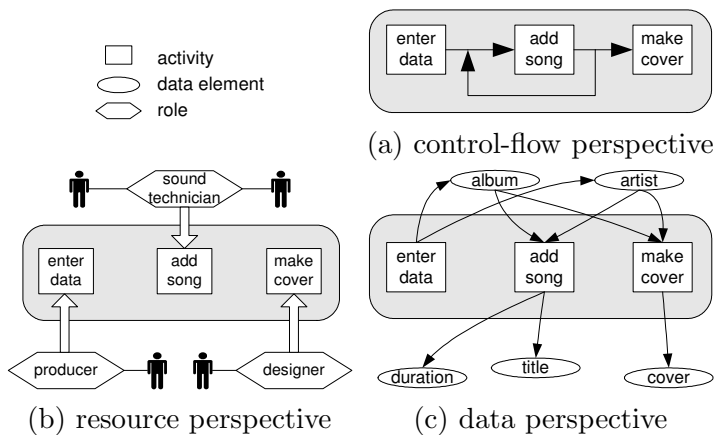


Figure 3.2: The three perspectives of process models

The *resource perspective* defines which (human) resources are authorized to execute each of the activities and how the actual resources are allocated to execute the activities [10, 29, 35, 106, 208, 211, 216]. Figure 3.2(b) shows the resource perspective of the process model *record album* with four users having three roles. The *producer* can *enter data*, each of the two *sound technicians* can *add song*

and the *designer* can *make cover*. If a user has the appropriate role to execute an activity, then we say that the user is *authorized* to execute the activity. Naturally, an activity that is supposed to be executed next will be offered only to the worklists of authorized users.

The *data perspective* of a process model defines which data elements are available in the process and how users can access them while executing activities [29, 208, 210, 212]. Figure 3.2(c) shows the data perspective of the *record album* process model. First of all, there are five data elements in this process, i.e., *artist*, *album*, song *title*, song *duration* and *cover*. Second, for each activity it is defined (1) which data elements are available (e.g., *album*, *artist*, *duration* and *title* are available in activity *add song*) and (2) how these data elements can be accessed (e.g., data elements *album* and *artist* can be seen but not edited while *duration* and *title* can be edited in activity *add song*). If the value of a data element can be accessed but not edited in an activity, then we say that this is an *input data element* for this activity. If the value of a data element can be edited in an activity, then we say that this is an *output data element* for this activity. Figure 3.2(c) shows that, for activity *add song*: (1) *album* and *artist* are input data elements and (2) *title* and *duration* are output data elements.

In the remainder of this section each of the perspectives is discussed in detail: the control-flow perspective in Section 3.1.1, the resource perspective in Section 3.1.2, and the data perspective in Section 3.1.3. Each of these three sections starts with a short description and it is organized as follows.

CPN models. First, we present the perspective in a system-independent way using *Colored Petri Nets* (CPNs) models [1, 138, 139, 152]. CPNs are an extension of classical Petri nets [199]. There are several reasons for selecting CPNs as the language for modeling in the context of workflow management: (1) CPNs have formal semantics and are independent of any workflow system, (2) CPNs are executable and allow for rapid prototyping, gaming, and simulation, (3) CPNs have a graphical representation and their notation is intuitively related to existing workflow languages, and (4) the CPN language is supported by CPN Tools – a graphical environment to model, enact and analyze CPNs.

Workflow management systems. Second, we present how the perspective is realized in three commercial workflow management systems: Staffware [238], FileNet [107] and FLOWer [180]. The goal is to provide insight into the functionality and look-and-feel of contemporary systems. As discussed in Section 2.2, Staffware and FileNet are two typical examples of traditional workflow management systems, while FLOWer is a case-handling system [195]. As such, these three systems provide a good overview.

Workflow patterns. Third, several *patterns* are described for the perspective in order to present the perspective in an system-independent way. In an attempt to identify unified solutions of standard issues in workflow tech-

nology, the *workflow patterns initiative* [10, 35] identified many workflow patterns [208, 211, 213]. In addition to the three basic perspectives (i.e., control-flow, resource and data) patterns are also identified for the exception handling perspective. Workflow patterns can be used for “examining the suitability of a particular process language or workflow system for a particular project, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular process-aware information system, and as a basis for language and tool development” [10]. Note that a large number of patterns is identified for each of the perspectives. However, due to page limits we present only few of the patterns for the illustration purpose. However, insights obtained through the complete set of patterns are used throughout this thesis.

Overview. Finally, a summarized overview of the perspective is given.

3.1.1 The Control-Flow Perspective

Despite the different languages (i.e., notations) used in various commercial and academic tools, the control-flow perspective plays an important role in process models because it determines the order in which users can execute activities.

CPN Model(s)

For illustration purposes we will use a simple example of the *Handle Complaint* process [29]. Figure 3.3 shows the CPN model of the *Handle Complaint* process. A CPN model consists of *places* and *transitions* connected by *arcs*. Places (represented by ovals) are typed, i.e., the tokens in a place have values of a particular type (or color in CPN jargon). These types are a subset of the default data types in Standard ML such as integer and string and additional types can be composed using constructs such as tuple, list and record. The number of tokens per place can vary over time. The value of a token indicates the properties of the object represented by this token. There are nine places in the CPN shown in Figure 3.3 and all of them are of the type *ID*, which stands for the complaint identification number. Transitions (represented by rectangles) may consume tokens from places and produce tokens in places, as specified by inscriptions on arcs between places and transitions. There are seven transitions in the CPN in Figure 3.3, i.e., *start*, *contact department*, *contact client*, *assess*, *pay*, *send letter* and *file*. A more detailed discussion of the CPN concepts is beyond the scope of this paper. In the remainder, we assume that the reader is familiar with the CPN language and refer to [1, 138, 152] for more details.

The CPN in Figure 3.3 defines the control-flow of the *Handle Complaint* process. After receiving a complaint from a client and *starting* the process, the officer can in parallel (i.e., in any order) *contact the client* and *contact the*

department to collect information about the conforming. After gathering this information, the officer *assesses* the complaint. If the assessment is positive, the complaint is accepted and the department *pays* the client. If the assessment is negative, a notification *letter is sent* to the client. At the end of the process, the complaint and the assessment result are *filed* in the archive.

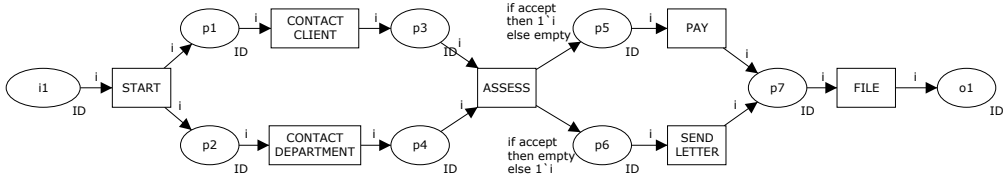


Figure 3.3: CPN for the *Handle Complaint* process

Workflow Management Systems

Workflow management systems tend to use system-specific notations (i.e., languages) for specifying the control-flow perspective in process models. Figures 3.4, 3.5 and 3.6 show the control-flow perspective of the *Handle Complaint* process in three commercial workflow management systems: Staffware, FileNet and FLOWer, respectively. Indeed, this process is modeled differently in these three systems: while Staffware and FileNet present the whole model on a single level, modeling decisions in FLOWer (i.e., to execute activity *pay* or *send letter*) are typically modeled on a separate level in the model. Also, the three systems present the model using different graphical elements and styles.

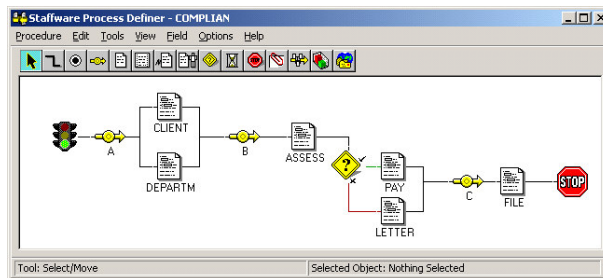


Figure 3.4: *Handle Complaint* process in Staffware

Patterns

Control-flow patterns [35, 208, 213] represent typical constructs that can occur in process models. The twenty initial patterns presented in [33–35] were revised and extended with twenty three new patterns in [213]. For illustration purposes,

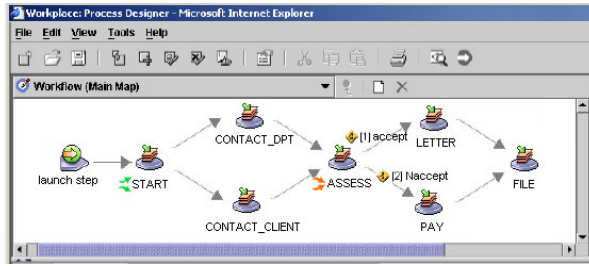
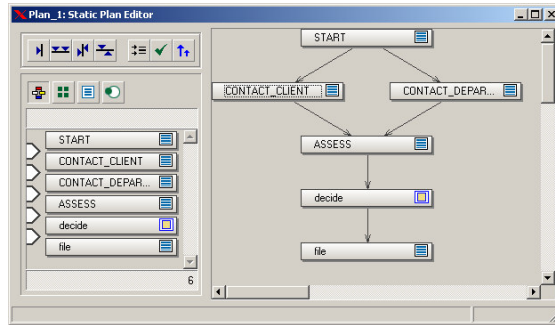


Figure 3.5: *Handle Complaint* process in FileNet



(a) main model



(b) decision

Figure 3.6: *Handle Complaint* process in FLOWer

we first describe four simple patterns and present them using CPN models in Figure 3.7.

First, the *sequence* pattern is used to specify that an activity is enabled after the completion of a preceding activity [213]. Figure 3.7(a) shows a CPN model representing a sequence of two activities A and B [213]. Second, the *parallel split* pattern represents the divergence of one control-flow thread into two or more branches that execute concurrently [213]. Figure 3.7(b) shows a CPN model representing a *parallel split* after activity A into parallel branches with activities B and C [213]. For example, the *Handle Complaint* process starts with a parallel split into two branches, i.e., *contact department* and *contact client*. Third, the *synchronization* pattern represents the convergence of two or more input branches into a single output thread only after the activities in all input branches have been completed [213]. Figure 3.7(c) shows a CPN model representing the synchronization of two branches containing activities A and

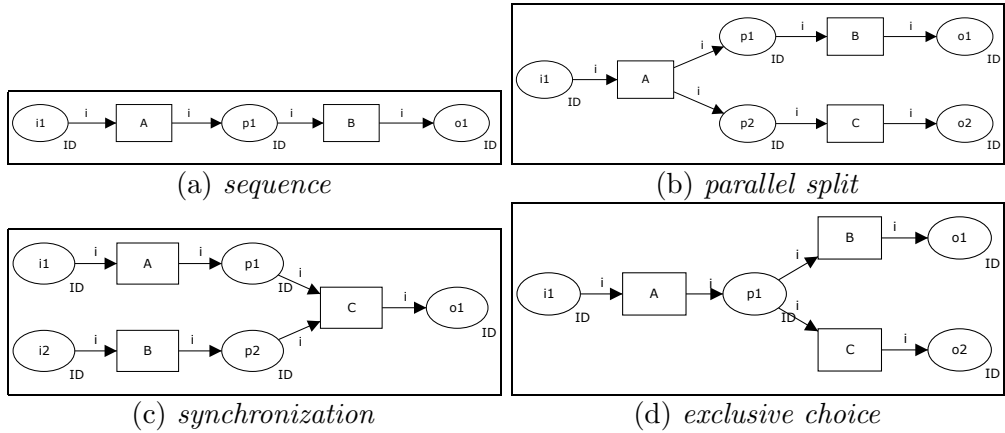


Figure 3.7: CPN models of several control-flow patterns [213]

B into a single control-flow thread containing activity C [213]. For example, activities *contact department* and *contact client* in the *Handle Complaint* process are synchronized into one control thread containing activity *assess*. Finally, the *exclusive choice* pattern represents divergence to two or more branches, where the thread of control is passed to only one outgoing branch [213]. Figure 3.7(d) shows a CPN model representing an exclusive choice between activities B and C after activity A [213]. The *Handle Complaint* process contains exclusive choice between activities *pay* and *send letter* after activity *assess*.

Some of the control-flow patterns are much more complex than the patterns shown in Figure 3.7. Consider, for example, the *blocking discriminator* pattern that represents a kind of the so-called *1-out-of- M join* [213]. This pattern represents a situation when two or more (i.e., M) branches join into a single control-flow branch. For example, “when handling a cardiac arrest, the check breathing and check pulse activities run in parallel. Once the first of these has completed, the triage activity is commenced. Completion of the other activity is ignored and does not result in a second instance of the triage activity” [213]. The CPN model of this pattern is presented in Figure 3.8: here we can see how two branches (i.e., transitions $A1$ and A_m) are joined into one branch with transition B .

In addition to identifying 43 control-flow patterns, several commercial workflow systems are evaluated in [35, 213] based on the pattern support. The evaluation results [213] of control-flow pattern support in Staffware, FileNet and FLOWer is given in Appendix B.1 of this thesis. For example, each of the three systems (i.e., Staffware, FileNet and FLOWer) supports *sequence*, *parallel split*, *synchronization* and *exclusive choice* patterns (cf. Figure 3.7) and none of them supports the *blocking discriminator* pattern (cf. Figure 3.8), as shown in Table 3.1. Evaluation of systems shows that, although different systems tend to

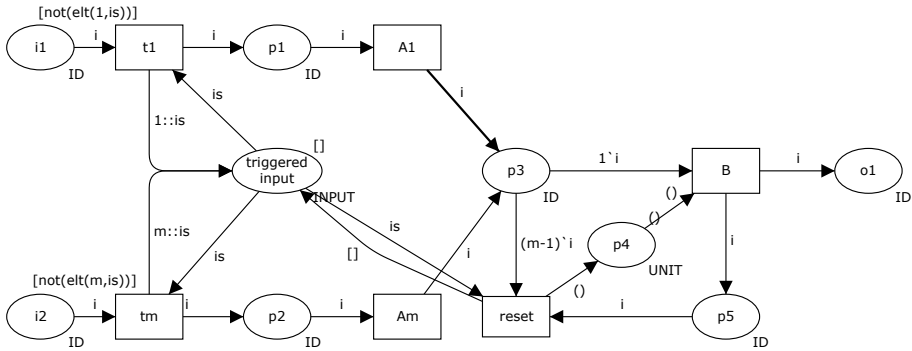


Figure 3.8: CPN model of the *blocking discriminator* pattern [213]

support different patterns, older systems tend to support only approximately half of identified patterns [35, 213]. For example, Staffware fully supports 14 patterns [35, 213]. FileNet fully supports 17 patterns and partially supports one pattern [35, 213]. FLOWer fully supports 16 patterns and provides partial support for 8 additional patterns [35, 213]. The fact that none of the systems supports all patterns reflects the diversity of the way various workflow management systems handle the control-flow perspective.

Table 3.1: Support for some control-flow patterns in Staffware, FileNet and FLOWer

Pattern	Staffware	FileNet	FLOWer
sequence	+	+	+
parallel split	+	+	+
synchronization	+	+	+
exclusive choice	+	+	+
blocking discriminator	-	-	-

(+ = support, - no support)

Introducing standards into the workflow technology remains an important challenge in the field. For example, BPEL [53, 54, 178] is one of the most popular initiatives to standardize in the workflow technology by proposing a standard language and its execution framework. Although it is widely accepted as a standard by both industry and research, BPEL does not support all the control-flow patterns. In fact, BPEL supports 17 patterns directly, 4 patterns only partially and does not provide any support for 22 patterns [35, 213].

Overview

The control-flow perspective of current workflow management systems typically has a *procedural* nature. In other words, process models are constructed using control-flow patterns which specify in detail the exact procedure of how the work

should be done. The procedural nature of process models is suitable for highly structured processes with a high repetition rate, i.e., when the work is repeatedly done in the same manner (cf. Section 1.2). However, when it comes to processes that should be controlled by users (i.e., people can choose how to work), the procedural process models become too complex. Consider, for example, the *branched discriminator* pattern presented in Figure 3.8. Even though it is applied in rather simple situations, many systems do not support this pattern.

The control-flow perspective of current systems implies detailed specification of exactly how the control flows through the model. This makes it very hard or even impossible to specify more ‘relaxed’ concepts that people use in their work. For example, there is no control-flow pattern that would specify that two (or more) activities should never be executed both in the same process instance, regardless of how often and at which point of time one of them is executed. Consider, for example, a medical process that contains (amongst others) activities *examine prostate* and *examine uterus*. Regardless of the fact if these operations are executed at all and how many times, they cannot be both executed for one patient. In the best case, implementing this simple requirement in a procedural model using control-flow patterns would require an extensive ‘work-around’ resulting in a complex model.

3.1.2 The Resource Perspective

After the system makes a decision which activities are the next in line to be executed based on the control-flow specification, the resources that can/will execute these activities are selected based on the resource perspective. The resource perspective depends on (1) how resources and their roles are defined and classified in the system and (2) how the system decides who and when can execute enabled activities. We refer to the mechanism that handles the resource perspective in a system as to the *work distribution* of the system. Just like it is the case with the control-flow perspective, the resource perspective is handled differently in different workflow management systems due to system-specific work distribution mechanisms. Note that we present the resource perspective in a more detailed manner than the control-flow perspective. The reason is that less attention has been devoted to the resource perspective in workflow literature. Therefore, it is worthwhile to discuss this perspective in more detail.

CPN Model(s)

We have developed a CPN model that represents a simple work distribution mechanism of a generic and simple workflow management system. We refer to this model as to the *basic model*. Colors presented in Table 3.2 are used in the *basic model* to represent the main concepts of workflow management systems. An activity is represented as a string carrying the name of the activity and an

instance as a number identifying the instance. A work item is a combination of an instance identifier and activity name, i.e., it represents an activity that needs to be executed for an instance. Each user, role and group is represented as a string carrying the object name. While a role represents qualifications of users (e.g., secretary), a group represents an organizational department (e.g., sales).

Table 3.2: CPN colors representing basic workflow concepts

colset ACTIVITY = string;	colset USER = string;
colset INSTANCE = int;	colset ROLE = string;
colset WI = product INSTANCE*ACTIVITY;	colset GROUP = string;

A life cycle model of a work item shows how a work item changes states during the work distribution [29, 91, 93, 136, 160, 175]. The *basic model* uses a simple model of the life cycle of work items and it covers only the general, rather simplified, behavior of workflow management systems (e.g., errors and aborts are not considered). Figure 3.9 shows the life cycle of a work item in the *basic model*. After the *new* work item has arrived, it is automatically also *enabled* and then taken into distribution (i.e., state *initiated*). Next, the work item is *offered* to the user(s). Once a user *selects* the work item, it is *assigned* to him/her, and (s)he can *start* executing it. After the *execution*, the work item is considered to be *completed*. This may trigger new work items based on the control-flow perspective of the model and the user can begin working on the next work item.

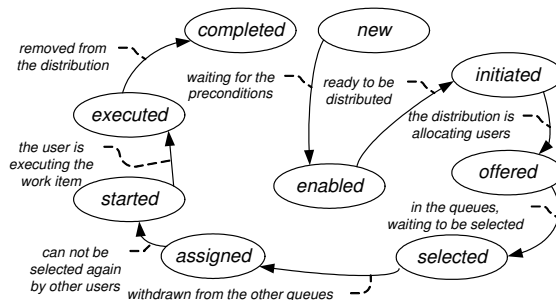


Figure 3.9: *Basic model* - work item life cycle

To simulate (execute) the work distribution model in the CPN tools, it is necessary to initiate the model by defining *input elements*. Table 3.3 shows the four input elements of the *basic model*. For every input element in Table 3.3 the element name is shown (i.e., *system users*, *new work items*, *activity maps* and *user maps*). Besides the name, there are a short description of the element, the CPN color that represents the element and a simple example showing a possible initial element value. Figure 3.10 shows input elements from Table 3.3 graphically. First, there are two system users (i.e., *Mary* and *Joe*), two roles (i.e., *secretary* and *manager*) and one group (i.e., *sales*). *User maps* define which users have

which roles and to which groups they belong to. For example, in the *user maps* it is specified that *Mary* has the role of the *secretary* in the *sales* department. *Activity maps* define what role and group a user needs to have in order to be able to execute an activity. For example, in the *activity maps* it is specified that activity *contact client* can be only by users that have role *secretary* and belong to group *sales*. Initial available work items are shown as *new work items* in Table 3.3. For example, work items for activities *contact department* and *contact client* from the instance with identification *1* are initially available in the work distribution.

As a model of an abstract workflow management system, we have developed the *basic model* on the basis of three simplifying assumptions: (1) we abstract from the control-flow perspective (i.e., how the system decides which activities are enabled and creates work items for them), (2) we only consider the ‘normal’ behavior (i.e., work items are completed successfully; errors and aborts are not included), and (3) we abstract from the user interface.

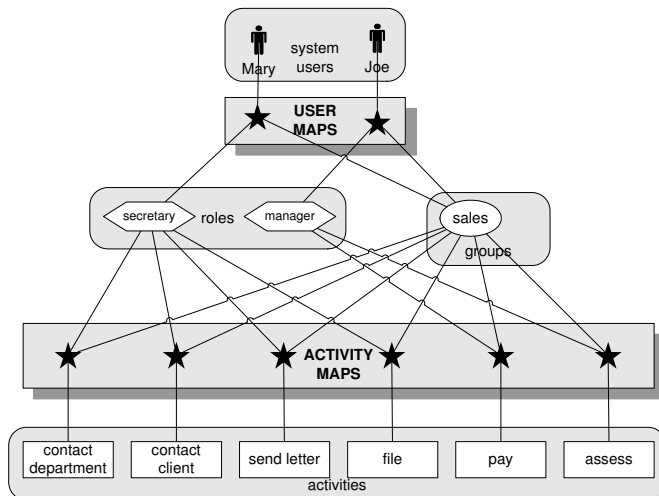
The *basic model* is organized into two modules: the *work distribution* and the *work lists* module, as shown in Figure 3.11. The CPN language allows for the decomposition of complex nets into sub-pages, which are also referred to as sub-systems, sub-processes or modules. By using such modules we obtain a layered hierarchical structure. The two modules communicate by exchanging messages via six places. These messages contain information about a user and a work item, i.e., each place is of the type ‘user work item’ ($colset\ UWI = product\ User * WI$).

Table 3.4 shows the description of the semantics of different messages that can be exchanged in the model. For each message the name of the referring CPN place is given in the first column and a short description in the second column.

The *work distribution* module manages the distribution of work items by making sure that work items are executed correctly. This module allocates (identifies) users to whom the *new work items* should be offered, based on authorization (*AMap*) and organization (*UMap*) data. Figure 3.12(a) shows the *work distribution* module. The new work items are determined as input values (i.e., initial marking) generated based on the control-flow perspective in place *new work items*. The first to fire is the transition *offers*, which uses the function *offer* to decide to which user the work item should be offered and creates user work items in place *to be offered*. For a given activity, this function first retrieves the authorized role and group from *amaps* in place *activity map* and then retrieves the authorized user(s) with this role and group from *umaps* in place *user map*. As a result a message is sent to the *work lists* module to offer the work item to selected users. Although in the *basic model* users authorized to execute an activity are the users that have the role and are in the group specified in the *amaps* for the activity, this criterion may vary from system to system. The *work lists* module sends a message that a user wishes to select a work item by placing a user work item token in place *selected*. The message (token) contains the infor-

Table 3.3: Input for the *basic model*

system users	a set of available users; CPN color: example:	colset Users = list User; $iUser = 1'Mary++1'Joe;$
user maps	the organizational structure is used to map users to organizational entities such as roles and groups; CPN color: example:	colset UMap = product User * Roles * Groups; (where colset Roles = list Role; colset Groups = list Group;) $iUMaps = [(Mary, [secretary], [Sales]),(Joe, [manager], [Sales]);$
activity maps	for every activity authorization is defined with a role and a group; CPN color: example:	colset AMap = product Activity * Role * Group; $iAMaps = [(contact department, secretary, Sales), (contact client, secretary, Sales), (assess, manager, Sales),(send letter, secretary, Sales),(pay, manager, Sales),(file, secretary, Sales)];$
new work items	work items that have arrived and are ready to be distributed to users; CPN color: example:	colset WI = product Instance * Activity; $iWI = 1'(1,contact department)++1'(1,contact client);$

**Figure 3.10:** Graphical illustration of the *basic model* input from Table 3.3

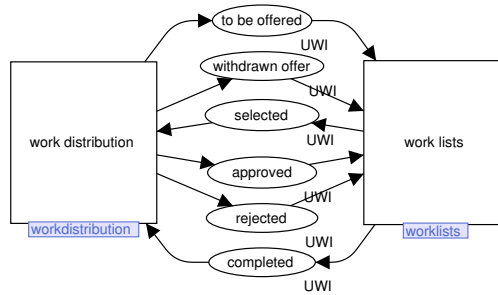


Figure 3.11: *Basic model*

Table 3.4: Exchange of messages between modules *work distribution* and *work lists*

place	message
<i>to be offered</i>	A work item is offered to the user.
<i>withdrawn offer</i>	Withdraw the offered work item from the user.
<i>selected</i>	The user requests to select the work item.
<i>approved</i>	Allow the user to select the work item.
<i>rejected</i>	Do not allow the user to select the work item.
<i>completed</i>	The user has completed executing the work item.

mation about the work item and the user that requests to select it. If the related work item already has been selected, transition *reject* cancels this request. If not, transition *selects* transfers the user work item from place *offered work items* to place *assigned work items*, approves the request from the *work lists* by putting a token in place *approved*, and withdraws all the other offers for the related work item via place *withdrawn offer*. Finally, when the user *completes* a work item, the related token appears in place *completed*. Transition *completes* matches the user work item tokens in places *completed* with tokens in place *assigned work items*, removes them from those two places, and produces the referring user work item token in place *completed work items*. This user work item is considered to be completed by the user, and it is archived as a closed work item.

Figure 3.12(b) shows the *work lists* module. This module receives messages from the *work distribution* module regarding work items that need to be offered to specified users. The *work lists* module further manages events associated with the activities of users. It is decomposed into three sub-modules, which correspond to three basic actions users can perform: *log on and off* (cf. Figure 3.12(c)) in the system, *select work* (cf. Figure 3.12(d)), *start work* (cf. Figure 3.12(e)), and *stop work* (cf. Figure 3.12(f)). In the *log on and off* sub-module (cf. Figure 3.12(c)) every user can freely choose when to log ‘on’ (transition *log on*) to or ‘off’ from (transition *log off*) the system. Users who are currently logged-on to the system are represented as tokens in place *logged on* and users who are currently logged-

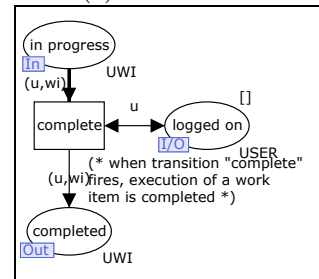
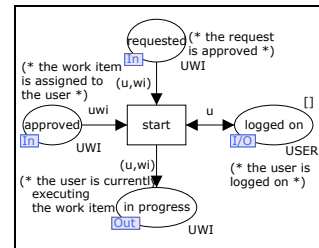
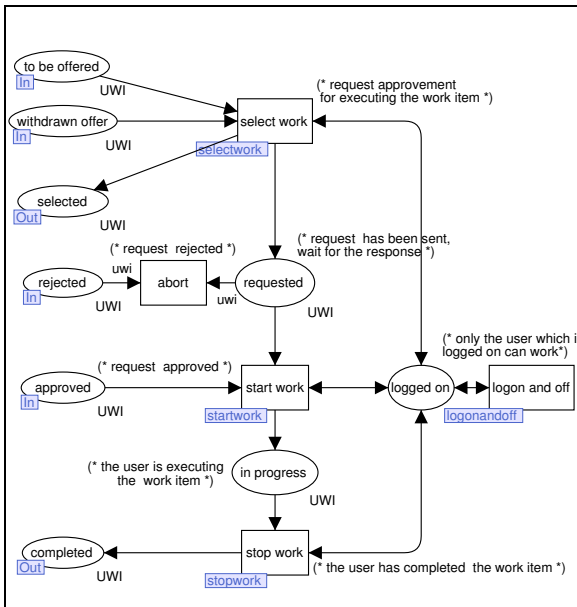
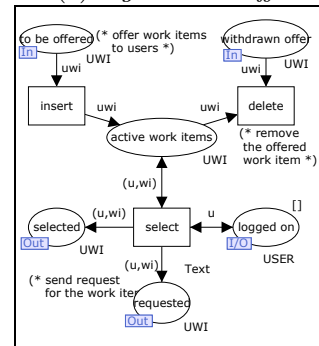
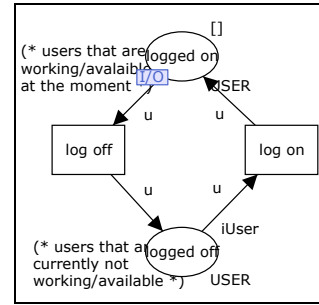
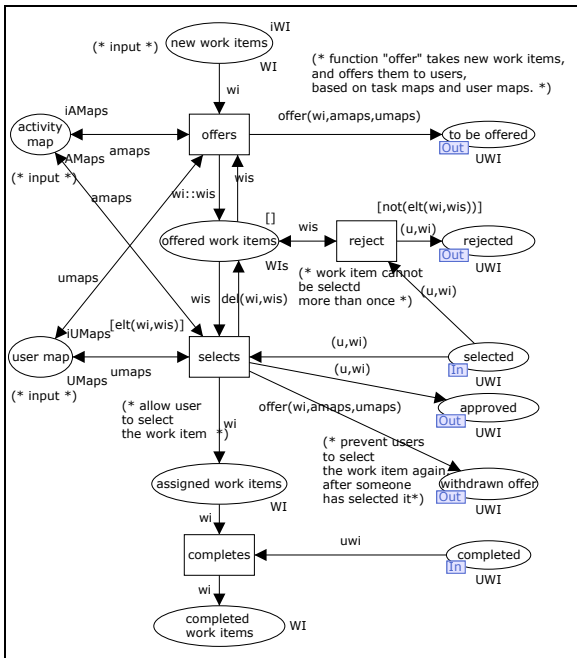


Figure 3.12: Modules of the basic model

off from the system are represented as tokens in place *logged off*. The *select work* sub-module (cf. Figure 3.12(d)) automatically fires transition *insert* and moves the user work item token from place *to be offered* to place *active work items*. If the work item is withdrawn, the token is removed from place *active work items*. When a user wishes to select a work item, transition *select* fires creating a token in place *selected* (to send a request to the *work distribution* module) and archives this request by creating a token in place *requested*. Note that only users that are *logged on* to the system can select work items. The *work lists* module (cf. Figure 3.12(b)) proceeds with the user work item in place *requested* following one of the two alternative scenarios. First, if a message (user work item token) arrives at place *rejected*, transition *abort* automatically fires and removes the token from places *rejected* and *requested*. Second, if a message (user work item token) arrives at place *approved*, the user can select the work item and further flow is directed to the *start work* sub-module. In the *start work* sub-module (cf. Figure 3.12(e)) transition *start* removes the user work item token from places *requested* and *approved* and creates a token in place *in progress*. Note that only users who are currently logged-on to the system can start work items. Users that are logged-on to the system can *complete* a work item that by removing a token from place *in progress* and creating a token in place *completed* (cf. Figure 3.12(f)).

Workflow Management Systems

In order to analyze work distribution of Staffware, FileNet and FLOWer, we have developed a CPN model of work distribution mechanism for each of them. These three systems (and workflow management systems in general) tend to use different work distribution concepts and completely different terminologies. To maintain a common basis for the models of work distribution in Staffware, FileNet and FLOWer, we have extended the *basic model* for the three specific systems. Due to the size and complexity of work distribution models for Staffware, FileNet and FLOWer [182], we present these models in Appendix A of this thesis. The work distribution CPN models of these systems indeed show that, although some concepts are represented and named differently in the systems, they are, actually, very similar (cf. Appendix A). On the other hand, work distribution and the related CPN model of FLOWer is more complex, due to the fact that users have many more actions available (i.e., execute, open, skip, undo and redo work items) when working with this case-handling system, as shown in Appendix A.3.

Patterns

The workflow resource patterns [10, 35, 208, 211, 216] capture the various ways in which resources are represented and utilized in workflows. In this chapter we do not elaborate on each of the 43 patterns described in [216], but we discuss four of them for the purpose of illustration. None of the modeled systems (i.e.,

Staffware, FileNet and FLOWer) supports patterns *round robin*, *shortest queue*, *piled execution*, and *chained execution* (cf. Appendix B.2). *Round robin* and *shortest queue* are push patterns, i.e., a work item is offered to only one user who has to execute it. As auto-start patterns, *piled execution*, and *chained execution* enable the automatic start of the execution of the next work item once the previous has been completed.

The *round robin* and *shortest queue* patterns push the work item to only one user of all users that qualify. The *round robin* pattern allocates work on a cyclic basis and the *shortest queue* pattern allocates to the user with the shortest queue in his/her worklist. This implies that each user has a counter to: (1) count the sequence of allocations in *round robin* and (2) count the number of pending work items in *shortest queue*. Figures 3.13 shows that these two patterns can be implemented in a similar way in the *work distribution* module of, e.g., the *basic model*. The required changes to the *basic model* are minimal. A counter is introduced for each user as a token in place *available* (*colset UCounter = product User * INT; colset UCounters = list UCounter*) and functions *round robin* and *shortest queue* are used to select one user from the set of possible users based on these counters. These allocation functions are used in the inscription on the arc(s) between the transition *offers* and place *to allocate*. Both functions take two parameters: (1) user work items created by the ‘classical’ allocation function *offer* from the *basic model*, and (2) appropriate counters. Both functions allocate the work item to the right user via three steps: (1) take the set of user work items created by the allocation function *offer*, (2) for every user work item search for the value of the counter, and (3) select and return only the user work item where the user has the smallest value of the counter. In this way, ‘push allocation functions’ can be seen as a filter that selects only one allocation from of the set of all possible allocations. The model for *shortest queue* has an additional connection (i.e., the two arcs between the transition *complete* and place *available*) that updates the counter when a work item is completed to remove it from the queue (decrease the value of the counter for the referring user).

Piled execution and *chained execution* are auto-start patterns, i.e., when a user completes the execution of current work item the next work item starts automatically. This prevents the user from repeatedly switching between worklist and application for routine tasks. When working in *chained execution*, the next work item will be for the same *instance* as the completed one – the user works on different activities for one instance. Similarly, if the user works in *piled execution* the next work item will be for the same *activity* as the completed one – the user works on the same activity for different instances. Figures 3.14(a) and 3.14(b) show that *piled execution* and *chained execution* are implemented similarly in the *stop work* sub-module. Users can choose to work in the normal mode or in the auto-start mode (which is represented by the token in place *special mode*). A parameter *x* is passed via the arc between place *ready* and transition *complete special*: parameter *x* carries activity name in *piled execution* or instance identi-

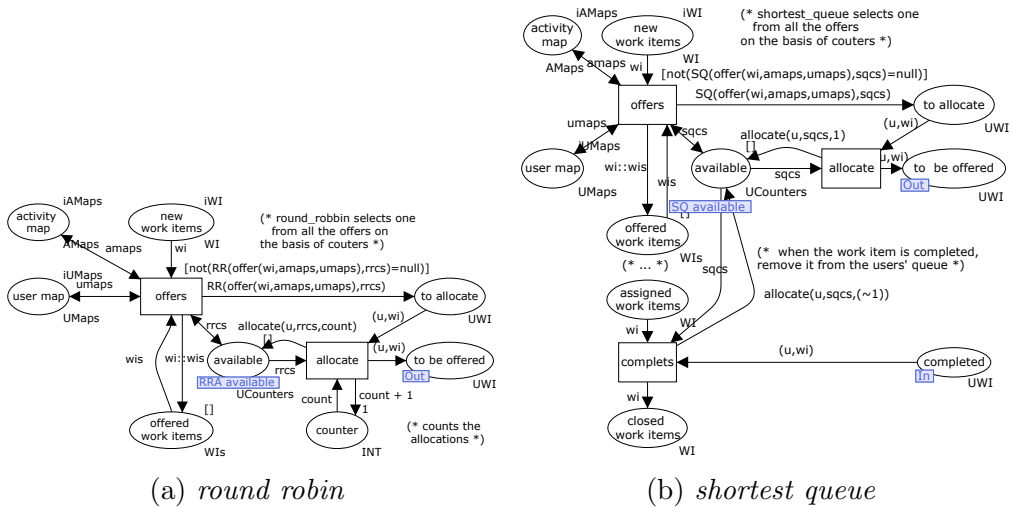


Figure 3.13: Two push patterns - work distribution module

fication in *chained execution*. This parameter is used for a possible auto-start. These two models show that transition *complete special*, besides the usual connection to places *completed* and *request*, has connections to places *active work items*, *select* and *special mode*. If the user is in the *special mode*, this transition retrieves work items from place *active work items*, and produces items in places *request* and *select*. The inscriptions on arcs leading to places *request* and *select* first check if the user is working in the special (i.e., ‘auto-start’) mode. If this is the case, the next user work item is auto-started, i.e., an appropriate token is produced in places *request* and *select*. Function *select* is implemented to search for the next matching work item based on the parameter x , i.e., (1) a work item with the same activity in *pled execution* or (2) a work item for the same instance in *chained execution*.

Overview

The CPN models of work distribution of Staffware, FileNet and FLOWer show that there is no consensus on terminology and functionality among contemporary systems (cf. Appendix A). For example, CPN models of work distribution in Staffware and FileNet are remarkable similar. However, after using these two systems one tends to have the impression that they handle work distribution in distinctive manners. On the other hand, the CPN model of work distribution in FLOWer shows that some systems can provide much quite different features than other systems.

Various workflow management systems are evaluated based on the support of resource patterns in [208, 211, 216]. Not only that this evaluation can serve

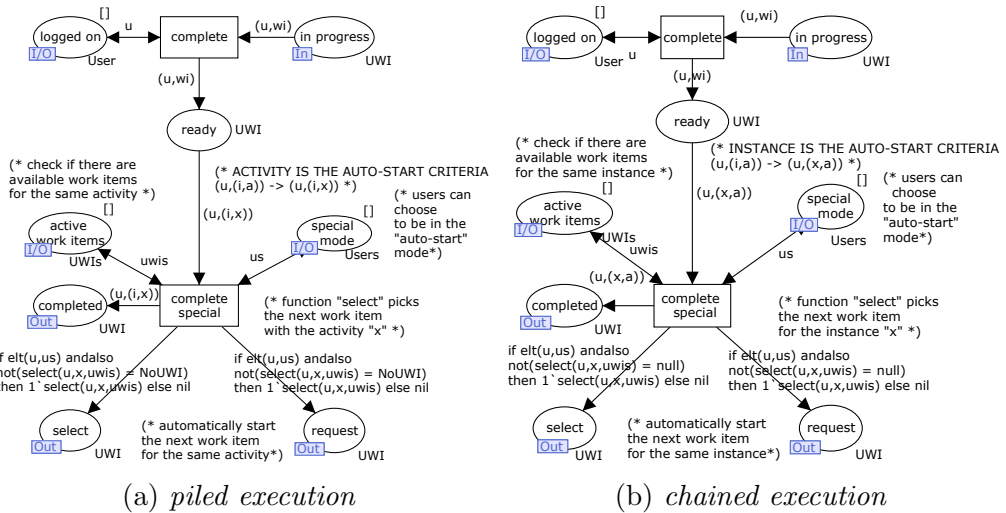


Figure 3.14: Two auto-start patterns - *stop work* module

as comparison of systems, but the evaluation results indeed show that systems support different patterns and, thus, handle the resource perspective in unique ways. In Appendix B.2 of this thesis we show the evaluation results of the CPN models of Staffware, FileNet and FLOWer, with respect to the patterns support. In addition, in Appendix B.2 we also show the evaluation results of the pattern support of our *basic model*: due to its simplicity, the *basic model* supports only few resource patterns.

CPN models of the *round robin*, *shortest queue*, *piled execution* and *chained execution* patterns show that it is remarkably simple to implement these patterns ‘on top’ of the work distribution of existing systems. Therefore, the lack of support for these patterns in Staffware and FileNet indicates the level of immaturity of contemporary systems with respect to the resource perspective, and especially when compared to the control-flow perspective.

Despite of the high impact that the resource perspective has on the way people work, this perspective does not draw much attention in research and industry. While there have been many attempts to improve the control-flow perspective (e.g., many control-flow modeling languages like Petri Nets [29, 72, 93], EPCs, BPEL [53, 54], etc. covering a wide range of application areas), there has been less research and industry interest in the resource perspective. Research efforts like [182, 208, 211, 216] are rare examples of investigations in the area of resource perspective. BPEL4People [150] and WS Human Task [47] are recent efforts aiming at enriching the resource perspective of workflow technology. Together with BPEL itself [53, 54, 178], BPEL4People is becoming a broadly recognized standard recognized by the Organization for the Advancement of Structured

Information Standards (OASIS) [7]. However, a pattern-based evaluation of these two standards [213, 217] indicates the necessity of further improvements in the area of the resource perspective.

3.1.3 The Data Perspective

The data perspective of a process model defines which data elements are available in the process and how workflow participants can access data elements while executing activities.

CPN Model(s)

Process models contain data elements of certain types (e.g., string, numeric, date, etc.). Consider, for example, a process model from the medical domain containing several data elements: *patient* name, *doctor* name and *description* all of type *string*, and *appointment* of type *date*. Once data elements have been defined on a process level, their usage can be defined for each activity, i.e., whether a data element is available and how it can be accessed and modified in an activity (cf. Figure 3.2). However, this is only a simplified, basic, view at the data perspective in workflow management systems. Real systems use powerful and very complex mechanisms that handle data elements and their values on the process and activity level. The complexity of these mechanisms yields complex CPN models of the data perspective. For example, the CPN model of the *newYAWL* workflow language ¹, which aims at fully supporting the data perspective (and the other workflow perspectives), contains 55 modules, over 480 places, 138 transitions and over 1500 lines of ML code [208, 210, 212]. Therefore, we do not present the CPN models of the data perspective in this thesis. Instead, we refer the interested reader to the *newYAWL* CPN model presented in [208, 210, 212].

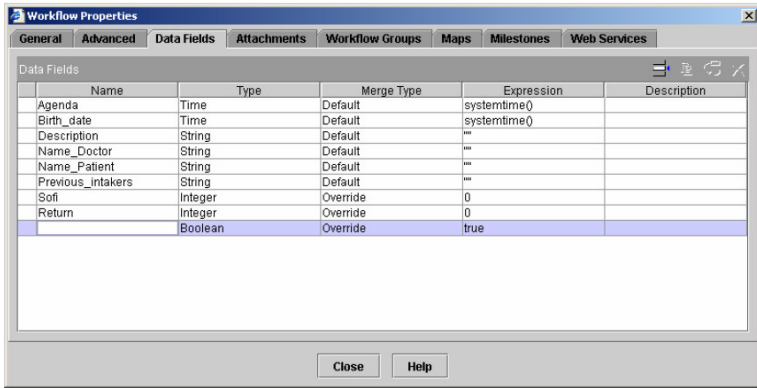
Workflow Management Systems

Staffware, FileNet and FLOWer each support the data perspective in a unique way. Generally, data elements are first defined on the process level and then for each activity it is defined which data elements are available and how users can access them. Figure 3.15 shows how data elements can be defined on the process level in Staffware, FileNet and FLOWer. Due to the complexity of the data perspective, we do not present this perspective in detail for the three workflow management systems in this thesis.

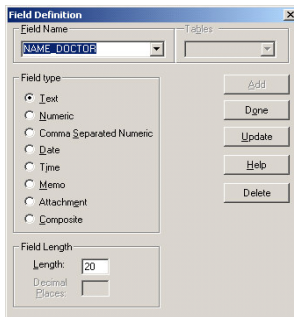
Patterns

Just like the control-flow and the resource perspective, various workflow management systems tend to handle the data perspective differently. In order to be able

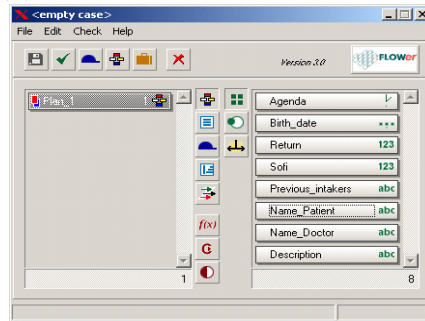
¹The *newYAWL* CPN model can be downloaded from [6].



(a) FileNet



(b) Staffware



(c) FLOWER

Figure 3.15: Defining data elements in a process model

to compare different systems with respect to the data perspective, a series of 40 workflow data patterns are identified in [214, 215]. Data patterns aim to capture the various ways in which data is represented and utilized in workflow management systems and are classified into four groups. First, *data visibility* patterns “relate to the definition and scope of data elements and the manner in which they can be utilized by various components of a workflow process” [214, 215]. Second, *data interaction* patterns “focus on the manner in which data is communicated between process components and describe the various mechanisms by which data elements can be passed across the interface of a process component” [214, 215]. Third, *data transfer* patterns focus on the manner in which the actual transfer of data elements occurs between workflow components. Finally, *data-based routing* patterns “capture the various ways in which data elements can interact with other perspectives and influence the overall operation of the process” [214, 215].

Due to the complexity of the data perspective and data patterns, CPN models representing these patterns are very large and complex. The CPN model of the

newYAWL language supports most of the data patterns [208, 210, 212]. Due to the complexity and size of this model, we do not present it in this thesis and refer the interested reader to [6, 208, 210, 212].

The evaluation of the support of the 40 data patterns in various workflow management systems [214, 215] shows that systems tend to support different patterns. Roughly half of the patterns is supported in each of the evaluated systems [214, 215]. This indicates a serious lack of uniformity in the way how the data perspective is handled amongst the systems. In Appendix B.3 of this thesis we present the data pattern support evaluation results for Staffware and FLOWer based on [208, 214, 215]. These results show that Staffware fully supports 13 and partially supports 12 patterns, while FLOWer fully supports 20 and partially 12 patterns. Unfortunately, this evaluation did not include the third system we use in this thesis, i.e., FileNet.

Overview

Although, at the first sight, the data perspective seems to be very simple, workflow management systems use very complex mechanisms to support this perspective. This yields complex data patterns [214, 215] and complex corresponding CPN models [6, 208, 210, 212].

Just like it is the case with the control-flow and resource perspectives, various workflow management systems tend to handle the data perspective in different ways. Results of the evaluation of the data patterns support shows that systems tend to support different patterns [214, 215].

Similarly like the resource perspective, the data perspective has also not drawn as much research attention as the control-flow perspective in the workflow area. Work presented in [208, 210, 212, 214, 215] is the first attempt to initiate more investigation of the data perspective in research and industry.

3.1.4 Summary

Workflow technology lacks a unique taxonomy and standards. Workflow management systems tend to handle the control-flow, resource and data perspectives in system-specific manners. The diversity of workflow patterns support [211, 213–216] in various systems indicates the diversity of functionalities offered by workflow management systems.

The control-flow perspective is the dominant workflow perspective in workflow technology research and industry. Despite of the high influence of the resource and data perspectives on the way people work, these two perspectives did not draw much attention in research and industry. While there have been many attempts to improve the control-flow perspective, there has been little interest in the other two perspectives. However, the data and resource perspectives have started to draw more attention recently. Papers like [182, 208, 211, 214–216] are

first efforts in the direction of deeper investigations of the data and resource perspectives of workflow technology.

Contemporary workflow management systems are of a procedural nature, i.e., models are detailed specifications of exactly how processes can and will be executed. Although this approach is suitable for highly-structured processes with high repetition rates, it is not appropriate for processes with high variation rate (e.g., processes that offer many execution alternatives to users). Already single control-flow patterns that offer advanced execution options tend to be very complex [213]. Thus, applying procedural models to processes that must offer many execution options tends to result in very complex process models.

3.2 Taxonomy of Flexibility

There is a fundamental gap between the workflow management systems and modern organizational science, as already indicated in Chapter 1. While modern organizational theories advocate more localized decision making, workflow management systems tend to impose old-fashioned centralized decision making due to their imperative procedural nature. In order to align themselves with the contemporary democratic style of work, workflow management systems must become more flexible by allowing users to make more decisions about how to work. Flexibility is an important research topic in the field of workflow management (cf. Chapter 2). In 1999, Heintz et al. [125] presented a classification scheme of flexibility in the context of workflow management systems (cf. Section 2.1.1). In 2007, by Schonenberg et al. re-visited the taxonomy of flexibility by looking at contemporary workflow management systems [226–228] (cf. Section 2.1.2). In this thesis, we use the four types of flexibility identified in [226–228]: *flexibility by design*, *flexibility by underspecification*, *flexibility by change* and *flexibility by deviation*. In this section we will present these four types of flexibility: flexibility by design in Section 3.2.1, flexibility by underspecification in Section 3.2.2, flexibility by change in Section 3.2.3, and flexibility by deviation in Section 3.2.4. Each type of flexibility is described with respect to the three workflow perspectives i.e., the control-flow, resource and data perspectives, using simple, system and language-independent illustrative examples.

3.2.1 Flexibility by Design

If a process model can be developed in a way that it allows for many alternative executions (execution traces or paths), then we speak about *flexibility by design*. In Figure 3.16 an execution alternative is represented with a directed arc from the ‘start’ until the ‘end’ point. In other words, a degree of flexibility by design is determined by the variety of alternatives available at run-time while executing instances of process models. This type of flexibility is identified in both taxonomies: in [125] it is called *flexibility by selection with advanced modeling* and

in [226–228] it is called *flexibility by design*.

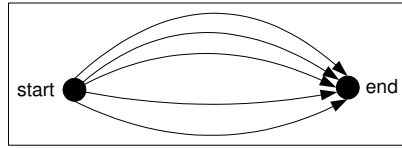


Figure 3.16: Flexibility by design [125, 226–228]

The control-flow perspective. The control-flow perspective determines which activities will be available for execution at run-time and in which order these activities can be executed (cf. Section 3.1.1). Consider, for example, the two process models presented in Figure 3.17. These models consist of activities A , B , C and D and use the *sequence*, *parallel split* and *synchronization* control-flow patterns (cf. Figure 3.7). The control-flow of the process model in Figure 3.17(a) is defined as a *sequence* of activities A , B , C and D . Thus, users have only one option while executing this model, i.e., to execute these activities only in the following order: $[A, B, C, D]$. The control-flow perspective of the process model shown in Figure 3.17(b) starts with activity A , after which a *parallel split* to activities B and C follows. Finally, there is a *synchronization* of activities B and C before activity D . Because activities B and C can be executed in any order, users have two alternatives while executing this model: (1) they can execute these activities in order $[A, B, C, D]$ or (2) they can execute these activities in order $[A, C, B, D]$. Due to the fact that the process model in Figure 3.17(a) has only one execution alternative (i.e., $[A, B, C, D]$) and the process model in Figure 3.17(b) has two execution alternatives (i.e., $[A, B, C, D]$ or $[A, C, B, D]$), we say that the model in Figure 3.17(b) has a *higher degree of flexibility by design* than the model in Figure 3.17(a).

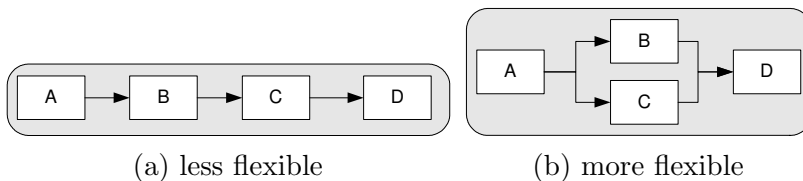


Figure 3.17: Flexibility by design and the control-flow perspective

The resource perspective. As discussed in Section 3.1.2, the resource perspective determines which users are authorized to execute activities and how these resources are allocated to execute the activities. Consider, for example,

the simple illustrative example presented in Figure 3.18. This figure shows the resource perspective of two hypothetical process models. Each of the models consists of activities *A*, *B* and *C*, and considers *two users*. In the model in Figure 3.18(a) the first user is authorized to execute activities *A* and *B* and the second user is authorized to execute activity *C*. Thus, this model offers only one execution alternative, i.e., the first user will execute activities *A* and *B* and the second user will execute activity *C*. In the model in Figure 3.18(b) both users are authorized to execute each of the three activities. Therefore, this model offers more (i.e., eight) execution alternatives and has a higher degree of flexibility by design with respect to the resource perspective.

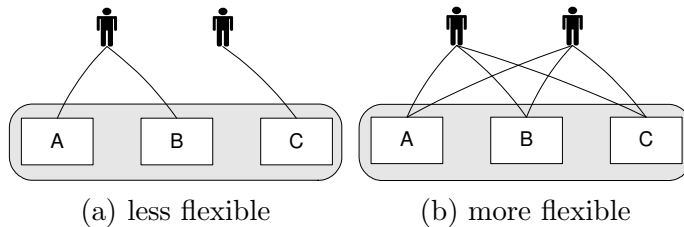


Figure 3.18: Flexibility by design and the resource perspective

The data perspective. The data perspective of a process model defines the availability of data during the execution (cf. Section 3.1.3). Flexibility by design is also influenced by the data perspective. Figure 3.19 shows the data perspective of two process models. Each of the models consists of activities *A*, *B* and *C* and uses two data elements. In the model in Figure 3.19(a) the first data element is output for activity *A* and input for activity *B*, while the second data element is output for activity *B* and input for activity *C*. The model in Figure 3.19(b) has a higher level of flexibility by design because both data elements are input and output elements for all three activities, i.e., all data elements can be accessed and edited in all three activities. Consider, for example, the situation where an incorrect value of the first data element is provided while executing activity *A*. On the one hand, in the model in Figure 3.19(a) this mistake cannot be corrected while executing activities *B* or *C* because the first data element *is not* an output data element for these two activities. On the other hand, in the model in Figure 3.19(b) this mistake can easily be corrected, because the first data element *is* an output data element for activities *B* and *C*. The same holds for input data elements: while the value of the first data element is presented to users *only* while executing activity *B* in the model in Figure 3.19(a), in the model in Figure 3.19(b) the value of this data element is presented to users while executing all three activities.

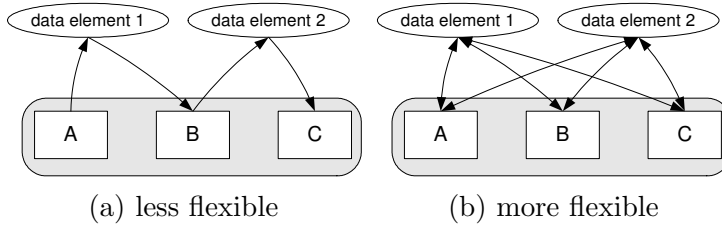


Figure 3.19: Flexibility by design and the data perspective

3.2.2 Flexibility by Underspecification

The possibility to only partially specify a process model, where certain parts of the model are left undefined as ‘black boxes’ and will be defined later during execution is called *flexibility by selection with late modeling* [125] or *flexibility by underspecification* [226, 227, 228]. In Figure 3.20 under-specified parts of a process model are presented as partial dashed lines of execution alternatives. An example of a system that allows for this type of flexibility is the worklet extension of the YAWL system [23, 44]. Some of the activities in YAWL models can be considered as unspecified parts of the model and during execution they are assigned to the Worklet Service [44] that chooses the exact specification (of a sub-process) that will be executed (cf. Section 2.2).

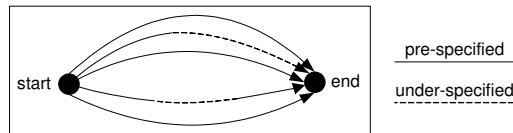


Figure 3.20: Flexibility by underspecification [125, 226–228]

The control-flow perspective. An ‘under-specified’ control-flow perspective of a process model is shown in Figure 3.21(a). This process starts with activity A, followed by activity B and completes with an *unspecified* block. During each execution of this model (i.e., for each instance), it is necessary to define explicitly the unspecified block. For example, as Figure 3.21(b) shows, a possible execution scenario could be that, after activities A and B were executed in an instance (indicated by the special ‘check box’ symbols), activity D was selected for the unspecified block. In another instance, some other activity (e.g., some activity G) may be executed for the unspecified block. Not only single activities can replace the unspecified blocks of the control-flow - an entire sub-process can be selected for an unspecified block. Note that, each time the process is executed i.e., for

each process instance), it is possible to select a different activity or sub-processes for the same unspecified block in the control-flow.

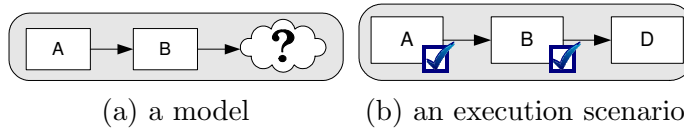


Figure 3.21: Flexibility by underspecification and the control-flow perspective

The resource perspective. Figure 3.22 shows a simple example of underspecification in the resource perspective. Two users are defined in the resource perspective of a model presented in Figure 3.22(a). The first user is authorized to execute activities *A* and *B*, while the authorization for activity *C* is intentionally left unspecified. This underspecification leaves the opportunity to specify the authorized user(s) for activity *C* later, during the execution of the model. Each time this model is executed, another user(s) can be selected as authorized to execute activity *C*. One of the possibilities is to authorize both users to execute activity *C* after executing activities *A* and *B*, as shown in Figure 3.22(b).

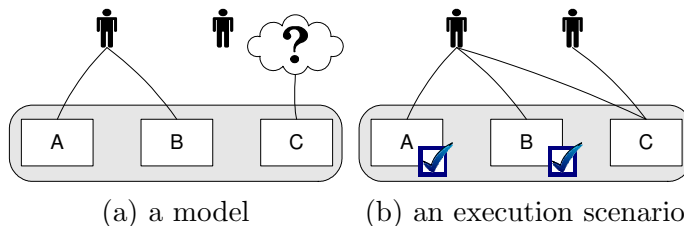


Figure 3.22: Flexibility by underspecification and the resource perspective

The data perspective. Figure 3.23(a) shows underspecification of the data perspective in a process model that uses two data elements. The first data element is output for activity *A* and input for activity *B*, while the second data element output for activity *B*. The data access for activity *C* is intentionally left unspecified, i.e., a reference to the right data element can be specified each time the process is executed. For example, it might be the case that, after activities *A* and *B* were executed, it is specified that both data elements are input but only the second data element is output for activity *C*, as shown in Figure 3.23(b).

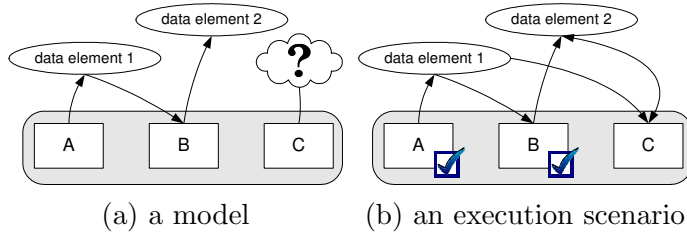


Figure 3.23: Flexibility by underspecification and the data perspective

3.2.3 Flexibility by Change

While an instance of a process model is being executed it may become necessary to change the set of execution alternatives of the model, e.g., by adding alternative(s) that were not initially foreseen when the model was developed. *Flexibility by change* [226–228] or *flexibility by instance adaptation* [125] allows for adding execution alternatives to the model while executing the model, as Figure 3.24 shows. Ad-hoc (or run-time) change of models is an important property of so-called adaptive workflow systems like, e.g., ADEPT [189, 191–193, 202]. Systems like ADEPT are equipped with powerful mechanisms that enable ad-hoc change of one or more instances by allowing adding, deleting and moving activities in instances that are already being executed (cf. Section 2.1.5). Moreover, it is possible to apply the ad-hoc change (a) to one instance or (b) to all instances of the referring model, i.e., the so-called *migration*.

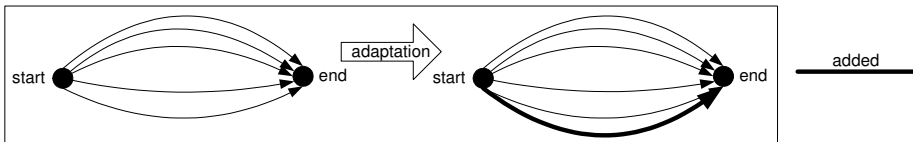


Figure 3.24: Flexibility by change [125, 226–228]

The control-flow perspective. An example of ad-hoc change in the control-flow perspective is shown in Figure 3.25. As shown in Figure 3.25(a), the control-flow perspective of the model is defined as a sequence of activities *A*, *B* and *C*. Thus, according to the control-flow specification, this model will be executed by first executing activity *A*, then activity *B* and finally activity *C*. However, it is possible to add execution alternatives by ad-hoc change of the control-flow perspective. Figure 3.25(b) shows an instance of this model where activities *A* and *B* are executed and then, instead of executing activity *C*, activity *D* is inserted before activity *C* in the control-flow specification. After this ad-hoc

change, the instance continues with an execution of activity *D* followed by an execution of activity *C*.

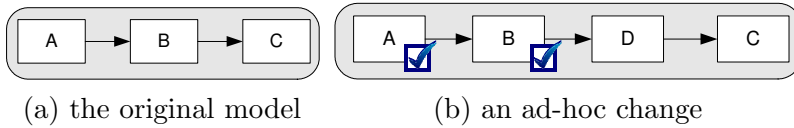


Figure 3.25: Flexibility by change and the control-flow perspective

The resource perspective. Figure 3.26 shows an example of ad-hoc change in the resource perspective. In the process model shown in Figure 3.26(a) the first user is authorized to execute activities *A* and *B* and the second user is authorized to execute activity *C*. However, it is possible that, in an execution scenario where activities *A* and *B* were already executed, the authorization for activity *C* is removed from the second user and assigned to the first user, as shown in Figure 3.26(b). After this ad-hoc change, the first user will execute activity *C*, instead of the originally authorized second user.

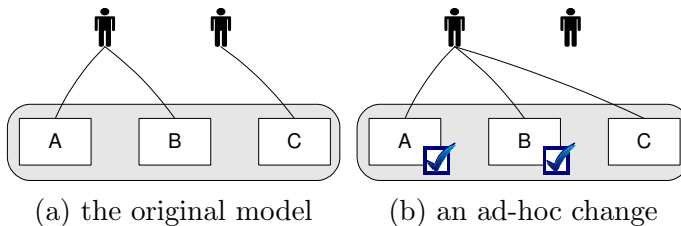


Figure 3.26: Flexibility by change and the resource perspective

The data perspective. Flexibility by change can also be applied to the data perspective, as shown in Figure 3.27. In the process model shown in Figure 3.27(a) the first data element is output for activity *A* and input for activity *B*, while the second data element is output for activity *B* and input for activity *C*. However, flexibility by change allows to change the data perspective in instances of this model in an ad-hoc manner. For example it is possible that, after activities *A* and *B* are executed in an instance, this specification is changed so that the first data element is added as an input and output element for activity *B*, as shown in Figure 3.27(b).

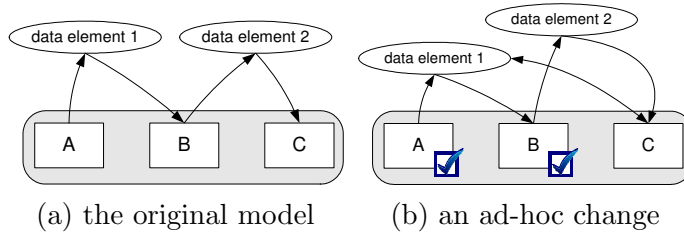


Figure 3.27: Flexibility by change and the data perspective

3.2.4 Flexibility by Deviation

Flexibility by deviation is the ability of a process instance to deviate from the execution alternatives prescribed in the instance's process model *without* changing the model. A deviation from the specified execution alternatives is illustrated with a thick line in Figure 3.28. FLOWer [180] is an example of a system that allows for deviation from the process model by allowing users to skip an activity that should be executed and redo or undo an activity that was already executed before (cf. Section 2.2).

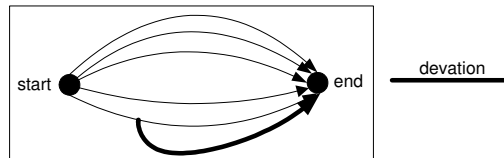


Figure 3.28: Flexibility by deviation [226–228]

The control-flow perspective. Figure 3.29 shows an example of deviation from the control-flow perspective specified in a process model. The control-flow perspective of the model presented in Figure 3.29(a) is specified as a sequence of activities *A*, *B* and *C*. However, if deviation is applied during the execution of this model, it becomes possible to execute the model in ways other than specified (i.e., other than executing *A*, *B* and *C* in a sequence). For example, it is possible to, after executing activity *A*, skip activity *B* and execute directly activity *C*, as shown in Figure 3.29(b) by a thick line. Note that the corresponding model in Figure 3.29(b) is not changed. Instead, the thick line represents the deviation from the model, i.e., it represents the actual execution of the instance.

The resource perspective. An application of flexibility by deviation on the resource perspective is presented in Figure 3.30. The resource perspective of

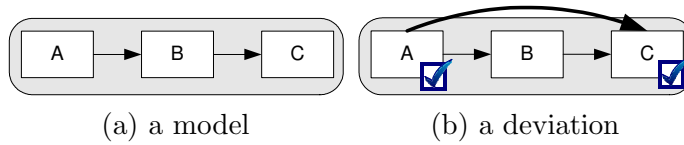


Figure 3.29: Flexibility by deviation and the control-flow perspective

a process model is shown in Figure 3.30(a): one user is authorized to execute activities *A* and *B* and the other user is authorized to execute activity *C*. This specification remains the same in the execution scenario (i.e., instance) shown in Figure 3.30(b). However, this instance deviates from the original specification because the first user executes activity *C*, although he/she is not authorized to execute this activity. Thick lines represent the actual execution of the instance shown in Figure 3.30(b), i.e., the first user executed all three activities in this instance.

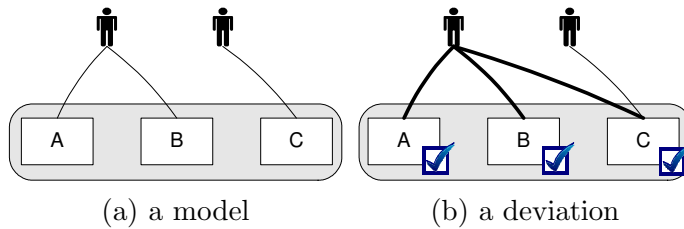


Figure 3.30: Flexibility by deviation and the resource perspective

The data perspective. Figure 3.31 shows how flexibility by deviation can be achieved in the data perspective. The original specification of the data perspective, where the first data element is output for activity *A* and input for activity *B*, is shown in Figure 3.31(a). Further on, the second data element is output for activity *B* and input for activity *C*. An example of a deviating execution (i.e., instance) is shown in Figure 3.31(b). Here the actual execution is shown by thick lines, i.e., the value of first data element is not provided while executing activity *A*, and, despite the original specification, it is left ‘unspecified’.

3.2.5 Summary

As already discussed in Section 3.1, the control-flow perspective remains in the focus of research and industry, while the resource and data perspectives tend to get less attention. This is also the case with respect to flexibility of workflow

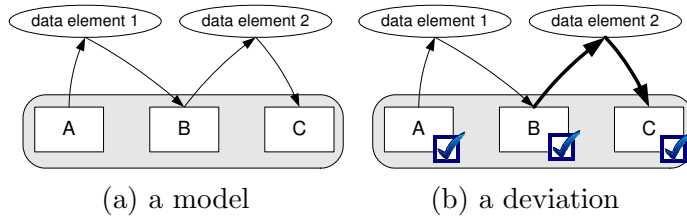


Figure 3.31: Flexibility by deviation and the data perspective

management systems. Note that the examples we presented in this section for the resource and data perspectives are hypothetical illustrative examples and, so far, flexible workflow management systems do not focus on these two perspectives. Instead, the control-flow perspective is dominant in the systems. For example, the YAWL system with worklets [23,44], ADEPT [189] and FLOWer [180] directly support flexibility by underspecification, change and deviation in the control-flow perspective, respectively (cf. Section 2.2).

In sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4 we deliberately use very simple examples of the control-flow, resource and data perspectives of process models to illustrate how different types of flexibility can be applied for each of the three workflow perspectives. In ‘real-life’ process models and workflow management systems these three perspectives require much more advanced forms of flexibility, resulting in much more complicated models. These models often use many workflow patterns [208,211,213–216] in order to offer multiple execution alternatives, which often increases the complexity of models, as discussed in Section 3.1.

The control-flow perspective of current workflow management systems is of *procedural* nature (cf. Section 3.1.4). Thus, *all execution alternatives must be explicitly specified* in the process model. This has several consequences:

- Procedural models with multiple execution alternatives tend to be *large and complex*, which makes it hard to understand and maintain these models. As discussed before, multiple execution alternatives with respect to the control-flow perspective require usage of many complex control-flow patterns [213].
- In a procedural approach *all execution alternatives must be anticipated in advance* [77, 109, 125, 143, 153, 166, 188, 233]. With respect to the flexibility by design, this means that all execution alternatives must be anticipated already in the development phase. With respect to the flexibility by underspecification, this means that it must be anticipated in the development phase exactly when the unspecified block should be executed. Flexibility by change and deviation might lead to frequent ad-hoc changes and deviations, i.e., each time a new alternative is identified a new ad-hoc change or

deviation must be applied.

- Explicitly specifying the procedure in the model can result in *over-specifying the process* [181]. To illustrate this, consider the case when it is necessary to specify the requirement that in one instance of a process model activities A and B cannot be executed both, i.e., it is possible to execute A once or more times as long as B is not executed and vice versa. It is also possible that none A nor B is executed at all. In a procedural approach one tends to *over-specify* this as shown in Figure 3.32. A decision activity X is introduced as an activity that needs to be executed at a particular time and requires conditions $c1$ and $c2$ to make this decision. Note that, although the requirement A and B exclude one another is very simple, all kinds of questions are introduced: ‘When should X be executed?’, ‘How many times should X be executed?’, ‘Who executes X ?’, ‘What are $c1$ and $c2$?’, etc.

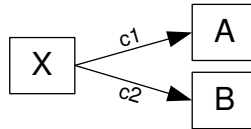


Figure 3.32: Over-specification in procedural models [181]

In this chapter we used few languages/systems (i.e., CPNs, Staffware, FLOWer, and FileNet) to describe how contemporary approaches specify the control-flow perspective of business processes. However, a procedural approach is dominant in the contemporary workflow technology, despite the variety of existing process modeling languages. On the one hand, formal languages like, for example, Petri nets [87, 177, 199], process algebras [57, 61, 127, 131, 173] and state charts [267] use various formalisms to explicitly specify the procedure of a processes. On the other hand, procedural approach is also adopted by languages that are frequently used in practice and by many available tools, like, for example, Event-Driven Process Chains (EPCs) [146, 147, 225], Business Process Execution Language for Web Services (BPEL) [54], and Business Process Modeling Notation (BPMN) [179].

3.3 A New Approach for Full Flexibility

As discussed in Section 3.2.5, the fact that the procedural approach requires all execution alternatives to be explicitly specified in the model causes some problems with respect to flexibility of workflow management systems. Descriptive or *declarative* languages are considered to be more suitable for achieving a higher degree of flexibility because they do not require explicit specification of execution

alternatives [48, 49, 55, 56, 80, 92, 115, 256, 269]. Instead, a declarative approach allows for the *implicit* specification of execution alternatives. In this thesis we propose a declarative approach for achieving a higher degree of flexibility in workflow technology.

The proposed approach is based on using activities and *constraints* for declarative specification of the control-flow perspective of process models. Constraints are rules that should be followed during the execution. Note that constraints of a model implicitly specify the possible execution alternatives: everything that does not violate constraints is allowed. Figure 3.33 shows an example of a constraint. This constraint involves two activities (i.e., *A* and *B*) and it specifies that these two activities cannot both be executed in the same process instance. By using this constraint in a process model, we implicitly specify its execution alternatives as all alternatives where (1) *A* is executed at least once and *B* is never executed, (2) *B* is executed at least once and *A* is never executed and (3) neither *A* nor *B* are executed. Execution of other activities does not influence this constraint, e.g., any other activity can be executed at any point of time, as long as activities *A* and *B* are not executed both. Note that, although the constraint in Figure 3.33 represents a very simple and useful rule, there is no control-flow pattern that represents it [213]. Instead, implementing such a rule in the procedural manner requires over-specification, as shown in Figure 3.32. Moreover, by over-specifying this rule in the procedural approach many of the intended execution alternatives are discarded, i.e., either activity *A* or activity *B* must be executed exactly once after activity *X*.

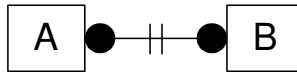


Figure 3.33: A constraint: activities *A* and *B* should not be executed both

The difference between the procedural and our constraint-based declarative approach to process modeling is shown in Figure 3.34. Procedural models take an ‘inside-to-outside’ approach: all execution alternatives are explicitly specified in the model and new alternatives must be explicitly added to the model. Declarative models take an ‘outside-to-inside’ approach: constraints implicitly specify execution alternatives as all alternatives that satisfy the constraints and adding new constraints usually means discarding some execution alternatives.

Our approach focuses on the dominant workflow perspective, i.e., the control-flow perspective, and it can support all types of flexibility, as Figure 3.35 shows. In this approach we advocate using constraints for the specification of the control-flow perspective of process models. Possible execution alternatives of a model are implicitly derived from the constraints: any alternative that satisfies all constraints is possible. Thus, even simple constraint-based process models can offer many execution alternatives, i.e., our approach is appropriate for achieving *flexi-*

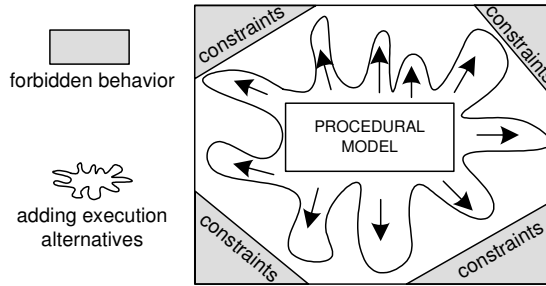


Figure 3.34: Declarative vs. procedural approach

bility by design. Moreover, other types of flexibility can also be supported. With help of the YAWL system and its worklets [44] it is possible to create arbitrary decompositions of procedural and declarative models and achieve *flexibility by underspecification*. *Flexibility by change* can be achieved by ad-hoc change, which can be easily applied to the constraint-based approach. The so-called *optional constraints* allow for *flexibility by deviation*.

flexibility by design	✓		
flexibility by underspecification	✓		
flexibility by change	✓		
flexibility by deviation	✓		
	control-flow perspective	resource perspective	data perspective

Figure 3.35: A new approach for all types of flexibility

Note that it *is possible* to develop procedural models that allow for flexibility by design. Consider, for example, a process model consisting of activities A , B and C , where any execution alternative is possible. Figure 3.36 shows a CPN representing this model. Between the initiation and termination of the process (represented by transitions *start* and *end*, respectively) it is possible that (1) each of activities A , B and C is executed zero or more times, (2) activities A , B and C are executed concurrently and (3) activities A , B and C are executed in any order. Thus, this model allows for infinitely many execution alternatives and offer a high degree of flexibility by design (cf. Section 3.2.1). However, adding some simple rules that should be followed during execution is often too costly

in the procedural approach because models become too complex and large, a lot of time and human efforts are needed, etc. Adding a simple rule, e.g., like the one presented in Figure 3.33, would result in a very complex model. Adding several similar rules might easily result in a model that is too large and complex to understand and maintain.

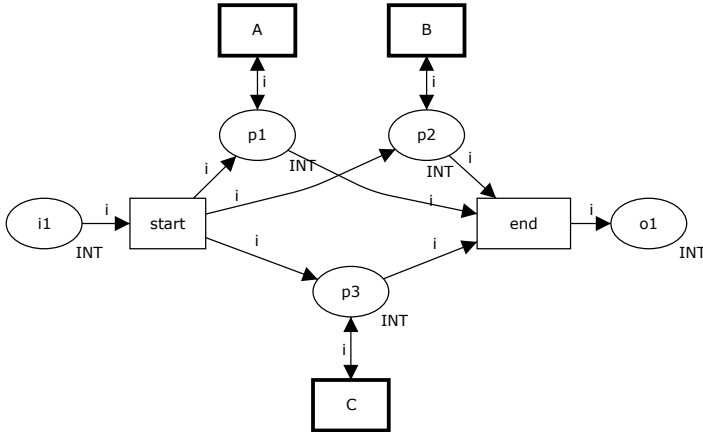


Figure 3.36: Any execution alternative of activities *A*, *B*, and *C* is possible between activities *start* and *end*

The remainder of this thesis is organized as follows. In Chapter 4 we provide a formal foundation for a constraint-based language on an abstract level. In Chapter 5 we present a language that can be used to specify constraints in process models and in Chapter 6 we present a prototype of a workflow management system implemented based on concepts presented in chapters 4 and 5. In Chapter 7 we will show how popular techniques for process mining [8] can be applied to the constrain-based approach both for analysis of past executions and generating run-time recommendations for users. Finally, in Chapter 8 we discuss and conclude the thesis.

Chapter 4

Constraint-Based Approach

A truly flexible approach to workflow management systems must provide for several aspects of workflow flexibility [125, 226–228], as discussed in Chapter 2. In Chapter 3 we showed that contemporary commercial tools use imperative process models that explicitly specify how to execute the process. This requires process model developers to predict all possible execution scenarios in advance and explicitly include them in the model. However, it is often not the case that all scenarios can be foreseen in advance [125]. Therefore, the procedural nature of process models makes it very difficult for contemporary workflow management systems to provide for a high degree of *flexibility by design* [125, 226–228].

In this chapter we present a formal foundation for a constraint-based approach for business processes. Instead of explicitly specifying the control flow, constraint-based process models focus on constraints as rules that have to be followed during the process execution. Possible executions of constraint models are specified implicitly as all executions that satisfy the model constraints, which makes it *not necessary* to explicitly predict all possible executions in advance. Due to the declarative nature of constraint models that offer a variety of possibilities for execution, the constraint-based approach is *flexible by definition* [226–228]. Moreover, all other types of flexibility that were discussed in [226–228] can also be achieved using this approach. First, while executing instances, people can violate one type of constraints and achieve *flexibility by deviation* [226–228]. Second, it is possible to change models of already running instances, which allows for the *flexibility by change* [226–228]. Although the *flexibility by underspecification* [226–228] is not explicitly built into the approach presented in this chapter, in Section 6.11 we will show how this type of flexibility can be achieved with the help of the YAWL system [23, 32, 210, 212] and its worklets [41, 44, 45].

We start this chapter by describing the notion of constraints in Section 4.1 and constraint models in Section 4.2. An illustrative example is presented in Section 4.3. In Section 4.4 we describe how instances of constraint models are executed. Ad-hoc change of already running instances is described in Section 4.5.

Section 4.6 presents how verification of constraint models can detect serious errors and help develop correct constraint models. A summarized overview of the chapter is given in Section 4.7.

4.1 Activities, Events, Traces and Constraints

Constraint-based models consist of *activities* and *constraints*. An activity is a piece of work that is executed as a whole by a resource (e.g., one person, a computer, etc.). A constraint specifies a certain rule that should hold in any execution of the model. Consider, for example, a model without constraints that contains only activities *perform surgery* and *prescribe rehabilitation*. Medical staff members that execute this model have the ultimate freedom to execute the two activities an arbitrary number of times and in any order. However, if the constraint “if *perform surgery*, then eventually *prescribe rehabilitation*” would be added to the model, this would limit (i.e., constrain) the possibilities that resources have: if activity *perform surgery* is executed, this constraint will demand to afterwards also execute activity *prescribe rehabilitation*.

The time needed for the resource to execute one activity can vary depending on the activity’s complexity, the resource capabilities, etc. For example, one activity might take a few minutes to execute (e.g., *prescribe rehabilitation*) and another might take several hours (e.g., *perform surgery*). One can imagine that the execution of activities can overlap, that some executions might fail and others might complete successfully. Resources execute models by triggering *events involving activities*, i.e., transferring activities through various states in their life cycles [29, 91, 93, 136, 160, 175]. Figure 4.1 shows an example of a simple activity life cycle. In this life cycle, an activity can be in one of the four states: *initial*, *execution*, *execution successful*, and *execution failed*. At the beginning of its life cycle, the activity is in the *initial* state. As shown in Figure 4.1, each event triggers a state change. A resource starts to actively work on an activity by triggering the event *started*. After the *started* event, the activity can be *completed* (i.e., successfully executed) or *cancelled* (i.e., execution has failed).

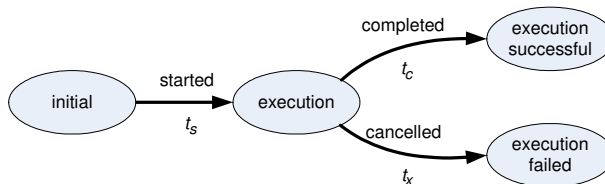


Figure 4.1: Three event types - *started* (t_s), *completed* (t_c) and *cancelled* (t_x)

In the remainder, we will use the following terms (concepts):

- \mathcal{A} is the set of all *activities*, i.e., a universe of activity identifiers,
- \mathcal{T} is the set of all *event types*, and
- $\mathcal{E} = \mathcal{A} \times \mathcal{T}$ is the set of all *events*.

Although the set of event types can by definition contain arbitrary types, for the purpose of simplicity we adopt the three event types from the life cycle in Figure 4.1. In other words, in the remainder of this thesis we will assume that \mathcal{T} contains three event types $\mathcal{T} = \{t_s, t_c, t_x\}$, such that $t_s = \textit{started}$, $t_c = \textit{completed}$ and $t_x = \textit{cancelled}$. Note that, however, the set of event types \mathcal{T} is customizable, i.e., the proposed constraint-based approach can be applied to any set of event types \mathcal{T} .

One execution of a model is defined as one *trace*, i.e., a sequence of events that represents the chronological order of events that occurred and were recorded during the execution (cf. Definition 4.1.1).

Definition 4.1.1. (Trace)

Trace $\sigma \in \mathcal{E}^*$ is a finite sequence of events, where \mathcal{E}^* is the set of all traces composed of zero or more elements of \mathcal{E} . We use $\sigma = \langle e_1, e_2, \dots, e_n \rangle$ to denote a trace.

- $|\sigma| = n$ represents the length of the trace,
- Empty trace is denoted by $\langle \rangle$, i.e., $|\langle \rangle| = 0$,
- $\sigma[i]$ denotes the i -th element of the trace, i.e., $\sigma[i] = e_i$,
- $e \in \sigma$ denotes $\exists_{1 \leq i < |\sigma|} \sigma[i] = e$,
- $\sigma^{i \rightarrow}$ denotes the suffix of σ starting at $\sigma[i]$, i.e., $\sigma^{i \rightarrow} = \langle \sigma[i], \sigma[i + 1], \dots, \sigma[n] \rangle$,
- We use $+$ to concatenate traces into a new trace, i.e., $\langle e_1, e_2, \dots, e_n \rangle + \langle f_1, f_2, \dots, f_m \rangle = \langle e_1, e_2, \dots, e_n, f_1, f_2, \dots, f_m \rangle$,
- We use $=$ to denote equal traces, i.e., if $\sigma = \gamma$ then $|\sigma| = |\gamma|$ and $\forall_{1 \leq i \leq |\sigma|} \sigma[i] = \gamma[i]$. We use \neq to denote non-equal traces, i.e., $\sigma \neq \gamma$ denotes that $\sigma = \gamma$ does *not* hold.

□

Example 4.1.2 illustrates the concepts of activities, events and traces in a medical department.

Example 4.1.2. (Activities, events and traces)

Consider an example of a medical department where staff members can execute activities $a_e, a_s, a_r, a_m \in \mathcal{A}$ where $a_e = \textit{examine patient}$, $a_s = \textit{perform surgery}$, $a_r = \textit{prescribe rehabilitation}$, $a_m = \textit{prescribe medication}$, $a_x = \textit{perform X ray}$. These activities are executed by triggering events that can be of the three types $t_s, t_c, t_x \in \mathcal{T}$ where $t_s = \textit{started}$, $t_c = \textit{completed}$ and $t_x = \textit{cancelled}$.

Staff of this department can trigger any of the event types on any of the activities. Therefore, possible events in this example are: $e_{es} = (a_e, t_s)$, $e_{ec} = (a_e, t_c)$, \dots , $e_{xc} = (a_x, t_c)$, where $e_{ij} = (a_i, t_j)$ and $e_{ij} \in \mathcal{E}$ for all $i \in \{e, s, r, m, x\}$ and $j \in \{s, c, x\}$.

By triggering events, staff creates a trace for each patient as a sequence of triggered events. Treatments of three patients refer to three traces $\sigma_1, \sigma_2, \sigma_3 \in \mathcal{E}^*$, such that:

- $\sigma_1 = \langle e_{es}, e_{ec}, e_{ms}, e_{mc}, e_{ss}, e_{sc} \rangle$,
- $\sigma_2 = \langle e_{es}, e_{ec}, e_{ms}, e_{ss}, e_{sc}, e_{ms}, e_{mc}, e_{rs}, e_{rc} \rangle$, and
- $\sigma_3 = \langle e_{ms}, e_{mc}, e_{es}, e_{ec}, e_{ms}, e_{mc} \rangle$.

□

We introduce the trace projection as a preliminary operation on traces. Projection $\sigma^{\downarrow E}$ of a trace $\sigma \in \mathcal{E}^*$ on a set of events $E \subseteq \mathcal{E}$ is specified in Definition 4.1.3. The projection $\sigma^{\downarrow E}$ is a set of traces such that for each projection trace $\gamma \in \sigma^{\downarrow E}$ it holds that (1) γ is of the same length as σ , (2) events from E are the same and exactly on the same positions in γ and σ , and (3) events not contained in E do not have to be the same γ and σ , but need to fill the same positions in γ and σ . In other words, the projection of a trace σ on a set of events E depends on *both occurrences and positions* of events $e \in E$ and *only positions* of events $e \notin E$ in trace σ .

Definition 4.1.3. (Trace projection $\sigma^{\downarrow E}$)

Let $\sigma \in \mathcal{E}^*$ be a trace and $E \subseteq \mathcal{E}$ be a set of events. The projection of trace σ on a set of events E is the set of traces defined as follows:

$$\sigma^{\downarrow E} = \begin{cases} \{ \langle \rangle \} & \text{if } \sigma = \langle \rangle; \\ \{ \langle e_1, e_2, \dots, e_{|\sigma|} \rangle \in \mathcal{E}^* \mid \forall_{1 \leq i \leq |\sigma|} (\sigma[i] \in E \Rightarrow e_i = \sigma[i]) \wedge \\ (\sigma[i] \notin E \Rightarrow e_i \notin E) \} & \text{otherwise.} \end{cases}$$

Note that a trace is always an element of its projection, i.e., $\sigma \in \sigma^{\downarrow E}$. □

Trace projection is used as a kind of trace equivalence regarding constraint satisfaction and is used to decide whether a trace satisfies a constraint, as will be specified later in Definition 4.1.4. Consider, for example, the three traces in Figure 4.2 where we want to measure if they satisfy rule “Before the *diploma* is issued, the student has to *enroll* and to pass one course by choice.”, i.e., “events (*enroll*, t_c) and (*diploma*, t_s) cannot appear next to each other.”.

In this case, both (1) appearances and positions of events (*enroll*, t_c) and (*diploma*, t_s) in the trace and (2) positions of other events (i.e., events (*enroll*, t_s), (*courseA*, t_s), (*courseA*, t_c), (*courseB*, t_s), (*courseB*, t_c) and (*diploma*, t_c)) in the trace are important when deciding if the trace satisfies the constraint. Consider, for example, trace $\sigma_1 = \langle (\textit{enroll}, t_s), (\textit{enroll}, t_c), (\textit{courseA}, t_s), (\textit{courseA}, t_c), (\textit{diploma}, t_s), (\textit{diploma}, t_c) \rangle$ where the student *enrolled* and passed the *courseA*

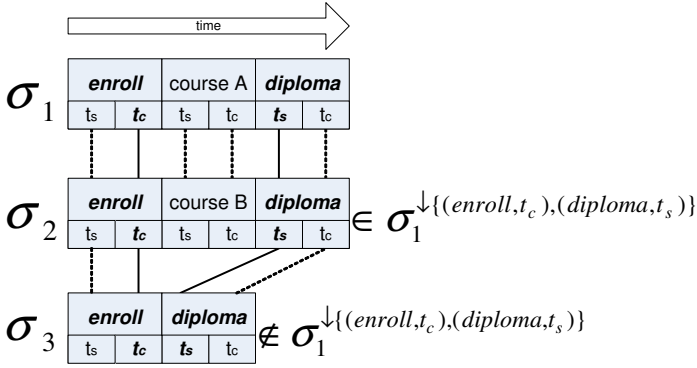


Figure 4.2: Example illustrating trace projection: $\sigma_1 \in \sigma_2^{\downarrow E}$, $\sigma_2 \in \sigma_1^{\downarrow E}$, but $\sigma_3 \notin \sigma_1^{\downarrow E}$, $\sigma_3 \notin \sigma_2^{\downarrow E}$

before the *diploma* was issued. Projection of this trace on the set of events $E = \{(enroll, t_c), (diploma, t_s)\}$ is a set $\sigma_1^{\downarrow E}$ containing all traces from \mathcal{E}^* that have the form of $\langle e_{*1}, (enroll, t_c), e_{*2}, e_{*3}, (diploma, t_s), e_{*4} \rangle$, such that events $e_{*1}, \dots, e_{*4} \in \mathcal{E}$ are not contained in E (i.e., $e_{*1}, \dots, e_{*4} \notin E$). For example, trace σ_2 is in the projection $\sigma_1^{\downarrow E}$ because events $(enroll, t_c)$ and $(diploma, t_s)$ have exactly the same positions relative to each other and events $e_* \notin \{(enroll, t_c), (diploma, t_s)\}$ in σ_1 and σ_2 . On the other hand, trace σ_3 is not in the trace projection $\sigma_1^{\downarrow E}$ because positions of events $e_* \notin \{(enroll, t_c), (diploma, t_s)\}$ do not match positions of such events in σ_1 . In other words, in $\sigma_1^{\downarrow E}$ it is not important which course the student took, as long as the diploma was not issued directly after enrolling.

A constraint is a rule that should be followed during the execution. For example, constraint “if *perform surgery is completed*, then afterwards eventually *prescribe rehabilitation is completed*” ensures that every patient who had a surgery successfully completes the rehabilitation. Another example is the constraint “cannot *start perform surgery* before *completing perform X ray*”, which makes sure that X rays of a patient are taken before surgery. As specified in Definition 4.1.4, a constraint is defined by its namespace and a function that evaluates to **true** or **false** for a given execution trace. The notion of namespace plays an important role when deciding if a trace satisfies the constraint.

Definition 4.1.4. (Constraint)

A constraint is a pair $c = (E, f)$, where:

- $E = (A \times T)$ is the namespace of c such that $A \subseteq \mathcal{A}$, $T \subseteq \mathcal{T}$. We say that c is a *constraint over E*,
- f is a function $f : \mathcal{E}^* \rightarrow \{\mathbf{true}, \mathbf{false}\}$. Let $\sigma \in \mathcal{E}^*$ be a trace. We denote $f(\sigma) = \mathbf{true}$ by $\sigma \models f$ and $f(\sigma) = \mathbf{false}$ by $\sigma \not\models f$.
 - If $f(\sigma) = \mathbf{true}$, then we say that σ *satisfies c*, denoted by $\sigma \models c$,
 - If $f(\sigma) = \mathbf{false}$, then we say that σ *violates c*, denoted by $\sigma \not\models c$,

- it holds that $\forall \gamma \in (\sigma \downarrow^E) : f(\gamma) = f(\sigma)$, i.e., satisfaction of a trace is decidable on the appearances of elements from the namespace in that trace,
- We say that $\mathcal{E}_{\models c}^* = \{\sigma \in \mathcal{E}^* \mid \sigma \models c\}$ is a set of all traces that satisfy constraint c .

Further on, we use the following shorthand notation $\Pi_A(c) = A$, $\Pi_T(c) = T$ and $\Pi_f(c) = f$. We use \mathcal{C} to denote the set of all constraints. \square

A constraint specifies a relation between events contained in its namespace. Using events instead of plain activities in the constraint namespace enables defining more sophisticated rules. For example, constraints $c_1 =$ “cannot *start perform surgery* before *completing perform X ray*” and $c_2 =$ “cannot *start perform surgery* before *starting perform X ray*” are semantically different. In case of the first constraint c_1 , *perform surgery* can start only after completing the *perform X ray*. In case of the second constraint c_2 , *perform surgery* can start immediately after starting the *perform X ray*, i.e., activity *perform surgery* can be *started* and *completed* even if the X ray photo is not available yet or if the activity *perform X ray* fails (does not complete).

Since an execution is represented by a trace, a constraint is a boolean expression that evaluates to **true** or **false** for every trace $\sigma \in \mathcal{E}^*$. Because a constraint defines a relation between elements in its namespace, trace satisfaction should be decidable only on the occurrences and positions of namespace elements in the trace, i.e., if a trace satisfies the constraint, then all traces in the projection of the trace on the namespace also satisfy the constraint and vice versa. Example 4.1.5 illustrates the notions of a constraint, namespace and satisfaction of traces.

Example 4.1.5. (Constraints)

The medical department from Example 4.1.2 needs to follow two constraints $c_1, c_2 \in \mathcal{C}$ where:

- $c_1 = (E_1, f_1)$ is a constraint over $E_1 = \{e_{sc}, e_{rc}\}$ specifying that $f_1 =$ “If *perform surgery* is *completed* then, afterwards at some point in time, *prescribe rehabilitation* is also *completed*.”¹, i.e., $f_1 =$ “event e_{sc} is eventually followed by event e_{rc} ”, and
- $c_2 = (E_2, f_2)$ is a constraint over $E_2 = \{e_{es}, e_{ec}, e_{ex}\}$ specifying that $f_2 =$ “Cannot execute any other activity until *completing examine patient*.”, i.e., $f_2 =$ “only events e_{es} and e_{ex} are possible before event e_{ec} .”

For the three traces (patients) σ_1, σ_2 and σ_3 from Example 4.1.2 it holds that:

1. trace $\sigma_1 = \langle e_{es}, e_{ec}, e_{ms}, e_{mc}, e_{ss}, e_{sc} \rangle$
 - violates c_1 (i.e., $\sigma_1 \not\models c_1$) because event e_{sc} is not followed by event e_{rc} in σ_1 , i.e., this patient had a surgery but did not have a rehabilitation after the surgery; for all traces $\gamma \in (\sigma_1 \downarrow^{E_1})$ (i.e., all traces of form $\langle e_{*1}, e_{*2}, e_{*3}, e_{*4}, e_{*5}, e_{sc} \rangle$, with $\forall 1 \leq i \leq 5 e_{*i} \notin E_1$) it holds that $\gamma \not\models c_1$,

¹The exact formal definition of constraints is not relevant at this point.

- satisfies c_2 (i.e., $\sigma_1 \models c_2$) because event e_{ec} is preceded only by event e_{es} in σ_1 , i.e., this patient was examined at the beginning of the treatment; for all traces $\gamma \in (\sigma_1^{\downarrow E_2})$ (i.e., all traces of form $\langle e_{es}, e_{ec}, e_{*1}, e_{*2}, e_{*3}, e_{*4} \rangle$, with $\forall_{1 \leq i \leq 5} e_{*i} \notin E_2$) it holds that $\gamma \models c_2$,
2. trace $\sigma_2 = \langle e_{es}, e_{ec}, e_{ss}, e_{sc}, e_{ms}, e_{mc}, e_{rs}, e_{rc} \rangle$
- satisfies c_1 (i.e., $\sigma_2 \models c_1$) because event e_{sc} is followed by event e_{rc} in σ_2 , i.e., this patient had a surgery and a rehabilitation after the surgery; for all traces $\gamma \in (\sigma_2^{\downarrow E_1})$ (i.e., all traces of form $\langle e_{*1}, e_{*2}, e_{*3}, e_{sc}, e_{*4}, e_{*5}, e_{*6}, e_{rc} \rangle$, with $\forall_{1 \leq i \leq 6} e_{*i} \notin E_1$) it holds that $\gamma \models c_1$,
 - satisfies c_2 (i.e., $\sigma_2 \models c_2$); for all traces $\sigma \in (\sigma_2^{\downarrow E_2})$ (i.e., all traces of form $\langle e_{es}, e_{ec}, e_{*1}, e_{*2}, e_{*3}, e_{*4}, e_{*5}, e_{*6} \rangle$, with $\forall_{1 \leq i \leq 6} e_{*i} \notin E_2$) it holds that $\sigma \models c_2$,
3. trace $\sigma_3 = \langle e_{ms}, e_{mc}, e_{es}, e_{ec}, e_{ms}, e_{mc} \rangle$
- satisfies c_1 (i.e., $\sigma_3 \models c_1$) because there is no event e_{sc} in trace σ_3 to be followed by event e_{rc} , i.e., this patient did not have a surgery; for all traces $\gamma \in (\sigma_3^{\downarrow E_1})$ (i.e., all traces of form $\langle e_{*1}, e_{*2}, e_{*3}, e_{*4}, e_{*5}, e_{*6} \rangle$, with $\forall_{1 \leq i \leq 6} e_{*i} \notin E_1$) it holds that $\gamma \models c_1$,
 - violates c_2 (i.e., $\sigma_3 \not\models c_2$) because there are events other than e_{es} and e_{ex} in trace σ_3 before event e_{ec} , i.e., medications were prescribed before the examination for this patient; for all traces $\gamma \in (\sigma_3^{\downarrow E_2})$ (i.e., all traces of form of $\langle e_{*1}, e_{*2}, e_{es}, e_{ec}, e_{*3}, e_{*4} \rangle$, with $\forall_{1 \leq i \leq 4} e_{*i} \notin E_2$) it holds that $\gamma \not\models c_2$.

Note that the set of satisfying traces $\mathcal{E}_{\models c_1}^*$ and $\mathcal{E}_{\models c_2}^*$ of constraints c_1 and c_2 , respectively, are infinite sets, e.g., *prescribe rehabilitation* can be repeated an arbitrary number of times. Also, a set of satisfying traces $\mathcal{E}_{\models c}^*$ of constraint $c = (E, f)$ contains all sequences (traces) of all events $e \in \mathcal{E}$ that satisfy constraint c , i.e., traces in the set of satisfying traces contain both events from the namespace E and events not in the namespace E . \square

In this chapter we only provide informal constraint specification, i.e., a natural (i.e., English) language is used to specify the semantics of constraints. In Chapter 5 we propose a formal language that can be used (1) to specify the semantics of each constraint and (2) for retrieving a finite representation of the set of all traces that satisfy such a constraint.

4.2 Constraint Models

As specified in Definition 4.2.1, a constraint model consists of *activities*, *mandatory constraints* and *optional constraints*. Each of the constraints is over names-

pace ($A \times \mathcal{T}$), i.e., constraints can only define relationships between events involving activities from the model. If a model activity is contained in the namespace of any of the constraints, then we say that this activity is constrained by the model.

Definition 4.2.1. (Constraint model cm)

A constraint model cm is defined as a triple $cm = (A, C_M, C_O)$, where:

- $A \subseteq \mathcal{A}$ is a set of *activities* in the model,
- $C_M \subseteq \mathcal{C}$ is a set of *mandatory constraints* where every element $(E, f) \in C_M$ is a constraint over E , such that $E \subseteq (A \times \mathcal{T})$,
- $C_O \subseteq \mathcal{C}$ is a set of *optional constraints* where every element $(E, f) \in C_O$ is a constraint over E , such that $E \subseteq (A \times \mathcal{T})$.

The set of constrained activities in cm is defined as $\Pi_{CA}(cm) = \bigcup_{c \in C_M \cup C_O} \Pi_A(c)$. We use \mathcal{U}_{cm} to denote the set of all constraint models. \square

The *set of satisfying traces of a model* contains all traces that satisfy the model, i.e., traces that satisfy all mandatory constraints in the model (cf. Definition 4.2.2). If a trace is not in this set, that means that the trace violates at least one mandatory constraint in the model. According to Definition 4.2.2, any trace satisfies a constraint model without mandatory constraints. This means that users can execute such a model in any way – they can execute any of the activities from the model an arbitrary number of times (including zero times) and they can execute these activities in an arbitrary order. For example, every trace $\sigma \in \mathcal{E}^*$ satisfies a constraint model $cm = (\{A, B, C\}, \emptyset, C_O)$, i.e., $\mathcal{E}_{\models cm}^* = \mathcal{E}^*$. Note that Figure 3.36 on page 82 shows a procedural version of model $(\{A, B, C\}, \emptyset, \emptyset)$.

Definition 4.2.2. (Constraint model satisfying traces $\mathcal{E}_{\models cm}^*$)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model where $cm = (A, C_M, C_O)$. The set of satisfying traces for model cm is defined as

$$\mathcal{E}_{\models cm}^* = \begin{cases} \mathcal{E}^* & \text{if } C_M = \emptyset; \\ \bigcap_{c \in C_M} \mathcal{E}_{\models c}^* & \text{otherwise.} \end{cases}$$

If for trace $\sigma \in \mathcal{E}^*$ it holds that $\sigma \in \mathcal{E}_{\models cm}^*$, then we say that σ *satisfies* model cm . If for trace $\sigma \in \mathcal{E}^*$ it holds that $\sigma \notin \mathcal{E}_{\models cm}^*$, then we say that σ *violates* model cm . \square

As specified in Definition 4.2.2, the set of traces that satisfy a constraint model is *compositional* [73, 205] with respect to traces that satisfy each of the constraints in the model, i.e., if a trace satisfies each of the constraints, then the trace also satisfies the model.

Note that the set of traces that satisfy a constraint model $\mathcal{E}_{\models cm}^*$ can contain many traces, which makes constraint models *flexible by design* [226–228]. The set $\mathcal{E}_{\models cm}^*$ can even contain infinitely many traces. To enable computer-supported

execution of constraint models, a finite representation of $\mathcal{E}_{\models cm}^*$ is needed. In Chapter 5 we present how this finite representation can be obtained and used in the constraint-based approach.

While traces that violate mandatory constraints cannot satisfy the model, traces that violate optional constraints can still be satisfying traces of the model (cf. Definition 4.2.2). Optional constraints are used as guidelines for executions. Their satisfaction can be measured and presented to users as additional information, but it does not determine the satisfaction of the model. As their name says, it is optional if an execution trace will satisfy them or not. In other words, when it comes to optional constraints, users can deviate from the model if they prefer to do so. Therefore, optional constraints enable *flexibility by deviation* [226–228] in constraint models.

Note that the set of traces that satisfy a constraint model contains all traces that satisfy mandatory constraints, i.e., all sequences of all events $e \in \mathcal{E}$ that satisfy mandatory constraints of the model. For example, trace $\sigma \in \mathcal{E}_{\models cm}^*$ that satisfies model $cm = (A, C_M, C_O)$ can contain events involving activities that are not in the model itself, i.e., it is possible that $(a', t) \in \sigma$ where $a' \notin A$ and $t \in \mathcal{T}$. This is due to the fact that traces that satisfy constraints can contain events that are not in the namespace of the constraint (e.g., Example 4.1.5). This might seem odd, but is an important property that enables changes of models during execution. As it will be described in Section 4.5, it is possible to change constraint models while they are being executed (the so-called *ad-hoc instance change*). For example, consider model $cm = (A, C_M, C_O)$ and trace σ . It is possible to, after executing activity $a_* \in A$ (i.e., $(a_*, t_s), (a_*, t_c) \in \sigma$), remove this activity from the model. Although model cm does not contain activity a_* anymore and trace σ does (i.e., $a_* \notin A$ and $(a_*, t_s), (a_*, t_c) \in \sigma$), this trace can still satisfy the model (i.e., $\sigma \in \mathcal{E}_{\models cm}^*$) if it satisfies all mandatory constraints from the model. However, after removing a_* from a model cm it will no longer be possible to execute this activity, i.e., events involving activity a_* cannot be added to the trace σ in the future. The execution of instances of constraint models and instance change is out of scope of this section and will be described in detail in Sections 4.4 and 4.5.

A constraint-based approach to process modeling allows developing models that offer a large number of possible executions to users while still enforcing the users to follow a set of basic rules - constraints. It is often the case that constraint models have an infinite set of satisfying traces, which allows people to choose from a large number of traces that satisfy the model and to select the trace that is the most appropriate for the specific situation people are working in. Consider, for example, a simple constraint model presented in Example 4.2.3. This model consists of three activities and one constraint. Due to this constraint, only a trace where all events (*curse, completed*) are eventually followed by at least event (*pray, completed*) satisfies this model.

Example 4.2.3. (A constraint model)

Let $cm^R \in \mathcal{U}_{cm}$ be a constraint model where $cm^R = (A^R, C_M^R, C_O^R)$ such that:

- $A^R = \{\textit{pray}, \textit{curse}, \textit{bless}\}$ is a set of activities,
- $C_M^R = \{c_1\}$ is a set of mandatory constraints where $c_1 = (E_1, f_1)$ such that $E_1 = \{(\textit{curse}, \textit{completed}), (\textit{pray}, \textit{completed})\}$ and $f_1 = \text{“Every occurrence of event } (\textit{curse}, \textit{completed}) \text{ must eventually be followed by at least one occurrence of event } (\textit{pray}, \textit{completed}).\text{”}$, and
- $C_O^R = \emptyset$ is a (empty) set of optional constraints.

The set of accepting traces for this model is $\mathcal{E}_{\models cm^R}^* = \bigcap_{c \in C_M^R} \mathcal{E}_{\models c}^* = \mathcal{E}_{\models c_1}^*$. Constrained activities in the model are $\Pi_{CA}(cm^R) = \{\textit{curse}, \textit{pray}\}$ because $\Pi_A(c_1) = \{\textit{curse}, \textit{pray}\}$. \square

The model cm^R from Example 4.2.3 allows people who are executing this model to choose from an infinite number of traces that satisfy this model. Table 4.1 shows only a few examples of traces (i.e., $\sigma_1, \sigma_2, \dots, \sigma_7$) that satisfy the model cm^R and one example of a trace (i.e., σ_8) that does not satisfy this model. Each column refers to a trace and the rows refer to events in particular traces. For simplicity, we assume that activities do not overlap in these traces, i.e., each listed activity refers to a subsequent starting and completing of the activity. For example, each *bless* in the trace σ_2 refers to a sequence of two events (*bless, started*) and (*bless, completed*). The first seven traces satisfy the model. First, the empty trace satisfies the model, i.e., $\sigma_1 \in \mathcal{E}_{\models cm^R}^*$, because there is no event (*curse, completed*) in the empty trace $\sigma_1 = \langle \rangle$. Second, any of the activities can occur an arbitrary number of times in a satisfying trace (e.g., traces $\sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6$ and σ_7). Third, it is possible that activities *curse* and *pray* do not occur at all (e.g., traces σ_1 and σ_2). Fourth, it is possible that activity *pray* occurs an arbitrary number of times (1) without an occurrence of activity *curse* (e.g., traces σ_3 and σ_6) and (2) before the occurrence of activity *curse* (e.g., traces σ_4 and σ_7). Fifth, it is possible that activity *curse* occurs multiple times followed by one occurrence of activity *pray* in order to satisfy constraint c_1 (e.g., trace σ_5). Finally, it is also possible to first *curse*, then *pray*, and later again *curse* and *pray* (e.g., trace σ_7). The last trace, i.e., trace σ_8 does not satisfy model cm^R because there is no occurrence of activity *pray* after the second occurrence of activity *curse* in this trace.

Note that some of the traces in Table 4.1 satisfy the model cm^R from Example 4.2.3 although they contain events that involve activities that are *not* in the model. Traces σ_2 and σ_6 satisfy model cm^R although they contain events on activity *become holy* $\in \mathcal{A}$, which is not in the model cm^R (i.e., *become holy* $\notin A^R$). This property of satisfying traces enables easy adding and removing activities in models that are already being executed (*ad-hoc instance change*), which will be described in detail in Section 4.5. For illustration, assume that the original model $cm^{R'} = (A^R \cup \{\textit{become holy}\}, C_M^R, C_O^R)$ contained activity *become holy* and that

Table 4.1: Examples of traces that (do not) satisfy the model cm^R from Example 4.2.3

	$\sigma_i \in \mathcal{E}_{\models cm^R}^*, i \in \{1, \dots, 7\}$							$\sigma_8 \notin \mathcal{E}_{\models cm^R}^*$
	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8
1		<i>bless</i>	<i>pray</i>	<i>pray</i>	<i>curse</i>	<i>bless</i>	<i>pray</i>	<i>pray</i>
2		<i>become holy</i>		<i>curse</i>	<i>pray</i>	<i>bless</i>	<i>curse</i>	<i>curse</i>
3		<i>bless</i>		<i>curse</i>	<i>bless</i>	<i>become holy</i>	<i>bless</i>	<i>bless</i>
4				<i>bless</i>	<i>curse</i>	<i>pray</i>	<i>pray</i>	<i>pray</i>
5				<i>pray</i>	<i>bless</i>	<i>bless</i>	<i>curse</i>	<i>curse</i>
6				<i>bless</i>	<i>curse</i>		<i>bless</i>	<i>bless</i>
7					<i>pray</i>		<i>pray</i>	
8					<i>bless</i>			

users executed activities *bless* and *become holy* in the trace σ_2 and then decided to remove activity *become holy* from the model. At this point, the model did not contain activity *become holy* anymore (model cm^R), but the trace σ_3 still satisfies the new model cm^R . However, after the activity *become holy* was removed from the model it was no longer possible to trigger new events on activity *become holy*.

Adding and removing activities and optional constraints to and from constraint models does not have any effect on the set of traces that satisfy the model, because this set is not dependent on the activities in the model (cf. Definition 4.2.2). Property 4.2.4 proves that the set of traces that satisfy the model does not change when activities or optional constraints are added to or removed from the model. Consider, for example the model cm^R from Example 4.2.3 and model $cm^{R'} = (A^R \cup \{\textit{become holy}\}, C_M^R, C_O^R \cup \{c\})$ where *become holy* $\in \mathcal{A}$ and $c \in \mathcal{C}$. Although model $cm^{R'}$ has more activities and optional constraints, the same traces satisfy both models cm^R and $cm^{R'}$, because they have the same set of mandatory constraints C_M^R . For example, traces $\sigma_1, \dots, \sigma_7$ from Table 4.1 satisfy both cm^R and $cm^{R'}$ while trace σ_8 does not satisfy either cm^R nor $cm^{R'}$.

Property 4.2.4. (Activities and optional constraints have no effect on $\mathcal{E}_{\models cm}^*$)

Let $cm, cm' \in \mathcal{U}_{cm}$ be constraint models where $cm = (A, C_M, C_O)$ and $cm' = (A', C_M, C'_O)$, then $\mathcal{E}_{\models cm}^* = \mathcal{E}_{\models cm'}^*$.

Proof.

If $C_M = \emptyset$, then $\mathcal{E}_{\models cm}^* = \mathcal{E}_{\models cm'}^* = \mathcal{E}^*$ (cf. Definition 4.2.2). If $C_M \neq \emptyset$, the set of traces that satisfy cm is $\mathcal{E}_{\models cm}^* = \bigcap_{c \in C_M} \mathcal{E}_{\models c}^*$ (cf. Definition 4.2.2) and the set of traces that satisfy cm' is $\mathcal{E}_{\models cm'}^* = \bigcap_{c \in C_M} \mathcal{E}_{\models c}^*$, i.e., $\mathcal{E}_{\models cm}^* = \mathcal{E}_{\models cm'}^*$. \square

Property 4.2.5 shows that, the more mandatory constraints the model has, the fewer traces can satisfy the model. In other words, the more mandatory constraints a model has, the less flexibility the users will have while executing the model.

Property 4.2.5. (Mandatory constraints can influence $\mathcal{E}_{\models cm}^*$)

Let $cm, cm' \in \mathcal{U}_{cm}$ be constraint models where $cm = (A, C_M, C_O)$ and $cm' = (A', C'_M, C'_O)$ such that $C_M \subset C'_M$, then $\mathcal{E}_{\models cm'}^* \subseteq \mathcal{E}_{\models cm}^*$.

Proof. If $C_M = \emptyset$ then $\mathcal{E}_{\models cm}^* = \mathcal{E}^*$ and $C'_M \neq \emptyset$. Further it holds that $\mathcal{E}_{\models cm'}^* = \bigcap_{c' \in C'_M} \mathcal{E}_{\models c'}^*$ and $\mathcal{E}_{\models cm'}^* \subseteq \mathcal{E}^*$, i.e., $\mathcal{E}_{\models cm'}^* \subseteq \mathcal{E}_{\models cm}^*$.

If $C_M \neq \emptyset$ then $\mathcal{E}_{\models cm}^* = \bigcap_{c \in C_M} \mathcal{E}_{\models c}^*$ and $C'_M \neq \emptyset$. Further it holds that $\mathcal{E}_{\models cm'}^* = \bigcap_{c \in (C_M \cup (C'_M \setminus C_M))} \mathcal{E}_{\models c}^* = \bigcap_{c \in C_M} \mathcal{E}_{\models c}^* \cap \bigcap_{c \in (C'_M \setminus C_M)} \mathcal{E}_{\models c}^* = \mathcal{E}_{\models cm}^* \cap \bigcap_{c \in (C'_M \setminus C_M)} \mathcal{E}_{\models c}^*$, i.e., $\mathcal{E}_{\models cm'}^* \subseteq \mathcal{E}_{\models cm}^*$. \square

Consider, for example, the model cm^R from Example 4.2.3 and a constraint $c_2 \in \mathcal{C}$ where $c_2 = (E_2, f_2)$ such that $E_2 = \{(pray, completed)\}$ and $f_2 =$ “Complete activity *pray* at least once.”. Let $cm^{R'}$ be a constraint model where c_2 is added as a mandatory constraint to model cm^R , i.e., $cm^{R'} = (A^R, C_M^R \cup \{c_2\}, C_O^R)$. The set of satisfying traces of model cm^R contains all traces $\sigma \in \mathcal{E}^*$ that satisfy constraint c_1 , while the set of satisfying traces of model $cm^{R'}$ contains all traces $\sigma \in \mathcal{E}^*$ that satisfy both constraint c_1 and constraint c_2 . In other words, the set of traces that satisfy $cm^{R'}$ is obtained by removing all traces that do not satisfy constraint c_2 from the set of traces that satisfy model cm^R . For example, although they satisfy model cm^R , traces σ_1 and σ_2 from Table 4.1 do not satisfy model $cm^{R'}$ because they do not satisfy constraint c_2 .

Understanding the effect of adding/removing mandatory constraints to/from a constraint model change is important for several purposes. First, while creating a model it is important to understand that the more mandatory constraints the model has, the less freedom (options) the users will have while executing the model and vice versa, i.e., fewer mandatory constraints mean more freedom for users. Second, in case of ad-hoc instance change, *only* adding mandatory constraints can cause errors and thus cause the instance change to fail, as it will be discussed in Section 4.5. In order to successfully perform such a runtime change after all, it is necessary to resolve (eliminate) this error, which can be achieved *only* by removing mandatory constraints. Third, errors can be introduced in models *only* by adding mandatory constraints and vice versa, errors can be eliminated *only* by removing mandatory constraints, as it will be described in Section 4.6.

4.3 Illustrative Example: The Fractures Treatment Process

As opposed to traditional models where all possible executions have to be predicted in detail in advance (i.e., during modeling), developers of constraint models need to specify a number of constraints (rules) that should be followed during the execution. This set of constraints indirectly determines the set of possible

executions of the constraint model – any execution that satisfies mandatory constraints is possible. Moreover, it is often the case that users can choose from an infinite number of possible executions of the constraint model and, therefore, adjust each execution to the specific situation as long as all mandatory constraints are satisfied. Consider the medical Fractures Treatment process described in Example 4.3.1. Although it consists of several important rules that have to be satisfied in order to prevent mistakes, the medical staff must make situation-dependent decisions and treat each patient in a way that is the most suitable for the specific patient's fracture.

Example 4.3.1. (Fractures Treatment process definition)

A team of medical staff is in charge of treatment of patients with fractures. The treatment of every patient begins with the *examination* of the patient, although the patient can be examined (again) at multiple stages during the treatment. Depending on the type(s) of the fracture(s), the staff can apply different types of treatments:

- *Applying cast* is the most common treatment for fractures. The *cast is removed* after the fracture has healed.
- Dislocations are treated by *repositioning*.
- A *surgery* can be performed for complex injuries.
- When none of the above mentioned treatments can be applied, the patient uses a *sling* for a prescribed period of time.

In cases of complex fractures, the staff can decide to use an arbitrary combination of the above mentioned treatment procedures.

It is obligatory to take *X ray* of the fracture(s) before *applying of cast, repositioning* or performing the *reposition* or *surgery*. If needed, *X ray* photos can be taken several times during the treatment. Due to danger of X rays, it is important to avoid any mistakes and the staff needs to *check risks* of X rays (e.g., pregnancy) and to take these risks into consideration each time before taking an X ray.

The staff can *prescribe necessary medication* or *prescribe rehabilitation* an arbitrary number of times during the treatment. In general, the hospital has the policy to send patients to *rehabilitation* at least once after a *surgery*. □

When developing a constraint model for the process described in Example 4.3.1, it is not necessary to predict all possible treatment tracks in advance. Moreover, the medical staff that treats fractures can perform many different treatments for different patients. Instead of predicting all possible treatments, it is sufficient to detect the activities and the constraints that should be followed during treatments. The constraint model for the Fractures Treatment process is given below.

Example 4.3.2. (Fractures Treatment constraint model)

Recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types where $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$. Model $cm^{FT} \in \mathcal{U}_{cm}$, $cm^{FT} = (A^{FT}, C_M^{FT}, C_O^{FT})$ is a constraint model for the Fractures Treatment process described in Example 4.3.1 such that

- $A^{FT} = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}\}$ is a set of activities where:

$a_1 = \text{examine patient},$	$a_6 = \text{prescribe sling},$
$a_2 = \text{apply cast},$	$a_7 = \text{prescribe medication},$
$a_3 = \text{remove cast},$	$a_8 = \text{check X ray risk},$
$a_4 = \text{perform reposition},$	$a_9 = \text{perform X ray, and}$
$a_5 = \text{perform surgery},$	$a_{10} = \text{prescribe rehabilitation},$

- $C_M^{FT} = \{c_1, c_2, c_3, c_4, c_5\}$ is a set of mandatory constraints where:
 - $c_1 = (E_1, f_1)$ where $E_1 = \{(a_1, t_s), (a_1, t_c)(a_1, t_x)\}$ and $f_1 = \text{“Start every treatment with an occurrence of the activity } \textit{examine patient} \text{”}$, i.e., $f_1 = \text{“Before the first occurrence of event } (a_1, t_c), \text{ only occurrences of events } (a_1, t_s) \text{ or } (a_1, t_x) \text{ are possible”}$,
 - $c_2 = (E_2, f_2)$ where $E_2 = \{(a_2, t_c), (a_4, t_c), (a_5, t_c), (a_6, t_c)\}$ and $f_2 = \text{“Perform at least one of the procedures: } \textit{apply cast}, \textit{perform reposition}, \textit{perform surgery} \text{ or } \textit{prescribe sling} \text{”}$, i.e., $f_2 = \text{“At least one of the events } (a_2, t_c), (a_4, t_c), (a_5, t_c) \text{ or } (a_6, t_c) \text{ must occur”}$,
 - $c_3 = (E_3, f_3)$ where $E_3 = \{(a_2, t_c), (a_3, t_s), (a_3, t_c)\}$ and $f_3 = \text{“Can } \textit{remove cast} \text{ only after } \textit{applying cast} \text{ and always } \textit{remove cast} \text{ after } \textit{applying cast} \text{”}$, i.e., $f_3 = \text{“Event } (a_3, t_s) \text{ can occur only after occurrence of event } (a_2, t_c) \text{ and every occurrence of event } (a_2, t_c) \text{ must eventually be followed by at least one occurrence of event } (a_3, t_c) \text{”}$,
 - $c_4 = (E_4, f_4)$ where $E_4 = \{(a_2, t_s), (a_4, t_s), (a_5, t_s), (a_9, t_c)\}$ and $f_4 = \text{“Must } \textit{perform X ray} \text{ before } \textit{applying cast}, \textit{repositioning} \text{ and } \textit{perform surgery} \text{”}$, i.e., $f_4 = \text{“Events } (a_2, t_s), (a_4, t_s) \text{ and } (a_5, t_s) \text{ can occur only after occurrence of event } (a_9, t_c) \text{”}$,
 - $c_5 = (E_5, f_5)$ where $E_5 = \{(a_8, t_c), (a_9, t_s)\}$ and $f_5 = \text{“Check X ray risk before each new occurrence of activity } \textit{perform X ray} \text{”}$, i.e., $f_5 = \text{“Each new occurrence of event } (a_9, t_s) \text{ must be preceded by a at least one new occurrence of } (a_8, t_c) \text{”}$; and
- $C_O^{FT} = \{c_6\}$ is a set of optional constraints where $c_6 = (E_6, f_6)$ such that $E_6 = \{(a_5, t_c), (a_{10}, t_c)\}$ and $f_6 = \text{“Eventually } \textit{prescribe rehabilitation} \text{ after } \textit{perform surgery} \text{”}$, i.e., $f_6 = \text{“Every occurrence of event } (a_5, t_c) \text{ must eventually be followed by at least one occurrence of event } (a_{10}, t_c) \text{”}$.

□

The Fractures Treatment model allows for an infinite number of different treatments of patients, as long as all mandatory constraints are satisfied. For

example, although every treatment has to start with the execution of the *examine patient* activity (constraint c_1), the staff can execute this activity multiple times at later stages of the treatment. At least one of the four procedures (*apply cast*, *perform reposition*, *perform surgery*, *prescribe sling*) has to be completed (constraint c_2), but it is always possible to combine several procedures during the treatment of one patient. The activity *remove cast* can be started only after the activity *apply cast* had been completed at least once and after activity *apply cast* has been completed activity *remove cast* will be completed at least once (constraint c_3). This constraint still makes it possible to handle situations when patients have two fractures both treated with cast for different periods of time. Only after completing activity *perform X ray* the staff can apply one of the four procedures (constraint c_4). However, it is still possible to execute the *perform X ray* activity several times if necessary in the treatment, as long as the *check X ray risk* activity is completed before starting each new *perform X ray* activity (constraint c_5). Although it is hospital policy to send all patients to rehabilitation after a completed surgery, it is possible to perform multiple surgeries and only after them one or more activities *prescribe rehabilitation* (optional constraint c_6). Also, it is possible to execute the activity *prescribe rehabilitation* for patients that did not undergo a surgery. Moreover, because c_6 is an optional constraint, it is also possible not to *prescribe rehabilitation* after a *surgery*.

Table 4.2 shows examples of six *execution traces* for different patients undergoing the Fractures Treatment process according to the cm^{FT} model from Example 4.3.2². For reasons of simplicity, we assume that executions of activities did not overlap, i.e., each of the listed activities $a \in A^{FT}$ refers to executions of two successive events: (a , *started*) and (a , *completed*). Each of the first five traces satisfies the cm^{FT} model (i.e., $\sigma_1 \in \mathcal{E}_{\models cm^{FT}}^*$, $\sigma_2 \in \mathcal{E}_{\models cm^{FT}}^*$, $\sigma_3 \in \mathcal{E}_{\models cm^{FT}}^*$, $\sigma_4 \in \mathcal{E}_{\models cm^{FT}}^*$ and $\sigma_5 \in \mathcal{E}_{\models cm^{FT}}^*$, denoted by the \checkmark symbol) because each of these five traces satisfies all mandatory constraints in the model cm^{FT} . This means that the first five patients were handled correctly according to the Fractures Treatment model cm^{FT} . Note that despite the fact that trace σ_3 violates the optional constraint c_6 from the model (because the staff did not *prescribe rehabilitation* after the activity *perform surgery* was executed), this trace satisfies the model cm^{FT} because it satisfies all its mandatory constraints. The last trace, trace σ_6 violates the constraint model (i.e., $\sigma_6 \notin \mathcal{E}_{\models cm^{FT}}^*$, denoted by the \boxtimes symbol) because it violates its mandatory constraint c_5 : for this patient the activity *check X ray risk* was not executed before the activity *perform X ray*.

²For the purpose of simplicity we give examples of traces that contain only activities from the cm^{FT} model.

Table 4.2: Examples of traces for six patients in the Fractures Treatment model from Example 4.3.2

	σ_1 - patient 1 ✓	σ_2 - patient 2 ✓	σ_3 - patient 3 ✓
1	<i>examine patient</i>	<i>examine patient</i>	<i>examine patient</i>
2	<i>prescribe sling</i>	<i>check X ray risk</i>	<i>check X ray risk</i>
3	<i>check X ray risk</i>	<i>prescribe medication</i>	<i>perform X ray</i>
4	<i>perform X ray</i>	<i>perform X ray</i>	<i>prescribe sling</i>
5	<i>apply cast</i>	<i>perform reposition</i>	<i>examine patient</i>
6	<i>prescribe medication</i>	<i>prescribe medication</i>	<i>perform surgery</i>
7	<i>remove cast</i>	<i>check X ray risk</i>	<i>examine patient</i>
8	<i>examine patient</i>	<i>perform X ray</i>	<i>perform surgery</i>
9		<i>examine patient</i>	<i>prescribe medication</i>
10			<i>prescribe rehabilitation</i>
	σ_4 - patient 4 ✓	σ_5 - patient 5 ✓	σ_6 - patient 6 ☒
1	<i>examine patient</i>	<i>examine patient</i>	<i>examine patient</i>
2	<i>prescribe sling</i>	<i>check X ray risk</i>	<i>perform X ray</i> ☒
3	<i>prescribe rehabilitation</i>	<i>perform X ray</i>	
4		<i>perform surgery</i>	
5		<i>examine patient</i>	
6		<i>prescribe medication</i>	

4.4 Execution of Constraint Model Instances

A constraint model can be executed an arbitrary number of times. We refer to one execution of a constraint model as to a *constraint model instance*. Actions that resources take while executing one instance form the execution trace of the instance. A constraint model instance consists of a constraint model and the instance trace, as specified in Definition 4.4.1. Consider, for example, the traces presented in Table 4.2. These traces could belong to six instances of the Fractures Treatment model cm^{FT} : instance $ci_1 = (\sigma_1, cm^{FT})$ relates to treatment of the “patient 1”, $ci_2 = (\sigma_2, cm^{FT})$ to “patient 2”, etc.

Definition 4.4.1. (Constraint model instance ci)

A constraint model instance ci is defined as a pair $ci = (\sigma, cm)$, where:

- $\sigma \in \mathcal{E}^*$ is the instance’s trace, and
- $cm \in \mathcal{U}_{cm}$ is a constraint model.

We use \mathcal{U}_{ci} to denote the set of all constraint instances. □

Note that not all constraint models need to have related instances and that there can be an arbitrary number of instances having the same constraint model, where each instance has its own trace. For example, Figure 4.3 shows four instances and four models. There are two instances of model cm_1 (i.e., instances $ci_1 = (\sigma_1, cm_1)$ and $ci_2 = (\sigma_2, cm_1)$), one instance of model cm_3 (i.e., instance

$ci_3 = (\sigma_3, cm_3)$) and one instance of model cm_4 (i.e., instance $ci_4 = (\sigma_4, cm_4)$). There are no instances of model cm_2 .

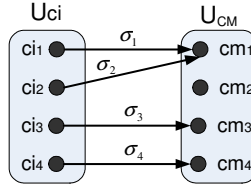


Figure 4.3: Constraint instances

In the remainder of this section we describe three important issues when it comes to execution of instances of constraint-based models. First, in Section 4.4.1 we describe how state of an instance changes during execution. Second, in Section 4.4.2 we describe how instances are executed by triggering the so-called enabled events. Finally, in Section 4.4.3 we describe the relationship between the state of an instance and states of its constraints.

4.4.1 Instance State

During execution, the *state* of an instance can change depending on the instance trace. As specified in Definition 4.4.2, an instance is *satisfied* if the instance trace satisfies the model of the instance. If the instance trace does not satisfy the instance model, but there is a suffix that could be added to the trace such that it satisfies the model, the instance is classified as *temporarily violated*. Finally, if the trace does not satisfy the model and the trace cannot satisfy the model whatever suffix would be added, then the instance is classified as *violated*.

Definition 4.4.2. (Instance state ω)

Let $ci \in \mathcal{U}_{ci}$ be an instance where $ci = (\sigma, cm)$. The function $\omega : \mathcal{U}_{ci} \rightarrow \{\textit{satisfied}, \textit{temporarily violated}, \textit{violated}\}$ of instance ci is defined as:

$$\omega(ci) = \begin{cases} \textit{satisfied} & \text{if } \sigma \in \mathcal{E}_{\text{F}cm}^*; \\ \textit{temporarily violated} & \text{if } (\sigma \notin \mathcal{E}_{\text{F}cm}^*) \wedge (\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\text{F}cm}^*); \\ \textit{violated} & \text{otherwise.} \end{cases}$$

□

Table 4.3 shows four possible instances for the Fractures Treatment model from Example 4.3.2 and their states. For each instance a sequence of events from the instance trace is given. In addition, for each instance a state change is given for each of event in the instance trace. For reasons of simplicity, we assume that executions of activities did not overlap, i.e., each of the listed activities $a \in A^{FT}$ refers to executions of two successive events: $(a, \textit{started})$ and $(a, \textit{completed})$. Due to constraints c_1 and c_2 , each instance is *temporarily violated* at the beginning

of the execution. After the execution of activities *examine patient* and *prescribe sling* in the first instance ci_1 , this instance is *satisfied* because its trace at that moment satisfies all mandatory constraints. The state of the first instance remained *satisfied* after performing the next two activities: *check X ray risk* and *perform X ray*. However, after the activity *apply cast* was executed, the state of the instance became again *temporarily violated*. This is because this partial trace does not satisfy the model (does not satisfy mandatory constraint c_3), but it was possible to eventually execute activity *remove cast* and return the instance into state *satisfied*. Optional constraints do not influence the state of the instance: the state of the third instance is *satisfied* although the optional constraint c_6 is not satisfied, i.e., there is not *prescribe rehabilitation* after *perform surgery*. The last instance (ci_4) ends in the state *violated* and it is not possible to extend this trace with some suffix that would return the instance to the state *satisfied* because *perform X ray* was executed before *check X ray risk* despite the constraint c_5 .

Table 4.3: States of four instances of the Fractures Treatment model from Example 4.3.2

	$ci_1 = (\sigma_1, cm^{FT})$ - patient 1		$ci_2 = (\sigma_2, cm^{FT})$ - patient 2	
	σ_1	$\omega(ci_1)$	σ_2	$\omega(ci_2)$
start		tv		tv
1	<i>examine patient</i>	tv	<i>examine patient</i>	tv
2	<i>prescribe sling</i>	sat	<i>check X ray risk</i>	tv
3	<i>check X ray risk</i>	sat	<i>perform X ray</i>	tv
4	<i>perform X ray</i>	sat	<i>perform surgery</i>	sat
5	<i>apply cast</i>	tv	<i>examine patient</i>	sat
6	<i>prescribe medication</i>	tv	<i>prescribe medication</i>	sat
7	<i>remove cast</i>	sat		
8	<i>examine patient</i>	sat		
	$ci_3 = (\sigma_3, cm^{FT})$ - patient 3		$ci_4 = (\sigma_4, cm^{FT})$ - patient 4	
	σ_3	$\omega(ci_3)$	σ_4	$\omega(ci_4)$
start		tv		tv
1	<i>examine patient</i>	tv	<i>examine patient</i>	tv
2	<i>prescribe sling</i>	sat	<i>perform X ray</i>	v
3	<i>prescribe rehabilitation</i>	sat		

(‘sat’=*satisfied*, ‘tv’=*temporarily violated* and ‘v’=*violated*)

An instance can be *successfully closed* (i.e., users can stop working on an instance) if and only if the instance state is *satisfied*. For example, the medical staff can successfully close the first instance from Table 4.3 *only* after activities number two, three, four, seven or eight because the instance state is *satisfied* after execution of these activities. Similarly, the second instance ci_2 can be closed *only* after the fourth activity, the third instance ci_3 can be closed after the second activity while the fourth instance ci_4 can never be successfully closed.

4.4.2 Enabled Events

An instance changes state by adding events to its partial trace. If added to the instance trace, some events would put the instance into the *satisfied* state, some into the *temporarily violated* state and some into the *violated* state. As specified in Definition 4.4.3, an event is enabled if and only if it refers to an activity contained in the instance model and if adding this event to the instance trace does not put the instance in the state *violated*. Hence, the DECLARE prototype presented in Chapter 6 does not allow instance ci_4 .

Definition 4.4.3. (Enabled event $ci[e]$)

Let $ci \in \mathcal{U}_{ci}$ be an instance where $ci = (\sigma, (A, C_M, C_O))$ and $e \in \mathcal{E}$ an event where $e = (a, t)$. Event e is *enabled*, denoted as $ci[e]$, if and only if $a \in A$ and $\omega(\sigma + \langle(a, t)\rangle, (A, C_M, C_O)) \neq \text{violated}$. \square

Users execute instances by executing events. Each executed event is added to the instance trace. By allowing users to execute only enabled events, the execution rule from Definition 4.4.4 makes sure that users cannot execute events that would bring the instance into the state *violated*. In other words, this rule makes sure that users can eventually bring the instance to the state *satisfied*. Also, by allowing users to execute only enabled events, the execution rule makes sure that only activities belonging to the instance model can be executed and added to the instance trace. Note that it is possible that the instance trace already contains events involving activities that once were in the model, but no longer are. The execution rule allows this, as long as the new event that is being added to the trace refers to an activity that from the model at the moment of execution.

Definition 4.4.4. (Execution rule $[-]_-$)

We define the execution relation $[-]_- \subseteq \mathcal{U}_{ci} \times \mathcal{E} \times \mathcal{U}_{ci}$ as the smallest relation satisfying $\forall_{(\sigma, cm) \in \mathcal{U}_{ci}} \forall_{e \in \mathcal{E}} : (\sigma, cm)[e] \Rightarrow (\sigma, cm)[e](\sigma + \langle e \rangle, cm)$. \square

Table 4.4 shows enabled and executed activities for one instance of the Fractures Treatment model cm^{FT} from Example 4.3.2. For the simplicity, we assume that execution of activities do not overlap, i.e., each of the listed activities $a \in A^{FT}$ in Table 4.4 refers to executions of two successive events: $(a, \text{started})$ and $(a, \text{completed})$ ³. At every moment during the execution of the instance some activities were enabled (marked with symbol “o”). For example, the first row in Table 4.4 shows that only events related to activity *examine patient* are enabled at the beginning of the execution. This is because the constraint c_1 from the model cm^{FT} specifies that each instance has to start with *examine patient*. At this point, events involving other activities are disabled because adding them to the execution trace would put the instance in the state *violated*. The execution

³This assumption is taken in order to simplify the examples of traces presented in Table 4.4. The same holds for traces that contain overlapping activities.

rule (cf. Definition 4.4.4) makes sure that users cannot bring instances to the state *violated* while executing instances by only allowing execution of enabled events. Table 4.4 shows that users can indeed execute (shown by the “●” symbol) only enabled events. At the beginning of the instance *examine patient* was executed, then *check X ray risk*, etc.

Table 4.4: Enabled (○) and executed (●) events for instance $ci = (\sigma, cm^{FT})$

	instance trace - σ	activities in the Fractures Treatment model									
		<i>examine patient</i>	<i>apply cast</i>	<i>remove cast</i>	<i>perform reposition</i>	<i>perform surgery</i>	<i>prescribe sling</i>	<i>prescribe medication</i>	<i>check X ray risk</i>	<i>perform X ray</i>	<i>prescribe rehabilitation</i>
1	<i>examine patient</i>	○●									
2	<i>check X ray risk</i>	○					○	○	○●		○
3	<i>prescribe medication</i>	○					○	○●	○	○	○
4	<i>perform X ray</i>	○					○	○	○	○●	○
5	<i>perform reposition</i>	○	○		○●	○	○	○	○		○
6	<i>prescribe medication</i>	○	○		○	○	○	○●	○		○
7	<i>check X ray risk</i>	○	○		○	○	○	○	○●		○
8	<i>perform X ray</i>	○	○		○	○	○	○	○	○●	○
9	<i>examine patient</i>	○●	○		○	○	○	○	○		○

Consider the *violated* instance ci_4 from Table 4.3. If users would execute activities *examine patient* and *perform X ray*, this would bring the instance in the state *violated* because it would not be possible to satisfy the mandatory constraint c_5 from the model cm^{FT} anymore. The execution rule prevents this: event (*perform X ray, started*) would not be enabled after the activity *examine patient* is executed (cf. Table 4.4) and users will not be able to execute activity *perform X ray* at this stage. Therefore, the *violated* instance (ci_4) in Table 4.3 is an example of an instance of the Fractures Treatment model that cannot be executed in the DECLARE prototype (cf. Chapter 6).

Note that some constraints from the Fractures Treatment model are the so-called ‘safety’ constraints, i.e., they will prevent users to execute certain events that would bring the instance to the *violated* state. For example, constraint c_4 specifies that events (*apply cast, t_s*), (*perform surgery, t_s*) and (*perform reposition, t_s*) can occur only after event (*perform X ray, t_c*) has occurred. A direct consequence of this is that events (*apply cast, t_s*), (*perform surgery, t_s*) and (*perform reposition, t_s*) will be disabled before the execution of event (*perform X ray, t_c*), as shown in Table 4.4. In other words, it is not possible to execute activities *apply cast*, *perform reposition* and *per-*

form surgery until activity perform X ray is executed. Other types of constraints represent some ‘expectation’ that has to be met in order to bring the instance to the state *satisfied*. For example, constraint c_2 specifies that “At least one of the events (*perform reposition*, t_c), (*apply cast*, t_c), (*prescribe sling*, t_c) or (*perform surgery*, t_c) must occur”. This type of a constraint will not influence enabled events, but it will prevent closing an instance before the expectation is met because the instance becomes *satisfied* only after the first occurrence of one of the four events.

An event is enabled even if adding the event to the trace violates an optional constraint, i.e., the execution rule allows triggering events that permanently violate optional constraints (cf. definitions 4.4.3 and 4.4.4). Although optional constraints do not directly influence the enabled events of constraint instances, they play an important role in the execution of instances. Optional constraints represent ‘light’ rules that are used as guidance for users but users are not forced to follow them. In Chapter 6 we will present the DECLARE prototype where, if the user is about to violate an optional constraint (by triggering an event), a special warning is issued and the user can choose whether to proceed (i.e., trigger the event and violate the optional constraint) or to abort (i.e., the event is not triggered and the optional constraint is not violated). In other words, optional constraints allow users to deviate from the constraint model and thus enable *flexibility by deviation* [226–228] (cf. Section 3.2.4).

4.4.3 States of Constraints

Similarly like the instance itself, the instance’s constraints also change states during execution. Definition 4.4.5 specifies that, given an execution trace, a constraint can be in one of the three states: *satisfied temporarily violated* and *violated*. During execution of each instance, the state of each constraint in the instance can be presented to users as additional information that can help them understand the instance they are working on.

Definition 4.4.5. (Constraint state ν)

Let $c \in \mathcal{C}$ be a constraint and $\sigma \in \mathcal{E}^*$ an execution trace. The function $\nu : (\mathcal{C} \times \mathcal{E}^*) \rightarrow \{\textit{satisfied}, \textit{temporarily violated}, \textit{violated}\}$ is defined as:

$$\nu(\sigma, c) = \begin{cases} \textit{satisfied} & \text{if } \sigma \in \mathcal{E}_{\models c}^*; \\ \textit{temporarily violated} & \text{if } (\sigma \notin \mathcal{E}_{\models c}^*) \wedge (\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models c}^*); \\ \textit{violated} & \text{otherwise.} \end{cases}$$

□

Table 4.5 shows states of constraints during the execution of one instance Fractures Treatment model from Example 4.3.2. At the beginning of the execution all constraints are in the state *satisfied*, except constraint c_2 , which is *temporarily violated* because it specifies that at least one of the activities *apply*

cast, *perform reposition*, *perform surgery* or *prescribe sling* has to be executed. Indeed, the constraint c_2 becomes *satisfied* only after the activity *prescribe sling* is executed. After the activity *apply cast* is executed, constraint c_3 becomes *temporarily violated* because this constraint requires that activity *remove cast* has to be executed after activity *apply cast*. Therefore, this constraint remains in the *temporarily violated* state until activity *remove cast* is executed. Execution of the activity *perform surgery* brings optional constraint c_6 into state *temporarily violated*. This constraint remains *temporarily violated* because activity *prescribe rehabilitation* is never executed after activity *perform surgery*.

Table 4.5: Constraint states in an instance (σ, cm^{FT}) of the Fractures Treatment model from Example 4.3.2

	instance trace σ	instance state	instance constraints					
			c_1	c_2	c_3	c_4	c_5	c_6
start		tv	sat	tv	sat	sat	sat	sat
1	<i>examine patient</i>	tv	sat	tv	sat	sat	sat	sat
2	<i>prescribe sling</i>	sat	sat	sat	sat	sat	sat	sat
3	<i>check X ray risk</i>	sat	sat	sat	sat	sat	sat	sat
4	<i>perform X ray</i>	sat	sat	sat	sat	sat	sat	sat
5	<i>perform surgery</i>	sat	sat	sat	sat	sat	sat	tv
6	<i>examine patient</i>	sat	sat	sat	sat	sat	sat	tv
7	<i>apply cast</i>	tv	sat	sat	tv	sat	sat	tv
8	<i>prescribe medication</i>	tv	sat	sat	tv	sat	sat	tv
9	<i>remove cast</i>	sat	sat	sat	sat	sat	sat	tv
10	<i>examine patient</i>	sat	sat	sat	sat	sat	sat	tv

(‘sat’=*satisfied* and ‘tv’=*temporarily violated*)

Table 4.5 shows that there is a relation between the state of the instance and states of its mandatory constraints and that optional constraint c_6 has no influence on the state of the instance. This is because the state of the instance depends only on the sets of satisfying traces of all mandatory constraints (cf. Definition 4.4.2). In some cases the instance state can be determined based on the states of mandatory constraints: (1) all mandatory constraints are *satisfied* in a *satisfied* instance, and (2) if at least one of the mandatory constraints in an instance is *violated*, then the instance is also *violated*. In all other cases, as we will show later, the instance state cannot always be derived from the states of individual mandatory constraints.

As specified in Definition 4.2.2, the set of traces that satisfy a constraint model is composed of traces that satisfy all constraints in the model. *Compositionality* in this context means that it is enough to prove that a trace satisfies each of the constraints in order to prove that the trace satisfies the model [73, 205].

Property 4.4.6 shows that, if an instance trace satisfies each of the mandatory constraints in the instance, then this trace also satisfies the instance and vice versa, i.e., if a trace satisfies an instance, then it also satisfies all mandatory constraints in the instance.

Property 4.4.6. (All mandatory constraints are satisfied in a satisfied instance)

Let $ci \in \mathcal{U}_{ci}$ be an instance where $ci = (\sigma, (A, C_M, C_O))$. Then $(\forall c \in C_M : \nu(\sigma, c) = \text{satisfied}) \Leftrightarrow (\omega(ci) = \text{satisfied})$.

Proof. $(\forall c \in C_M : \nu(\sigma, c) = \text{satisfied}) \Leftrightarrow (\forall c \in C_M : \sigma \in \mathcal{E}_{\text{Fc}}^*) \Leftrightarrow (\sigma \in \bigcap_{c \in C_M} \mathcal{E}_{\text{Fc}}^*) \Leftrightarrow (\sigma \in \mathcal{E}_{\text{Fcm}}^*) \Leftrightarrow (\omega(ci) = \text{satisfied})$. \square

Property 4.4.7 shows that, if at least one of the mandatory constraints is *violated*, then the instance is also *violated*. To prove this, Property 4.4.7 shows that, if the instance trace does not and cannot (in the future) satisfy (at least) one mandatory constraint in the instance, then this trace also does not and cannot (in the future) satisfy the instance.

Property 4.4.7. (If at least one mandatory constraint in an instance is violated then the instance is violated)

Let $ci \in \mathcal{U}_{ci}$ be an instance where $ci = (\sigma, (A, C_M, C_O))$. Then $(\exists c \in C_M : \nu(\sigma, c) = \text{violated}) \Rightarrow (\omega(ci) = \text{violated})$.

Proof. $(\exists c \in C_M : \nu(\sigma, c) = \text{violated}) \Rightarrow (\exists c \in C_M : (\sigma \notin \mathcal{E}_{\text{Fc}}^* \wedge \nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\text{Fc}}^*)) \Rightarrow ((\sigma \notin \bigcap_{c \in C_M} \mathcal{E}_{\text{Fc}}^*) \wedge (\nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \bigcap_{c \in C_M} \mathcal{E}_{\text{Fc}}^*)) \Rightarrow ((\sigma \notin \mathcal{E}_{\text{Fcm}}^*) \wedge (\nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\text{Fcm}}^*)) \Rightarrow \omega(ci) = \text{violated}$. \square

Note that it might be the case that, although none of the mandatory constraints in an instance is *violated* but some are *temporarily violated*, the instance itself is *violated*. Example 4.4.8 shows how an instance can be *violated* although none of its mandatory constraints is *violated*.

Example 4.4.8. (A violated instance without violated constraints)

Recall that $t_s, t_c \in \mathcal{T}$ are two event types such that $t_s = \text{started}$ and $t_c = \text{completed}$. Let $ci \in \mathcal{U}_{ci}$, $ci = (\sigma, cm)$ be a constraint instance where $cm = (A, C_M, C_O)$ such that:

- $A = \{a_1, a_2\}$ is a set of two activities $a_1, a_2 \in \mathcal{A}$,
- $C_M = \{c_1, c_2\}$ is a set of two mandatory constraints $c_1, c_2 \in \mathcal{C}$ such that
 - $c_1 = (\{(a_1, t_c)\}, \text{“}a_1 \text{ has to be completed exactly once”})$;
 - $c_2 = (\{(a_1, t_c)\}, \text{“}a_1 \text{ has to be completed exactly twice”})$;
- $C_O = \emptyset$ is an (empty) set of optional constraints, and
- $\sigma = \langle (a_1, t_s), (a_1, t_c) \rangle$ is the execution trace of the instance ci .

\square

States of constraints in instance ci from Example 4.4.8 (given the instance execution trace σ) are:

- $\nu(\sigma, c_1) = \textit{satisfied}$ because $\sigma \in \mathcal{E}_{\models c_1}^*$, and
- $\nu(\sigma, c_2) = \textit{temporarily violated}$ because it can be satisfied by executing a_1 once more.

Constraint c_1 is *satisfied* because trace σ satisfies c_1 and constraint c_2 is *temporarily violated* because trace σ does not satisfy c_2 but there is a suffix (e.g., suffix $\langle (a_1, t_s), (a_1, t_c) \rangle$) that, when added to σ , satisfies c_2 . Although c_1 is *satisfied* and c_2 is *temporarily violated*, there is no suffix that, when added to σ , satisfies both constraints c_1 and c_2 , i.e., it is not possible to execute activity a_1 exactly once and twice at the same time. This means that the instance is *violated*, i.e., $(\sigma \notin \mathcal{E}_{\models cm}^* \wedge (\nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*)) \Rightarrow \omega(ci) = \textit{violated}$.

The instance presented in Example 4.4.8 cannot be *satisfied* because the instance model contains an error. Two constraints that can never be satisfied at the same time cause this error, i.e., there is no trace that satisfies both constraints. This type of error can occur in constraint models and is called *conflict*. Verification of constraint models and how to detect errors like conflicts will be described in Section 4.6.

The formal language for constraint specification that will be presented in Chapter 5 enables execution of constraint instances because it is able to (1) determine state of an instance (cf. Definition 4.4.2) and (2) determine which events are enabled (cf. Definition 4.4.3). Moreover, it is possible to monitor states of each constraint from the model. Because the prototype DECLARE uses this language for constraint specification, it enables execution of constraint models, as will be described in Chapter 6.

4.5 Ad-hoc Instance Change

In some cases, it is necessary that the model of the instance changes (i.e., add and remove activities, mandatory and optional constraints) although the instance is already executing and the instance trace might not be empty. We refer to such a change as to an *ad-hoc instance change* or *instance change*. Workflow management systems that support ad-hoc change are called adaptive systems. For example, ADEPT [189, 191–193, 202] is a workflow management system that uses powerful mechanisms to support ad-hoc change of procedural process models by allowing adding activities to the control-flow, removing activities from the control-flow and moving activities in activities to the control-flow at run-time (cf. Section 2.2). On the one hand, as we discussed in Section 2.1.5, various problems can occur when it comes to ad-hoc change of imperative process models (e.g., the “dynamic change bug” [101] and other problems described in [201]). On the other hand, the constraint-based approach offers a simple method for ad-hoc

change that is based in a single requirement: the instance should not become *violated* after the change.

An instance of a constraint model refers to one execution of this model, i.e., the instance assigns one execution trace to the model. Due to the fact that, in ad-hoc change the trace remains the same and the model changes, it might happen that the instance state changes according to the new model (cf. Definition 4.4.2). For example, it is possible that a *satisfied* instance would become *violated* if a mandatory constraint would be added to the instance, which is an undesired state of constraint instances (cf. Definition 4.4.4). Therefore, instance change can be applied (i.e., is successful) if and only if the change does not bring the instance into the *violated* state. After a successful change, the instance continues execution with the original trace. In Definition 4.5.1 ad-hoc instance change is defined as a function Δ that changes the instance model and assigns the changed model to the instance without changing the instance trace if and only this does not bring the instance in the state *violated*.

Definition 4.5.1. (Ad-hoc instance change Δ)

Let $\Delta : \mathcal{U}_{ci} \times \mathcal{U}_{cm} \rightarrow \mathcal{U}_{ci}$ be a partial function with domain $dom(\Delta) = \{((\sigma, cm), cm') \in \mathcal{U}_{ci} \times \mathcal{U}_{cm} \mid \omega((\sigma, cm')) \neq \textit{violated}\}$. For all $((\sigma, cm), cm') \in dom(\Delta)$ it holds that $\Delta((\sigma, cm), cm') = (\sigma, cm')$. \square

Figure 4.4 shows examples of a successful and unsuccessful ad-hoc instance change. Originally, there are four instances in Figure 4.4(a): instances $ci_1 = (\sigma_1, cm_1)$, $ci_2 = (\sigma_2, cm_1)$, $ci_3 = (\sigma_3, cm_3)$, and $ci_4 = (\sigma_4, cm_4)$. The state of every instance is represented by a special line: instances ci_1 , ci_2 and ci_4 are *satisfied* and instance ci_3 is *temporarily violated*. Figure 4.4(b) shows that it is possible to change the model of instance ci_2 to model cm_2 , because $\omega((\sigma_2, cm_2)) = \textit{temporarily violated}$. On the other hand, Figure 4.4(c) shows that it is *not* possible to change the model of instance ci_2 to model cm_3 (indicated with the \emptyset symbol on the arrow), because $\omega((\sigma_2, cm_3)) = \textit{violated}$.

Consider, for example an instance $ci = (\sigma, cm^{FT})$ of the Fractures Treatment model cm^{FT} from Example 4.3.2 where the patient was first *examined*, then the *sling* was prescribed and finally *rehabilitation* was prescribed. The state of this instance is $\omega(ci) = \textit{satisfied}$ because the trace satisfies all mandatory constraints, i.e., $\sigma \in \mathcal{E}_{\models cm^{FT}}^*$. Assume now that users want to add a mandatory constraint $c \in \mathcal{C}$ to instance ci where $c = (E, f)$ such that $E = \{(prescribe\ sling, t_c), (perform\ X\ ray, t_c)\}$ and $f = \text{“Can not complete prescribe sling before completing perform X ray”}$. This cannot be done because adding c as a mandatory constraint to the instance would bring the instance into state *violated*, i.e., the *sling* was already prescribed before the *perform X ray* was completed. On the other hand, adding activity $consult\ external \in \mathcal{A}$ to the instance is allowed because it leaves the instance in the *satisfied* state.

Ad-hoc adding or removing activities and optional constraints to a *satisfied* or *temporarily violated* instance will always be successful, i.e. this change always

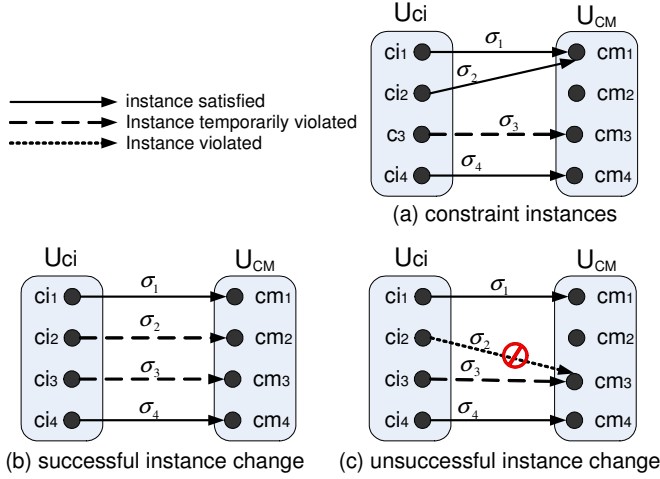


Figure 4.4: Instance change

results in a changed instance, as shown in Property 4.5.2. This is because these types of model changes do not influence the set of traces that satisfy the model (cf. Property 4.2.4), and, therefore, to not change the state of the instance (cf. Definition 4.4.2). On the one hand, the only effect that adding an activity to an instance model has it that this activity becomes available for execution (cf. Definition 4.4.4). On the other hand, after an activity has been removed from an instance, its users can no longer execute events involving this activity but the activity might still be in the instance trace if it was executed prior to the change.

Property 4.5.2. (Ad-hoc adding and removing activities and optional constraints cannot cause failure of Δ)

Let $ci \in \mathcal{U}_{ci}$ be a constraint instance such that $\omega(ci) \neq \text{violated}$ where $ci = (\sigma, (A, C_M, C_O))$ and let $cm' \in \mathcal{U}_{cm}$ be a constraint model where $cm' = (A', C_M, C'_O)$, then it holds that $\Delta(ci, cm') = (\sigma, cm')$, i.e., the change is successful.

Proof. It holds that $\mathcal{E}_{\models cm}^* = \mathcal{E}_{\models cm'}^*$ (cf. Property 4.2.4). Therefore, it also holds that $\omega(ci) = \omega(\sigma, cm')$ (cf. Definition 4.4.2) and $(\omega(ci) \neq \text{violated}) \Rightarrow (\omega(\sigma, cm') \neq \text{violated})$, i.e. $(ci, cm') \in \text{dom}(\Delta)$ and $\Delta(ci, cm') = (\sigma, cm')$. \square

Since mandatory constraints influence the set of traces that satisfy the model (cf. Property 4.2.5) and state of an instance (cf. Definition 4.4.2), adding and removing these constraints can have an effect on the ad-hoc instance change. A mandatory constraint can always be successfully removed from a *satisfied* or *temporarily violated* instance because this type of ad-hoc change will never cause a *violated* state of the instance, as shown in Property 4.5.3.

Property 4.5.3. (Ad-hoc removing mandatory constraints cannot cause failure of Δ)

Let $ci \in \mathcal{U}_{ci}$ be a constraint instance such that $\omega(ci) \neq \text{violated}$ where $ci = (\sigma, (A, C_M, C_O))$ and let $cm' \in \mathcal{U}_{cm}$ be a constraint model where $cm' = (A, C'_M, C_O)$ such that $C'_M \subset C_M$, then it holds that $\Delta(ci, cm') = (\sigma, cm')$, i.e., the change is successful.

Proof. Because $\omega((\sigma, cm)) \neq \text{violated}$ it holds that $(\sigma \in \mathcal{E}_{\models cm}^*) \vee (\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*)$ (cf. Definition 4.4.2). Because it holds that $\mathcal{E}_{\models cm}^* \subseteq \mathcal{E}_{\models cm'}^*$ (cf. Property 4.2.5), it also holds that $((\sigma \in \mathcal{E}_{\models cm}^*) \vee (\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*)) \Rightarrow ((\sigma \in \mathcal{E}_{\models cm'}^*) \vee (\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm'}^*))$, i.e., it holds that $\omega(cm') \neq \text{violated}$. Therefore, it holds that $(ci, cm') \in \text{dom}(\Delta)$ and $\Delta(ci, cm') = (\sigma, cm')$. \square

As we showed in this section, the only requirement of ad-hoc change in constraint models is that the instance does not become *violated* after the change. In Chapter 5 we will show a formal language for constraint specification that offers a simple method for determining the state of a changed instance. Using this language, the DECLARE prototype supports ad-hoc change by accepting it if the new state of the instance is not *violated*. If it is, the prototype reports this error and the instance continues its execution based on the original model. The ad-hoc change in DECLARE will be described in Chapter 6.

4.6 Verification of Constraint Models

Verification of process models enables automated detection of possible errors in models [89]. Many verification techniques for identifying possible errors in procedural process models aim at detecting syntactical errors (e.g., deadlck and livelock) [221] and semantical errors (e.g., ‘can a process always lead to an acceptable final state’) [89]. On the one hand, verification of models specified in languages with a formal mathematical definition (e.g., Petri nets [87,177,199]) classifies the model as a *correct* or *incorrect* [19,31,83,89,124,254,255]. On the other hand, various verification techniques have been proposed for informal process modeling languages that use graph reduction [30,86,161,170,177,222,223,268] or translation of an informal model to a formal one, which is then verified [17,82,88,157,170].

Just like the procedural ones, constraint models can also contain errors. In this section we describe verification of constraint models. While executing an instance of a constraint model $cm = (A, C_M, C_O)$ users execute activities $a \in A$ by triggering events $e \in A \times \mathcal{T}$ on these activities (cf. the execution rule in Definition 4.4.4). During the execution the instance can change state depending on the events that were triggered. At the end of the execution, an instance must be *satisfied*, in order to satisfy all mandatory constraints (cf. Section 4.4.1). Certain combinations of constraints can cause two types of errors in constraint models that affect later executions of model instances: (1) *dead events* and (2)

conflicts. An event is dead in a model if none of the traces that satisfy the model contains this event, i.e., a dead event cannot be executed in any instance of the model. A model contains conflicts if there is no trace that satisfies the model, i.e., instances of the model can never be *satisfied*.

4.6.1 Dead Events

An event is dead in a model if none of the traces that satisfy the model contains this event, as specified in Definition 4.6.1. In other words, if a dead event would be triggered and added to the execution trace of a model instance, the instance would become *violated*. For example, it is possible that, due to the combination of two mandatory constraints in model $cm = (A, C_M, C_O)$, event $(a, started)$ (where $a \in A$) is dead in the model, i.e., none of the traces that satisfy the model contains event $(a, started)$. The consequence of event $(a, started)$ being dead is that, during the execution of instances of this model, event $(a, started)$ will never be enabled (cf. Definition 4.4.3) and it will never be possible to start (and execute) activity a from the model because adding this event to the instance trace would cause the instance to become *violated* (cf. Definition 4.4.4). Although it will be possible that users execute instances of the model in a way that instances are *satisfied* by never executing activity a , it is important to verify the model against this error and to issue a warning that, although activity a is in the model, it will never be possible to execute it.

Definition 4.6.1. (Dead event)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model. Event $e \in \mathcal{E}$ is a *dead* in model cm , if and only if $\nexists \sigma \in \mathcal{E}_{\models cm}^* : e \in \sigma$. The set of dead events of model cm is defined as $\Pi_{DE}(cm) = \{e \in \mathcal{E} \mid \nexists \sigma \in \mathcal{E}_{\models cm}^* : e \in \sigma\}$ \square

The composition of all mandatory constraints in the model determines which traces satisfy the model (cf. Definition 4.2.2). Therefore, a combination of mandatory constraints may cause an event to be dead. The smallest subset of mandatory constraints for which an event is dead is called the *cause of the dead event*.

Definition 4.6.2. (Cause of dead event)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model where $cm = (A, C_M, C_O)$ and $e \in \Pi_{DE}(cm)$ be a dead event in cm . The set of constraints $C \subseteq C_M$ is a cause of the dead event e if and only if it holds that $(e \in \Pi_{DE}((A, C, C_O))) \wedge (\forall C' \subset C : e \notin \Pi_{DE}((A, C', C_O)))$.

We use $C_e^{dead}(cm)$ to denote the set of all causes of dead event e in model cm , i.e., $C_e^{dead}(cm) = \{C \subseteq C_M \mid (e \in \Pi_{DE}((A, C, C_O))) \wedge (\forall C' \subset C : e \notin \Pi_{DE}((A, C', C_O)))\}$ \square

Consider the constraint model cm^R from Example 4.6.3 containing four activities and three constraints. Event $(curse, completed)$ is dead in constraint

model cm^R . Due to the constraint c_2 all traces that satisfy model cm^R must contain event (*become holy, completed*). On the other hand, due to constraint c_3 all traces that satisfy model cm^R and contain event (*become holy, completed*) cannot contain event (*curse, completed*). Therefore, due to the combination of constraints c_2 and c_3 event (*curse, completed*) is a dead event in this model, i.e., $(curse, completed) \in \Pi_{DE}(cm^R)$ and constraints $\{c_2, c_3\}$ are the only cause of this dead event (i.e., $C_e^{dead}(cm) = \{\{c_2, c_3\}\}$).

Example 4.6.3. (A constraint model with a dead event)

Recall that $t_c \in \mathcal{T}$ is an event type where $t_c = completed$. Let $cm^R \in \mathcal{U}_{cm}$, $cm^R = (A^R, C_M^R, C_O^R)$ be a constraint model such that:

- $A^R = \{pray, curse, bless, become\ holy\}$ is a set of activities,
- $C_M^R = \{c_1, c_2, c_3\}$ is a set of mandatory constraints where
 - $c_1 = (E_1, f_1)$ such that $E_1 = \{(curse, t_c), (pray, t_c)\}$ and $f_1 =$ “Must complete at least one occurrence of activity *pray* after every occurrence of activity *curse*”,
 - $c_2 = (E_2, f_2)$ such that $E_2 = \{(become\ holy, t_c)\}$ and $f_2 =$ “Must complete at least one occurrence of activity *become holy*”,
 - $c_3 = (E_3, f_3)$ such that $E_3 = \{(curse, t_c), (become\ holy, t_c)\}$ and $f_3 =$ “If *completed* an occurrence of activity *curse* then cannot complete any occurrence of activity *become holy* and if *completed* an occurrence of activity *become holy* then cannot complete any occurrence of activity *curse*”, and
- $C_O^R = \emptyset$ is a (empty) set of optional constraints.

□

If there is a dead event in a model, then there must be at least one cause of the dead event, i.e., then must be at least one combination of mandatory constraints that causes this event to be dead. A dead event can be removed from a model if all sets that cause the dead event are removed from the model. In other words, it is necessary to remove at least one constraint from each of the sets that cause the dead event in order for the dead event to become ‘alive’ again.

Property 4.6.4. (A dead event is removed from a constraint model if and only if at least one mandatory constraint is removed from each of the constraint sets that cause the dead event)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model where $cm = (A, C_M, C_O)$ and $e \in \mathcal{E}$ an event such that $e \in \Pi_{DE}(cm)$. Let $cm' \in \mathcal{U}_{cm}$ be a constraint model where $cm' = (A', C'_M, C'_O)$ such that $C'_M \subset C_M$, then it holds that $(\forall C \in C_e^{dead}(cm) : C \not\subseteq C'_M) \Leftrightarrow e \notin \Pi_{DE}(cm')$.

Proof. First, we prove that it holds that $(\forall C \in C_e^{dead}(cm) : C \not\subseteq C'_M) \Rightarrow e \notin \Pi_{DE}(cm')$. We will prove that this holds by showing that it does not hold that

$(\forall C \in C_e^{dead}(cm) : C \not\subseteq C'_M) \Rightarrow e \in \Pi_{DE}(cm')$. If it holds that $e \in \Pi_{DE}(cm')$, then it also holds that $\forall C \in C_e^{dead}(cm') : C \subseteq C'_M$. Because it holds that $\forall C \in C_e^{dead}(cm') : C \subseteq C'_M$ (cf. Definition 4.6.2) and $C'_M \subset C_M$, it also holds that $\forall C \in C_e^{dead}(cm') : C \subset C_M$ and, therefore, it holds that $\forall C \in C_e^{dead}(cm') : C \in C_e^{dead}(cm)$. In other words, if there is a cause $C \subseteq C'_M$ of dead event e in model cm' , then this cause also exists in the original model cm , i.e., $C \subseteq C_M$. Therefore, it holds that $\exists C \in C_e^{dead}(cm) : C \subseteq C'_M$, which is in contradiction with the statement $\forall C \in C_e^{dead}(cm) : C \not\subseteq C'_M$, and it does not hold that $(\forall C \in C_e^{dead}(cm) : C \not\subseteq C'_M) \Rightarrow e \in \Pi_{DE}(cm')$.

Second, we prove that it holds that $(\exists C \in C_e^{dead}(cm) : C \subseteq C'_M) \Rightarrow e \in \Pi_{DE}(cm')$. If it holds that $\exists C \in C_e^{dead}(cm) : C \subseteq C'_M$, then it holds that $\exists C \subseteq C'_M : e \in \Pi_{DE}(A', C, C'_O)$, i.e., it holds that $\exists C \subseteq C'_M : \forall \sigma \in \mathcal{E}_{\models A', C, C'_O}^* : e \notin \sigma$. Because $C \subseteq C'_M$ it holds that $\mathcal{E}_{\models (A', C'_M, C'_O)}^* \subseteq \mathcal{E}_{\models A', C, C'_O}^*$ (cf. Property 4.2.5) and, therefore, it holds that $\forall \sigma \in \mathcal{E}_{\models A', C'_M, C'_O}^* : e \notin \sigma$, i.e., it holds that e is dead in cm' (i.e., $e \in \Pi_{DE}(cm')$). \square

A formal language for constraint specification will be presented in Chapter 5. This LTL-based language enables a simple method for verification of constraint models against dead events. This method is used in the DECLARE prototype (that will be presented in Chapter 6) to verify constraint models against dead events and detect so-called *dead activities*. When verifying a model $cm = (A, C_M, C_O)$, for each model activity $a \in A$ DECLARE tests if events $(a, started)$ and $(a, completed)$ are dead. If at least one of these events is dead, the dead activity verification error is reported and the smallest set(s) that cause it. In this way, dead events can be easily detected and eliminated in DECLARE by removing at least one constraint from each of the sets of constraints that cause this error.

4.6.2 Conflicts

If there exists no trace that satisfies a constraint model, then this model has a conflict. Unlike models with dead activities, models with conflicts are not executable because they can never be satisfied by any trace, i.e., instances of models with conflicts are always *violated*.

Definition 4.6.5. (Conflict)

Model $cm \in \mathcal{U}_{cm}$ has a *conflict* if and only if $\mathcal{E}_{\models cm}^* = \emptyset$. \square

The composition of all mandatory constraints in a model determines which traces satisfy the model (cf. Definition 4.2.2). A certain combination of mandatory constraints may cause a conflict in the model. The smallest subset of mandatory constraints that contains a conflict is called the cause of conflict.

Definition 4.6.6. (Cause of conflict)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model with a conflict where $cm = (A, C_M, C_O)$.

The set of constraints $C \subseteq C_M$ is a cause of the conflict if and only if it holds that $(\mathcal{E}_{\models(A,C,C_O)}^* = \emptyset) \wedge (\forall C' \subset C : \mathcal{E}_{\models(A,C',C_O)}^* \neq \emptyset)$.

We use $C^{conf}(cm)$ to denote the set of all causes of conflict in model cm , i.e., $C^{conf}(cm) = \{C \subseteq C_M \mid (\mathcal{E}_{\models(A,C,C_O)}^* = \emptyset) \wedge (\forall C' \subset C : \mathcal{E}_{\models(A,C',C_O)}^* \neq \emptyset)\}$ \square

Constraint model cm^R from Example 4.6.7 has a conflict. Due to constraints c_2 and c_4 all traces that satisfy model cm^R must contain events (*become holy, completed*) and (*curse, completed*), respectively. On the other hand, due to constraint c_3 all traces that satisfy model cm^R must not contain both events (*become holy, completed*) and (*curse, completed*). Therefore, due to the combination of constraints c_2 , c_3 and c_4 this model has a conflict, i.e., $\mathcal{E}_{\models cm^R}^* = \emptyset$ and constraints $\{c_2, c_3, c_4\}$ are the cause of this conflict ($C^{conf}(cm^R) = \{\{c_2, c_3, c_4\}\}$).

Example 4.6.7. (A constraint model with a conflict)

Recall that $t_c \in \mathcal{T}$ is an event type where $t_c = \textit{completed}$ ⁴. Let $cm^R \in \mathcal{U}_{cm}$, $cm^R = (A^R, C_M^R, C_O^R)$ be a constraint model such that:

- $A^R = \{\textit{pray, curse, bless, become holy}\}$ is a set of activities,
- $C_M^R = \{c_1, c_2, c_3, c_4\}$ is a set of mandatory constraints where:
 - $c_1 = (E_1, f_1)$ such that $E_1 = \{(\textit{curse}, t_c), (\textit{pray}, t_c)\}$ and $f_1 =$ “Must *complete* at least one occurrence of activity *pray* after every occurrence of activity *curse*”,
 - $c_2 = (E_2, f_2)$ such that $E_2 = \{(\textit{become holy}, t_c)\}$ and $f_2 =$ “Must *complete* at least one occurrence of activity *become holy*”,
 - $c_3 = (E_3, f_3)$ such that $E_3 = \{(\textit{curse}, t_c), (\textit{become holy}, t_c)\}$ and $f_3 =$ “If *completed* an occurrence of activity *curse* then cannot *complete* any occurrence of activity *become holy* and if *completed* an occurrence of activity *become holy* then cannot *complete* any occurrence of activity *curse*”,
 - $c_4 = (E_4, f_4)$ such that $E_4 = \{(\textit{curse}, t_c)\}$ and $f_4 =$ “Must *complete* at least one occurrence of activity *curse*”, and
- $C_O^R = \emptyset$ is a (empty) set of optional constraints.

\square

If a model has a conflict, then there must be at least one cause of the conflict, i.e., then must be at least one combination of mandatory constraints that cause the conflict. A conflict can be removed from a model if all sets that cause the conflict are removed from the model, i.e., it is necessary to remove at least one constraint from each of the sets that cause the conflict in order to remove the conflict from the model.

⁴For the simplicity we will assume in this example that an activity is successfully executed when event of the type *completed* was triggered on this activity.

Property 4.6.8. (A conflict is removed from a constraint model if and only if at least one mandatory constraint is removed from each of the constraint sets that cause the conflict)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model where $cm = (A, C_M, C_O)$ such that $\mathcal{E}_{\models cm}^* = \emptyset$. Let $cm' \in \mathcal{U}_{cm}$ be a constraint model where $cm' = (A', C'_M, C'_O)$ such that $C'_M \subset C_M$, then it holds that $(\forall C \in C^{conf}(cm) : C \not\subseteq C'_M) \Leftrightarrow \mathcal{E}_{\models cm'}^* \neq \emptyset$.

Proof. First, we prove that it holds that $(\forall C \in C^{conf}(cm) : C \not\subseteq C'_M) \Rightarrow \mathcal{E}_{\models cm'}^* \neq \emptyset$. We will prove that it holds that $\mathcal{E}_{\models cm'}^* \neq \emptyset$ by showing that it does not hold that $(\forall C \in C^{conf}(cm) : C \not\subseteq C'_M) \Rightarrow \mathcal{E}_{\models cm'}^* = \emptyset$. If it holds that $\mathcal{E}_{\models cm'}^* = \emptyset$, then it also holds that $\forall C \in C^{conf}(cm') : C \subseteq C'_M$. Because it holds that $\forall C \in C^{conf}(cm') : C \subseteq C'_M$ (cf. Definition 4.6.6) and $C'_M \subset C_M$, it also holds that $\forall C \in C^{conf}(cm') : C \subset C_M$ and, therefore, it holds that $\forall C \in C^{conf}(cm') : C \in C^{conf}(cm)$. In other words, if there is a cause $C \subseteq C'_M$ of dead event e in model cm' , then this cause also exists in the original model cm , i.e., $C \subseteq C_M$. Therefore, it holds that $\exists C \in C^{conf}(cm) : C \subseteq C'_M$, which is in contradiction with the statement $\forall C \in C^{conf}(cm) : C \not\subseteq C'_M$, and it does not hold that $(\forall C \in C^{conf}(cm) : C \not\subseteq C'_M) \Rightarrow \mathcal{E}_{\models cm'}^* = \emptyset$.

Second, we prove that it holds that $(\exists C \in C^{conf}(cm) : C \subseteq C'_M) \Rightarrow \mathcal{E}_{\models cm'}^* = \emptyset$. If it holds that $\exists C \in C^{conf}(cm) : C \subseteq C'_M$, then it holds that $\exists C \subseteq C'_M : \mathcal{E}_{\models A', C, C'_O}^* = \emptyset$. Because $C \subseteq C'_M$ it holds that $\mathcal{E}_{\models (A', C'_M, C'_O)}^* \subseteq \mathcal{E}_{\models A', C, C'_O}^*$ (cf. Property 4.2.5) and, therefore, it holds that $\mathcal{E}_{\models cm'}^* = \emptyset$. \square

If a model contains a conflict, then its set of satisfying traces is empty and therefore all events are dead in the model.

Property 4.6.9. (All events are dead in a model with conflict)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model such that $\mathcal{E}_{\models cm}^* = \emptyset$, then it holds that $\Pi_{DE}(cm) = \mathcal{E}$.

Proof. If it holds that $\mathcal{E}_{\models cm}^* = \emptyset$, then $\forall e \in \mathcal{E} : \nexists \sigma \in \mathcal{E}_{\models cm}^* : e \in \sigma$, i.e., $\Pi_{DE}(cm) = \mathcal{E}$. \square

A constraint model $cm \in \mathcal{U}_{cm}$ does not necessarily have a conflict even if all events are dead in cm because, depending on the model, the empty trace could satisfy the model.

In Chapter 5 we will show a formal language for constraint specification with a simple method for detecting conflicts. This method is also used in the DECLARE prototype to verify constraint models against conflicts and detect the smallest set(s) that cause this error, which enables easy detection and elimination of conflicts.

4.6.3 Compatibility of Models

In some cases it is necessary to check if two or more processes are compatible with each other. This can be checked by first merging the process (constraint)

models and then verifying the merged model for errors. Constraint models can be merged together to create a new constraint model containing all activities and constraints from the two or more original models.

Definition 4.6.10. (Merging constraint models \oplus)

Let $cm^1, cm^2 \dots cm^n \in \mathcal{U}_{cm}$ be n constraint models where $\forall_{1 \leq i \leq n} cm^i = (A^i, C_M^i, C_O^i)$. The merged model $cm^1, cm^2 \dots cm^n$ is a constraint model $cm' = cm^1 \oplus cm^2 \oplus \dots \oplus cm^n$ where $cm' = (A', C'_M, C'_O)$ such that $A' = A^1 \cup A^2 \cup \dots \cup A^n$, $C'_M = C_M^1 \cup C_M^2 \cup \dots \cup C_M^n$ and $C'_O = C_O^1 \cup C_O^2 \cup \dots \cup C_O^n$. Note that by definition $\mathcal{E}_{\models cm'}^* = \mathcal{E}_{\models cm^1}^* \cap \mathcal{E}_{\models cm^2}^* \cap \dots \cap \mathcal{E}_{\models cm^n}^*$. \square

Example 4.6.11 shows three constraint models cm^1 , cm^2 and cm^3 and how these models can be merged into $cm^1 \oplus cm^2$, $cm^1 \oplus cm^3$, $cm^2 \oplus cm^3$ and $cm^1 \oplus cm^2 \oplus cm^3$.

Example 4.6.11. (Merging three constraint models)

Recall that $t_c \in \mathcal{T}$ is an event type where $t_c = completed$. Let $cm^1, cm^2, cm^3 \in \mathcal{U}_{cm}$ be three constraint models where:

- $cm^1 = (A^1, C_M^1, C_O^1)$ such that:
 - $A = \{pray, curse, bless, become\ holy}\}$ is a set of activities.
 - $C_M = \{c_1, c_2\}$ is a set of mandatory constraints where
 - * $c_1 = (E_1, f_1)$ such that $E_1 = \{(curse, t_c), (pray, t_c)\}$ and $f_1 =$ “Must *complete* at least one occurrence of activity *pray* after every *completed* occurrence of activity *curse*”, and
 - * $c_2 = (E_2, f_2)$ such that $E_2 = \{(become\ holy, t_c)\}$ and $f_2 =$ “Must *complete* at least one occurrence of activity *become holy*”, and
 - $C_O = \emptyset$ is a (empty) set of optional constraints.
- $cm^2 = (A^2, C_M^2, C_O^2)$ such that:
 - $A^2 = \{curse, become\ holy}\}$ is a set of activities,
 - $C_M^3 = \{c_3\}$ is a set of mandatory constraints where $c_3 = (E_3, f_3)$ such that $E_3 = \{(curse, t_c), (become\ holy, t_c)\}$ and $f_3 =$ “If *completed* occurrence of activity *curse* then cannot *complete* any occurrence of activity *become holy* and if *completed* occurrence of activity *become holy* then cannot *complete* any occurrence of activity *curse*”, and
 - $C_O^2 = \emptyset$ is a (empty) set of optional constraints.
- $cm^3 = (A^3, C_M^3, C_O^3)$ such that:
 - $A^3 = \{curse\}$ is a set of activities,
 - $C_M^2 = \{c_4\}$ is a set of mandatory constraints where $c_4 = (E_4, f_4)$ such that $E_4 = \{(curse, t_c)\}$ and $f_4 =$ “Must *complete* at least one occurrence of activity *curse*”, and
 - $C_O^2 = \emptyset$ is a (empty) set of optional constraints.

Merging models cm^1 and cm^2 yields the constraint model $cm^{1\oplus 2} = cm^1 \oplus cm^2$ where $cm^{1\oplus 2} = (A^{1\oplus 2}, C_M^{1\oplus 2}, C_O^{1\oplus 2})$ such that:

- $A^{1\oplus 2} = A^1 \cup A^2 = \{\text{pray, curse, bless, become holy}\},$
- $C_M^{1\oplus 2} = C_M^1 \cup C_M^2 = \{c_1, c_2, c_3\},$ and
- $C_O^{1\oplus 2} = C_O^1 \cup C_O^2 = \emptyset.$

Merging models cm^1 and cm^3 yields the constraint model $cm^{1\oplus 3} = cm^1 \oplus cm^3$ where $cm^{1\oplus 3} = (A^{1\oplus 3}, C_M^{1\oplus 3}, C_O^{1\oplus 3})$ such that:

- $A^{1\oplus 3} = A^1 \cup A^3 = \{\text{pray, curse, bless, become holy}\},$
- $C_M^{1\oplus 3} = C_M^1 \cup C_M^3 = \{c_1, c_2, c_4\},$ and
- $C_O^{1\oplus 3} = C_O^1 \cup C_O^3 = \emptyset.$

Merging models cm^2 and cm^3 yields the constraint model $cm^{2\oplus 3} = cm^2 \oplus cm^3$ where $cm^{2\oplus 3} = (A^{2\oplus 3}, C_M^{2\oplus 3}, C_O^{2\oplus 3})$ such that:

- $A^{2\oplus 3} = A^2 \cup A^3 = \{\text{curse, become holy}\},$
- $C_M^{2\oplus 3} = C_M^2 \cup C_M^3 = \{c_3, c_4\},$ and
- $C_O^{2\oplus 3} = C_O^2 \cup C_O^3 = \emptyset.$

Merging models cm^1 , cm^2 and cm^3 yields the constraint models $cm^{1\oplus 2\oplus 3} = cm^1 \oplus cm^2 \oplus cm^3$ where $cm^{1\oplus 2\oplus 3} = (A^{1\oplus 2\oplus 3}, C_M^{1\oplus 2\oplus 3}, C_O^{1\oplus 2\oplus 3})$ such that:

- $A^{1\oplus 2\oplus 3} = A^1 \cup A^2 \cup A^3 = \{\text{pray, curse, bless, become holy}\},$
- $C_M^{1\oplus 2\oplus 3} = C_M^1 \cup C_M^2 \cup C_M^3 = \{c_1, c_2, c_3, c_4\},$ and
- $C_O^{1\oplus 2\oplus 3} = C_O^1 \cup C_O^2 \cup C_O^3 = \emptyset.$

□

Verification of a merged model can detect that the merging causes some events to be dead. This type of incompatibility suggests that, even though the event can be executed in individual models, if the two models (i.e., processes) would be merged, the event would be dead in the resulting model.

Definition 4.6.12. (Dead event incompatibility)

Let $cm^1, cm^2 \dots cm^n \in \mathcal{U}_{cm}$ be n constraint models and $e \in \mathcal{E}$ be an event that is not dead in these models, i.e. $\forall_{1 \leq i \leq n} e \notin \Pi_{DE}(cm^i)$. If $e \in \Pi_{DE}(cm^1 \oplus cm^2 \oplus \dots \oplus cm^n)$, then we say that e is dead due to the incompatibility of models cm^1, cm^2, \dots, cm^n . □

Consider the original models cm^1 , cm^2 and cm^3 from Example 4.6.11. There are no dead events in the original models, i.e., $\forall_{1 \leq i \leq 3} \Pi_{DE}(cm^i) = \emptyset$. However, event $(\text{curse, completed})$ is dead in the merged model $cm^{1\oplus 2} = cm^1 \oplus cm^2$, i.e., $\Pi_{DE}(cm^{1\oplus 2}) = \{(\text{curse, completed})\}$ and the cause of this dead event are constraints $C_e^{dead}(cm^{1\oplus 2}) = \{\{c^2, c^3\}\}$ (note that the model $cm^{1\oplus 2}$ is the same like the model from Example 4.6.3). Therefore, event $(\text{curse, completed})$ is dead due to incompatibility of models cm^1 and cm^2 .

An even more problematic type of incompatibility is conflict incompatibility, where the merged process are *fully incompatible* with each other.

Definition 4.6.13. (Conflict incompatibility)

Let $cm^1, cm^2, \dots, cm^n \in \mathcal{U}_{cm}$ be n constraint models without conflicts, i.e., $\forall_{1 \leq i \leq n} \mathcal{E}_{\models cm^i}^* \neq \emptyset$. If $\mathcal{E}_{\models cm^1 \oplus cm^2 \oplus \dots \oplus cm^n}^* = \emptyset$, then we say that $cm^1 \oplus cm^2 \oplus \dots \oplus cm^n$ has a conflict due to incompatibility of cm^1, cm^2, \dots, cm^n . \square

Consider the original models cm^1, cm^2 and cm^3 from Example 4.6.11. There are no conflicts in any of the original models, i.e., $\forall_{1 \leq n \leq 3} \mathcal{E}_{\models cm^i}^* \neq \emptyset$. However, the merged model $cm^{1 \oplus 2 \oplus 3} = cm^1 \oplus cm^2 \oplus cm^3$ has a conflict, i.e., $\mathcal{E}_{\models cm^{1 \oplus 2 \oplus 3}}^* = \emptyset$ and the cause of this conflict are constraints $C^{conf}(cm^{1 \oplus 2 \oplus 3}) = \{\{c^2, c^3, c^4\}\}$ (note that the model $cm^{1 \oplus 2 \oplus 3}$ is the same like the model from Example 4.6.7). Therefore, the merged model $cm^{1 \oplus 2 \oplus 3}$ has a conflict due to the full incompatibility of models cm^1, cm^2 and cm^3 .

4.7 Summary

In this chapter we have presented a formal foundation for a constraint-based approach to process models is based on models that consist of activities, optional constraints and mandatory constraints. Mandatory constraints in the model determine which are the possible executions of the model's instances, i.e., they determine the set of satisfying traces for the model⁵. The more mandatory constraints the model has, the fewer possible executions there are likely to be. The fewer mandatory constraints the model has, the more possible executions there are likely to be. As an extreme example, any execution is possible if a model does not have any mandatory constraints.

People execute instances of constraint models by triggering events on activities from the instances (e.g., *started*, *completed*, etc.). Events that users trigger by executing activities are added to the instance trace. Each change of the instance trace causes the instance to change state – given the instance trace, the instance can be *satisfied*, *temporarily violated* or *violated*. The execution rule makes sure that users execute each instance in a way that does not violate the instance model. Users can decide to finish executing a constraint instance (i.e., to close the instance) only if the instance state is *satisfied*.

The constraint-based approach presented in this chapter offers all types of flexibility identified by [226–228] (cf. Section 2.1.2). By allowing everything that does not violate mandatory constraints, constraint models offer many possibilities for execution of model instances while enforcing a number of basic rules expressed as constraints. Constraint models are, therefore, *flexible by design* [226–228], unlike traditional approaches which require much more effort in order to support

⁵Chapter 5 describes a formal specification of constraints and sets of satisfying traces for constraint models.

this type of flexibility (cf. Chapter 3). Optional constraints make constraint models *flexible by deviation* [226–228] because these constraints are used as a guidance for execution and people can decide whether to violate them or not. Although constraint models allow for many possible executions, sometimes it might be necessary to change the model of an already running instance (i.e., an instance that is being executed). This type of change is referred to as *ad-hoc change* and it makes constraint models be able to offer *flexible by change* [226–228]. Ad-hoc change can be applied on an instance if and only if this change does not result in a *violated* instance. Finally, in Section 6.11 we describe how the DECLARE prototype offers *flexibility by underspecification* [226–228] by allowing for model decomposition.

Two types of errors can occur in constraint models. First, some events can be dead in the model, i.e., users can never trigger these events while executing model instances. This can lead to situations where users can never execute an activity, although the activity is in the model. Second, it might happen that some mandatory constraints are conflicting in the model. Due to this error, it is never possible for any model instance to become *satisfied* and users cannot execute such instances in a correct way. Verification of constraint models against dead activities and conflicts is necessary in order to support development of correct models. Moreover, constraint model verification needs to detect the smallest group of mandatory constraints that cause the error in order to help developers to understand and eliminate the error.

In the remainder of this thesis we will present a formal language that can be used for constraint specification (in Chapter 5) and the DECLARE system - a prototype of a workflow management system based on this language and the constraint-based approach (in Chapter 6). Note that the proposed language is one *example* of a formal language suitable for a constraint-based approach and that other languages could be used for the same purpose. The prototype DECLARE is developed to support easy implementation of any other LTL-based language that can be used for this purpose.

Chapter 5

Constraint Specification with Linear Temporal Logic

In Chapter 4 we presented a formal foundation for constraint-based process models. However, so far we did not propose a formal language for constraint specification and in all examples in Chapter 4 we used a natural language (i.e., English) to specify constraints (e.g., Example 4.3.2 on page 96). In this chapter we propose Linear Temporal Logic (LTL) [74] as a formalism for constraint specification. We will show how LTL can be used to specify constraints and to retrieve a finite representation of the set of traces that satisfy a constraint and a model (cf. sections 4.1 and 4.2). Using LTL makes it easy to determine states of instances and their constraints (cf. Section 4.4), to change instances in an ad-hoc manner (cf. Section 4.5) and to verify constraint models (cf. Section 4.6). In this chapter we present one particular example of an LTL-based language – the ConDec language. ConDec constraints and models are represented graphically to users, while underlying LTL formulas are hidden. Therefore, users of ConDec models do not need to have knowledge of LTL. Note that, although we will use the ConDec language throughout this chapter, the principles presented in this chapter can be applied to any other language based on LTL or similar temporal logics. All principles that are presented in this chapter are implemented in the DECLARE prototype, which is described in Chapter 6.

5.1 LTL for Business Process Models

LTL is a special kind of logic and is used for describing sequences of transitions between states in reactive systems [74]. In LTL formulas time is not considered explicitly. Instead, LTL can specify properties like ‘eventually some state is reached’ (cf. the so-called ‘expectation’ properties in Section 4.4) or ‘some error state is never reached’ (cf. the so-called ‘safety’ properties in Section 4.4). This type of logic is extensively used in the field of model-checking: systems can be

checked against properties specified in LTL [74]. For this purpose, many algorithms that generate automata from LTL formulas have emerged in the model-checking field [74,111,112]. Automata generated from properties specified as LTL formulas can be used to check if a system satisfies the properties. For example, the SPIN tool [132] can check a system model specified in PROMELA (Process Meta-Language) against properties specified in LTL. In order to check a property against a system modeled in PROMELA, the SPIN tool generates two automata: one representing the system model and one representing the *negation* of the LTL formula representing the property. If the intersection of the two automata is empty, then the system satisfies the property.

Due to its declarative nature, LTL can be used for the formal specification of constraints in constraint-based process models. Automata generated from LTL formulas can be used for automated execution and verification of such models. Automata generated from LTL are used to represent the constraint-based model itself and regulate the execution of the model in a way that satisfies the constraints specified as LTL formulas. However, there are *two differences* between the regular LTL and the LTL applied to business process models. The two special properties of LTL for business processes are illustrated in Figure 5.1.

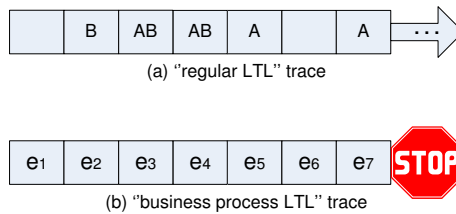


Figure 5.1: LTL for business processes

The first difference between regular LTL and the business processes LTL is the length of traces. On the one hand, the model-checking field is mostly concerned with complex systems that are designed not to halt, e.g., schedulers, telephone switches, power plants, traffic lights, etc. Therefore, regular LTL considers infinite traces (cf. Figure 5.1(a)). On the other hand, execution of each business process models eventually terminates (cf. Section 4). Because of this, the infinite semantics of regular LTL cannot be applied [74, 111, 112] to constraint models and we need to adjust LTL to consider finite traces (cf. Figure 5.1(b)) in order to apply LTL to constraint models. Note that it is possible to check a finite system in the SPIN tool by using the Stuttering extension: the system is modeled in such a way in PROMELA that the last state is repeated infinitely. However, in the constraint-based approach the LTL formula represents the system itself, and the finite trace semantics needs to be applied to LTL itself. In order to apply LTL to the finite semantics of execution traces of constraint models, we use a simple and efficient approach for applying LTL to finite traces presented by Giannakopoulou

et al. in [112]. Further in this section, we will consider the LTL for finite traces presented in [112] in more detail.

The second difference between regular LTL and the business processes LTL is the semantics of elements in a trace. Regular LTL assumes that one element of the trace can refer to more than one property. For example, it is possible to monitor two properties: (A) the motor temperature is higher than 80 degrees and (B) the speed of the turbine is higher than 150 km/h. As Figure 5.1(a) shows, each element of the trace could then refer to: (1) none of the two properties, i.e., neither A nor B hold, (2) only property A, i.e., A holds and B does not hold, (3) only property B, i.e., B holds and A does not hold, or (4) properties A and B, i.e., both A and B hold. In the case of execution traces of constraint models (cf. Definition 4.1.1 on page 85) we assume that only one property holds at one moment, i.e., each of the elements of the trace refers to exactly one event, as shown in Figure 5.1(b).

In the remainder of this section we present how we adjust LTL itself and automata generated from LTL to finite traces consisting of single events, as shown in Figure 5.1(b). A well-formed LTL formula consists of classical logical operators and several temporal operators, and it evaluates to **true** or **false** given an execution trace (cf. Definition 5.1.1).

Definition 5.1.1. (Well-Formed LTL Formula)

Recall that \mathcal{E} is the set of all possible events (cf. Section 4.1). Let $E \subseteq \mathcal{E}$ be a set of events. Every $e \in E$ is a well-formed formula over E . If p and q are well-formed formulas, then also **true**, **false**, $!p$, $p \wedge q$, $p \vee q$, $\Box p$, $\Diamond p$, $\bigcirc p$, pUq and pWq are also well-formed formulas over E .

From a semantical point of view, a well-formed LTL formula p over E is a function $p : \mathcal{E}^* \rightarrow \{\mathbf{true}, \mathbf{false}\}$. Let $\sigma \in \mathcal{E}^*$ be a trace. If p is a well-formed formula and it holds that $p(\sigma) = \mathbf{true}$ then we say that p satisfies σ , denoted by $\sigma \models p$. If $p(\sigma) = \mathbf{false}$ then we say that p does not satisfy σ , denoted by $\sigma \not\models p$. Recall that $\sigma^{i \rightarrow}$ denotes the suffix of σ starting at $\sigma[i]$ (cf. Definition 4.1.1 on page 85). The semantics of LTL are defined as follows:

proposition: $\sigma \models e$ if and only if $e = \sigma[1]$, for $e \in E$,

not (!): $\sigma \models !p$ if and only if not $\sigma \models p$,

and (\wedge): $\sigma \models p \wedge q$ if and only if $\sigma \models p$ and $\sigma \models q$,

or (\vee): $\sigma \models p \vee q$ if and only if $\sigma \models p$ or $\sigma \models q$,

next (\bigcirc): $\sigma \models \bigcirc p$ if and only if $\sigma^{2 \rightarrow} \models p$,

until (U): $\sigma \models pUq$ if and only if $(\exists_{1 \leq i \leq n} : (\sigma^{i \rightarrow} \models q \wedge (\forall_{1 \leq j < i} : \sigma^{j \rightarrow} \models p)))$, and

Also, abbreviations are used:

implication ($p \Rightarrow q$): for $!p \vee q$,

equivalence ($p \Leftrightarrow q$): for $(p \wedge q) \vee (!p \wedge !q)$,

true (true**):** for $p \vee !p$,

false (false): for !true ,

eventually (\diamond): for $\diamond p = \text{trueUp}$,

always (\square): for $\square p = \text{!}\diamond\text{!}p$, and

weak until (W): for $pWq = (pUq) \vee (\square p)$.

□

As specified in Definition 5.1.1, a well-formed LTL formula can use classical logical operators ($!$, \wedge and \vee) and several additional temporal operators (\circlearrowleft , U , W , \square and \diamond). The semantics of operators $!$, \wedge and \vee is the same like in the classical logic, while operators \circlearrowleft , U , W , \square and \diamond have a special, temporal, semantics:

- Operator always ($\square p$) specifies that p holds at every position in the trace (cf. Figure 5.2(a)),
- Operator eventually ($\diamond p$) specifies that p will hold at least once in the trace (cf. Figure 5.2(b)),
- Operator next ($\circlearrowleft p$) specifies that p holds in the next element of the trace (cf. Figure 5.2(c)),
- Operator until (pUq) specifies that there is a position where q holds and p holds in all preceding positions in the trace (cf. Figure 5.2(d)),
- Operator weak until (pWq) is similar to operator until (U), but it does not require that q ever becomes true.

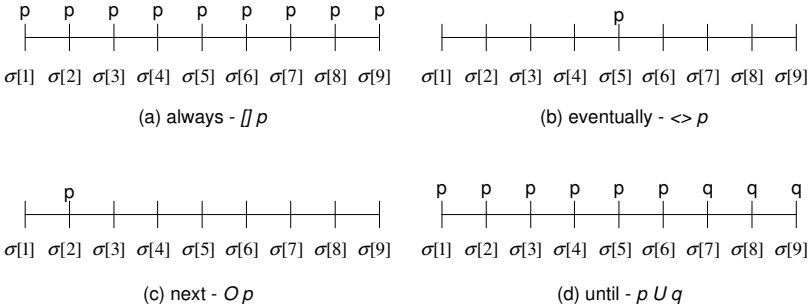


Figure 5.2: Semantics of some LTL operators

The difference between the regular LTL and LTL in Definition 5.1.1 is two-fold. First, the finite semantics is expressed in the until (U) operator [112]. In regular LTL the until operator is defined as follows: $\sigma \models \varphi U \psi$ if and only if $(\exists_{1 \leq i} : (\sigma^{i \rightarrow} \models \psi \wedge (\forall_{1 \leq j < i} : \sigma^{j \rightarrow} \models \varphi)))$. The fact that i does not have an upper bound (i.e., $\exists_{1 \leq i} : \dots$) reflects the infinite semantics of regular LTL. The finite semantics of traces is reflected in the upper bound n of i in the until (U) operator in Definition 5.1.1 (i.e., $\exists_{1 \leq i \leq n} : \dots$ because $|\sigma| = n$). Second, the fact that one element of a trace can refer to multiple properties in regular LTL and to one

property in LTL for business processes is expressed in the way the proposition is defined. On the one hand, in regular LTL it is defined that $\sigma \models e$ if and only if $e \in \sigma[1]$, for $e \in E$, i.e., each element $\sigma[i]$ of a trace is a set of properties. On the other hand, in Definition 5.1.1 we consider a special case of this requirement where each element of a trace refers to exactly one event, i.e., we check if the event is the first element of the trace $e = \sigma[1]$.

Because LTL formulas evaluate execution traces to **true** or **false** (cf. Definition 5.1.1), LTL can be used to formally specify the semantics of a constraint $c = (E, f)$ (cf. Definition 4.1.4 on page 87), i.e., LTL can be used to formally specify f . We refer to any constraint for which f is a well-formed LTL formula over E as to a *LTL constraint*. Similarly, if all mandatory and optional constraints in a constraint model are LTL constraints, then we say that the constraint model is an LTL constraint model.

Note that LTL is not the only language that can be used for formal specification constraints. Other declarative languages can be also used. For example, Computation Tree Logic (CTL) [74] is another logic that can be used to specify the semantics of constraints. Although LTL and CTL are similar languages, each of them has some advantages over the other. However, so far, the debate about which of these two languages is ‘better’ remains unsolved [132, 253]. For example, there are some properties that can be specified only in LTL or in CTL, but not in both languages [132]. On the one hand, the *fairness* property (for each execution, there is some state at which a property starts to hold forever) can be expressed in LTL, but not in CTL. On the other hand, the *reset* property (from every state there exists at least one execution that can return the system to its initial state) can be expressed in CTL, but not in LTL. We chose LTL for the specification of constraints for two reasons. First, we were inspired by the so called LTL Checker plug-in [25] of the process mining tool ProM [8, 91], which can be used for verification of past executions against properties specified in LTL (the LTL Checker is described in more detail in Chapter 7). Second, as a simple and straight-forward language, LTL seems to be a good starting point for the constraint-based approach.

5.2 ConDec: An Example of an LTL-Based Constraint Language

ConDec is a constraint-based language that uses LTL to formally specify the semantics of constraints. Because LTL formulas can be difficult to understand by non-experts, ConDec associates a *graphical representation* to each constraint. By using this approach, users of ConDec models do not need to have knowledge of LTL. Instead, they can learn the intuitive meanings of names and graphical representations of constraints.

Traditional process modeling languages offer a small set of constructs that

can be used to model relationships between process activities (e.g., sequence, choice branching, parallel branching and loops). Because it uses LTL for constraint specification, ConDec offers many *constraint templates*, i.e., types of constraints that can be used to create constraints in ConDec models. Consider, for example, the following two constraints: (1) the constraint c_1 from the model cm^R in Example 4.2.3 on page 92 specifying that $f_1 =$ “Every occurrence of event (*curse, completed*) must eventually be followed by at least one occurrence of event (*pray, completed*).” and (2) the constraint c_6 from model cm^{FT} in Example 4.3.2 on page 96, which specifies that $f_6 =$ “Every occurrence of event (*perform surgery, completed*) must eventually be followed by at least one occurrence of event (*prescribe rehabilitation, completed*).” The first constraint can be specified with formula $\Box((\textit{curse, completed}) \Rightarrow \Diamond(\textit{pray, completed}))$ and the second one with a similar formula $\Box((\textit{perform surgery, completed}) \Rightarrow \Diamond(\textit{prescribe rehabilitation, completed}))$. Both constraints impose the same type of relation, i.e., one event is a *response* of the other event, and their LTL specifications are similar: $\Box((A, \textit{completed}) \Rightarrow \Diamond(B, \textit{completed}))$. Instead of having to individually specify formulas for these two constraints in their models, the constraints can be created from the constraint template called *response*. Each template in ConDec has (1) a name, (2) an LTL formula and (3) a graphical representation. Figure 5.3 shows how templates are used to create constraints in ConDec models.

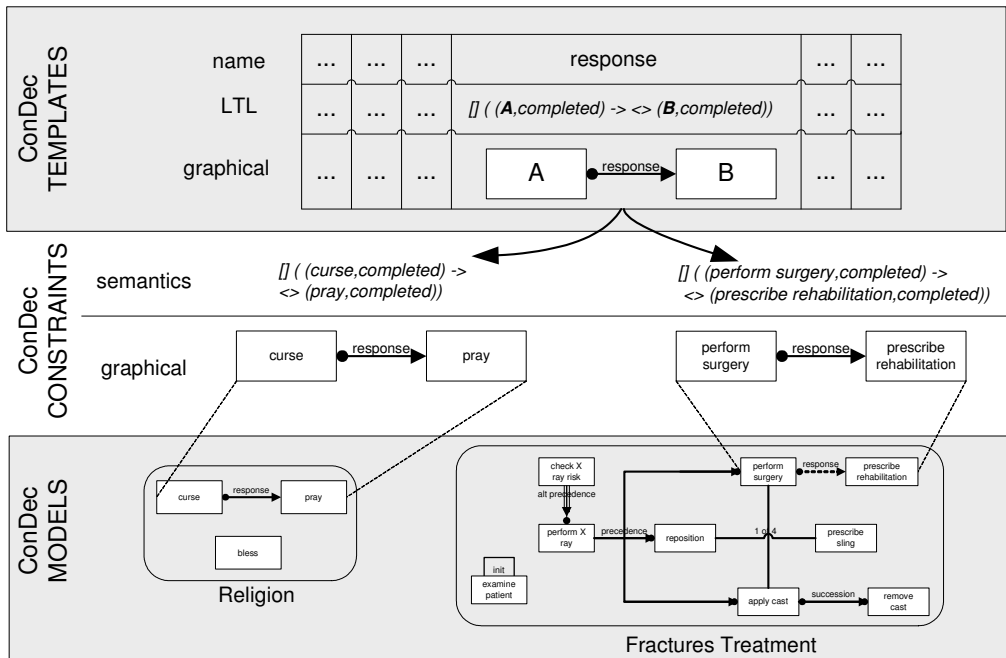


Figure 5.3: ConDec templates, constraints and models

Instead of specifying the LTL formula and graphical representation separately for each constraint, a constraint is created based on a template: a constraint inherits the name, graphical representation and the LTL formula from its template. Constraints are presented graphically in ConDec models, while the underlying LTL formula remains hidden. For example, activities and constraints in the two ConDec models in Figure 5.3 are represented graphically. Activities are presented as rectangles and constraints as special lines between activities. The model cm^R from Example 4.2.3 on page 92 has three activities and one constraint and the Fractures Treatment model cm^{FT} from Example 4.3.2 on page 96 has nine activities and six constraints. ConDec models for these two examples are shown in Figure 5.3. Each of the constraints in these two models is created from a ConDec template, i.e., the graphical representation of the template represents the constraint while the underlying LTL formula remains hidden. Due to this fact, users do not need to be LTL experts in order to develop and work with ConDec models. Instead, it is enough to learn the graphical representation and semantics of ConDec templates, e.g., the *response* template between activities A and B is represented by the special line like in Figure 5.3 and it specifies that every occurrence of A has to be eventually followed by at least one occurrence of B . We refer to ‘imaginary’ activities of a template as to template’s *parameters*. For example, the *response* template has two parameters, i.e., A and B .

The ConDec language is just one example of a language for constraint specification. Other languages can use different types of constraints depending on the application area, e.g., the DecSerFlow language [37,38] for web services domain, the CIGDec language [176] for medical processes, etc. We used property specification patterns presented in [95] as an inspiration for developing ConDec templates. ConDec has more than twenty templates, which structured into three groups. First, there are *existence* templates that specify how many times or when one activity can be executed. For example, the $1..*$ template specifies that an activity must be executed at least once and the *init* template can be used to specify that execution of each instance has to start with a specific activity. Second, *relation* templates define some relation between two (or more) activities. For example, *response* is a relation template. Third, *negation* templates define a negative relation between activities. For example, the *responded absence* template defines that two activities cannot be executed both within the same instance. Finally, *choice* templates can be used to specify that one must choose between activities. An example of such a template is the *1 of 4* template, which is used to specify that at least one of the four given activities has to be executed.

5.2.1 Existence Templates

Figure 5.4 shows the so-called *existence* templates. These templates involve only one activity and they define the cardinality or the position of the activity in a trace. For example, the first template is called *existence* and it is graphically

represented with the annotation “1..*” above the activity. This indicates that A is executed at least once. Templates $existence2$, $existence3$, and $existence_N$ all specify a lower bound for the number of occurrences of A . It is also possible to specify an upper bound for the number of occurrences of A . Templates $absence$, $absence2$, $absence3$, and $absence_N$ are also visualized by showing the range above the activity, e.g., “0.. N ” for the requirement $absence_{N+1}$. Similarly, it is possible to specify the exact number of occurrences as shown in Figure 5.4, e.g., template $exactly_N(A)$ is denoted by an N above the activity and specifies that A should be executed exactly N times. Finally, the template $init(A)$ can be used to specify that activity A must be the first executed activity in an instance.

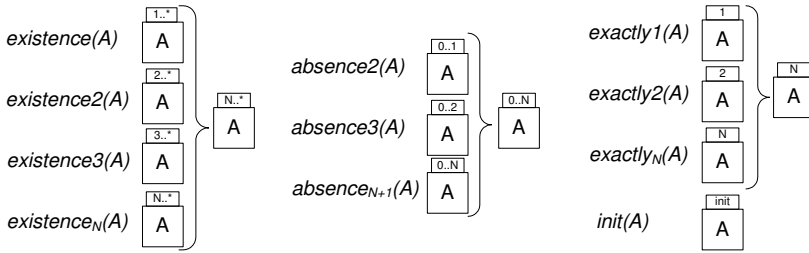


Figure 5.4: Notations for the existence templates

Table 5.1 provides the semantics for each of the notations shown in Figure 5.4, i.e., each template is expressed in terms of an LTL formula. Formula for template $existence(A)$ defines that event (A, t_c) has to hold eventually which implies that in any instance A has to be executed at least once¹. Formula for template $existence_N(A)$ shows how it is possible to express a lower bound N for the number of occurrences of A in a recursive manner, i.e., $existence_N(A) = \diamond((A, t_c) \wedge \bigcirc(existence_{N-1}(A)))$. Formula for template $absence_N(A)$ can be defined as the negation of $existence_N(A)$. Together they can be combined to express that for any full execution, A should be executed a pre-specified number N , i.e., $exactly_N(A) = existence_N(A) \wedge absence_{N+1}(A)$. Formula for template $init(A)$ defines that the only possible events before event (A, t_c) are events (A, t_s) and (A, t_x) , i.e., events involving activities other than A can be executed only after event (A, t_c) .

5.2.2 Relation Templates

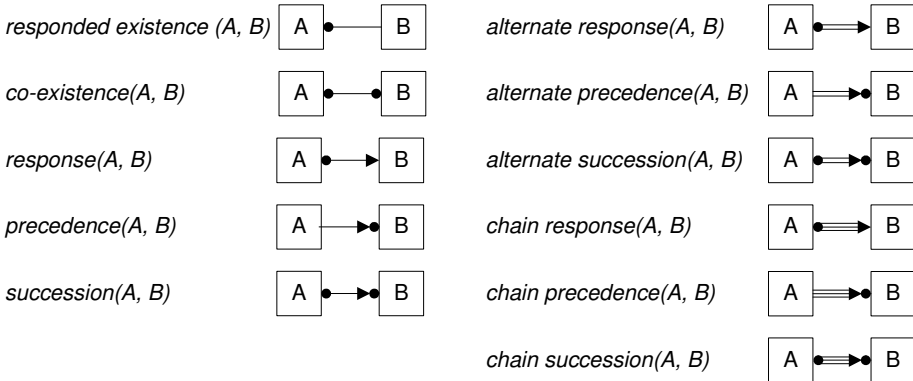
Figure 5.5 shows the so-called relation templates. While an existence template describes the cardinality of one activity, a relation template defines a dependency between multiple activities. Figure 5.5 only shows binary relationships (i.e., between two activities), however, in ConDec there are also templates that can involve generalizations to three or more activities. For simplicity however, we

¹Recall that an event consists of an activity (e.g., A) and an event type (e.g., t_c).

Table 5.1: LTL formulas for existence templates (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$)

name of formula	LTL expression
$existence(A)$	$\diamond(A, t_c)$
$existence_2(A)$	$\diamond((A, t_c) \wedge \circ(existence(A)))$
$existence_3(A)$	$\diamond((A, t_c) \wedge \circ(existence_2((A))))$
...	...
$existence_N(A)$	$\diamond((A, t_c) \wedge \circ(existence_{N-1}(A)))$
$absence_2(A)$	$\neg existence_2(A)$
$absence_3(A)$	$\neg existence_3(A)$
...	...
$absence_N(A)$	$\neg existence_N(A)$
$exactly_1(A)$	$existence(A) \wedge absence_2(A)$
$exactly_2(A)$	$existence_2(A) \wedge absence_3(A)$
...	...
$exactly_N(A)$	$existence_N(A) \wedge absence_{N+1}(A)$
$init(A)$	$((A, t_s) \vee (A, t_x))W(A, t_c)$

first focus on the binary relationships shown in Figure 5.5. All relation templates have activities A and B as parameters. The line between the two activities in the graphical representation is unique for the formula, and reflects the semantics of the relation. The *responded existence* template specifies that if activity A is executed, activity B also has to be executed (at any time, i.e., either before or after activity A is executed). According to the *co-existence* template, if one of the activities A or B is executed, the other one has to be executed as well.

**Figure 5.5:** Notations for the relation templates

While the first two templates do not consider the order of activities, templates *response*, *precedence* and *succession* do consider the ordering of activities. Template *response* requires that every time activity A executes, activity B has to be executed after it. Note that this is a very relaxed relation of *response*, because

B does not have to execute straight after A , and another A can be executed between the first A and the subsequent B . The template *precedence* requires that activity B is preceded by activity A , i.e., it specifies that activity B can be executed only after activity A is executed. Just like in the *response* template, other activities can be executed between activities A and B . The combination of the *response* and *precedence* templates defines a bi-directional execution order of two activities and is called *succession*. In this template, both *response* and *precedence* relations have to hold between the activities A and B .

Templates *alternate response*, *alternate precedence* and *alternate succession* strengthen the *response*, *precedence* and *succession* templates, respectively. In the *alternate* templates activities A and B have to alternate. If activity B is *alternate response* of activity A , then after the execution of an activity A activity B has to be executed and the activity A can be executed again only after activity B is executed. Similarly, in the *alternate precedence* every instance of activity B has to be preceded by an instance of activity A and the activity B cannot be executed again before the activity A is also executed again. The *alternate succession* is a combination of the *alternate response* and *alternate precedence*.

Even more strict ordering relations are specified by the last three templates shown in Figure 5.5: *chain response*, *chain precedence* and *chain succession*. These templates require that the executions of the two activities (A and B) are next to each other. In the *chain response* template activity B has to be executed directly after activity A . The *chain precedence* template requires that the activity A directly precedes each B . Since the *chain succession* template is the combination of the *chain response* and *chain precedence* templates, it requires that activities A and B are always executed next to each other.

Table 5.2 shows LTL formulas for the templates shown in Figure 5.5. The formula for *responded existence*(A, B) indicates that an occurrence of (A, t_c) should always imply an occurrence of event (B, t_c) , either before or after (A, t_c) . Formula for *co existence*(A, B) means that the existence of (A, t_c) implies the existence of (B, t_c) and vice versa. The formula for *response*(A, B) specifies that at any point in time when event (A, t_c) occurs there should eventually be an occurrence of event (B, t_c) . The formula for *precedence*(A, B) is similar to *response* but now looking backwards, i.e., (B, t_s) , (B, t_c) and (B, t_x) cannot occur before occurrence of event (A, t_c) . The formula for *succession*(A, B) is defined by combining both into: $\text{response}(A, B) \wedge \text{precedence}(A, B)$. The *alternate response*(A, B) formula specifies that any occurrence of (A, t_c) implies that in the next state and onwards no (A, t_c) may occur until a (B, t_c) occurs. The formula for *alternate precedence*(A, B) is a bit more complicated: it implies that at any point in time where (B, t_c) occurs and at least one other occurrence of (B, t_c) follows, an (A, t_c) should occur before the following occurrence of (B, t_s) , (B, t_c) or (B, t_x) . The formula for *alternate succession*(A, B) combines both into $\text{alternate response}(A, B) \wedge \text{alternate precedence}(A, B)$. The formula for *chain response*(A, B) indicates that any occurrence of (A, t_c) should be di-

rectly followed by (B, t_s) . The formula for *chain precedence* (A, B) is the logical counterpart: it specifies that any occurrence of (A, t_c) should be directly followed by (B, t_s) and any occurrence of (B, t_s) should be directly preceded by (A, t_c) .

Table 5.2: LTL formulas for relation templates (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$)

name of formula	LTL expression
<i>responded existence</i> (A, B)	$\diamond(A, t_c) \Rightarrow \diamond(B, t_c)$
<i>co existence</i> (A, B)	$\diamond(A, t_c) \Leftrightarrow \diamond(B, t_c)$
<i>response</i> (A, B) <i>precedence</i> (A, B) <i>succession</i> (A, B)	$\square((A, t_c) \Rightarrow \diamond(B, t_c))$ $(\neg((B, t_s) \vee (B, t_c) \vee (B, t_x)))W(A, t_c)$ $\text{response}(A, B) \wedge \text{precedence}(A, B)$
<i>alternate response</i> (A, B) <i>alternate precedence</i> (A, B)	$\text{response}(A, B) \wedge$ $\square((A, t_c) \Rightarrow \circ(\text{precedence}(B, A)))$ $\text{precedence}(A, B) \wedge$ $\square((B, t_c) \Rightarrow \circ(\text{precedence}(A, B)))$
<i>alternate succession</i> (A, B)	$\text{alternate response}(A, B) \wedge$ $\text{alternate precedence}(A, B)$
<i>chain response</i> (A, B) <i>chain precedence</i> (A, B) <i>chain succession</i> (A, B)	$\text{response}(A, B) \wedge \square((A, t_c) \Rightarrow \circ(B, t_s))$ $\text{precedence}(A, B) \wedge \square(\circ(B, t_s) \Rightarrow (A, t_c))$ $\text{chain response}(A, B) \wedge \text{chain precedence}(A, B)$

5.2.3 Negation Templates

Figure 5.6 shows the negation templates, which are the negated versions of the relation templates. (Ignore the grouping of constraints on the right-hand side of Figure 5.6 for the moment. Later, we will show that the eight constraints can be reduced to three equivalence classes.) The first two templates negate the *responded existence* and *co-existence* templates. The *not responded existence* template specifies that if activity A is executed activity B must never be executed (not before nor after activity A). The *not co-existence* template applies *not responded existence* from A to B and from B to A . *It is important to note that the term ‘negation’ should not be interpreted as the ‘logical negation’, e.g., if activity A never occurs, then both *responded existence* (A, B) and *not responded existence* (A, B) hold (i.e., one does not exclude the other).* The *not response* template specifies that after the execution of activity A , activity B cannot be executed any more. According to the template *not precedence* activity B cannot be preceded by activity A . The last three templates are negations of templates *chain response*, *chain precedence* and *chain succession*. The *not chain response* template specifies that A should never be followed directly by B and the *not chain precedence* template specifies that B should never be preceded directly by A . Templates *not chain response* and *not chain precedence* are combined in the *not chain succession* template. Note that Figure 5.6 does not show negation templates for the alternating variants of *response*, *precedence*, and *succession*.

The reason is that there is no straightforward and intuitive interpretation of the converse of an alternating *response*, *precedence*, or *succession*.

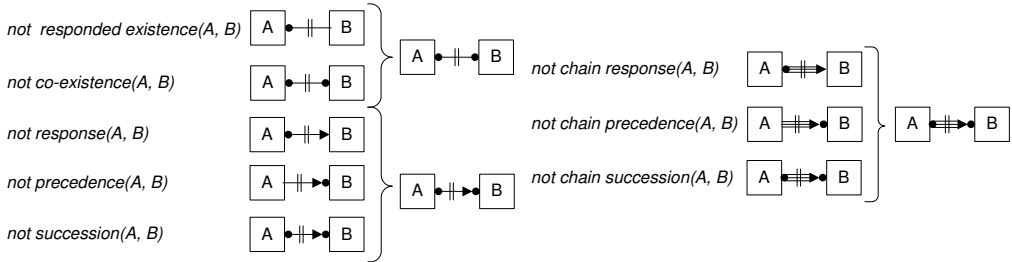


Figure 5.6: Notations for the negations templates

Table 5.3 shows the LTL expressions of the templates shown in Figure 5.6. Table 5.3 also shows that some of the templates are equivalent, i.e., *not co-existence* and *not responded existence* are equivalent and similarly the next two pairs of three formulas are equivalent. Note that a similar grouping is shown in Figure 5.6 where a single representation for each group is suggested. The formula for *not responded existence*(A, B) specifies that event (B, t_c) cannot occur if event (A, t_c) occurs. However, since the ordering here does not matter, *not responded existence*(A, B) = *not responded existence*(B, A) and hence coincides with *not co existence*(A, B). The formula for *not response*(A, B) defines that after any occurrence of (A, t_c) , (B, t_s) may never happen (or formulated alternatively: any occurrence of (B, t_c) should take place before the first (A, t_c)). The formula for *not precedence*(A, B) defines that, if (B, t_s) may occur in some future state, then (A, t_c) cannot occur in the current state. It is easy to see that *not precedence*(A, B) = *not response*(A, B) because both state that no (B, t_c) should take place after the first (A, t_c) (if any). Since the formula for *not succession*(A, B) combines both *not response*(A, B) and *not precedence*(A, B), also *not succession*(A, B) = *not response*(A, B). The last three formulas are negations of formulas *chain response*, *chain precedence* and *chain succession*. It is easy to see that they are equivalent *not chain response*(A, B) = *not chain precedence*(A, B) = *not chain succession*(A, B).

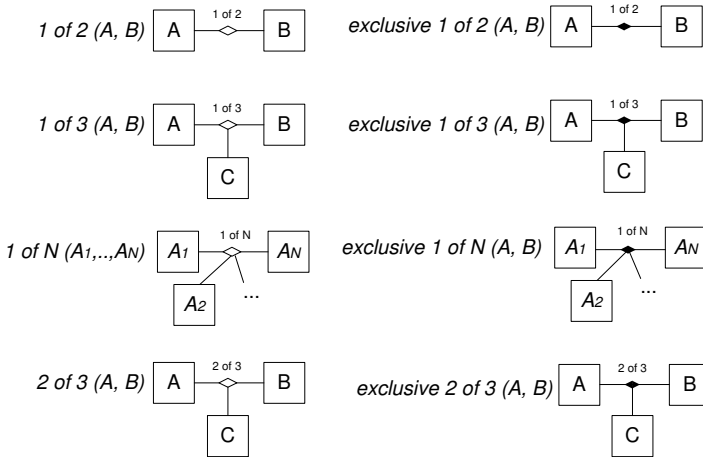
5.2.4 Choice Templates

Figure 5.7 shows the so-called choice templates, which specify that it is necessary to choose between several activities. The *1 of 2* template specifies that at least one of the two activities A and B has to be executed, but both can be executed and each of them can be executed an arbitrary number of times. The *1 of 3* template specifies that at least one of the three activities A , B and C has to be executed, but all three activities can be executed an arbitrary number of times as long as at least one of them occurs at least once. Similarly, a template *1 of N* can

Table 5.3: LTL formulas for negation templates (templates grouped together are equivalent) (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \textit{started}$ and $t_c = \textit{completed}$)

name of formula	LTL expression
<i>not responded existence</i> (A, B) <i>not co existence</i> (A, B)	$\diamond(A, t_c) \Rightarrow !(\diamond(B, t_c))$ <i>not responded existence</i> (A, B) \wedge <i>not responded existence</i> (B, A)
<i>not response</i> (A, B) <i>not precedence</i> (A, B) <i>not succession</i> (A, B)	$\square((A, t_c) \Rightarrow !(\diamond((B, t_s) \vee (B, t_c))))$ $\square(\diamond(B, t_s) \Rightarrow !(A, t_c))$ <i>not response</i> (A, B) \wedge <i>not precedence</i> (A, B)
<i>not chain response</i> (A, B) <i>not chain precedence</i> (A, B) <i>not chain succession</i> (A, B)	$\square((A, t_c) \Rightarrow \circ(! (B, t_s)))$ $\square(\circ(B, t_s) \Rightarrow !(A, t_c))$ <i>not chain response</i> (A, B) \wedge <i>not chain precedence</i> (A, B)

specify that at least one of the N activities has to be executed, but all activities can be executed an arbitrary number of times. It is also possible to specify that M of N activities has to be executed, but this has to be done explicitly, i.e., it is not possible to specify this in a recursive way like in the existence templates. For example, the *2 of 3* template specifies that at least two of the three activities A , B and C have to be executed, but all three can be executed an arbitrary number of times.

**Figure 5.7:** Notations for the choice templates

The *exclusive* choice templates are stronger than the templates described above. The *exclusive 1 of 2* template specifies that one of the two activities A and B has to be executed, while the other one cannot be executed at all. The *exclusive 1 of 3* template specifies that one of the three activities A , B and C has to be executed one or more times, while the other two cannot be executed at

all. The *exclusive 1 of N* template specifies that one of the N activities has to be executed one or more times, while the other $N-1$ cannot be executed at all. More general exclusive templates must be specified explicitly, i.e., the *exclusive 2 of 3* template specifies that two of the three activities A , B and C have to be executed one or more times, while the remaining one cannot be executed at all.

Table 5.4 shows LTL formulas for each choice template from Figure 5.7. Formula for *1 of 2*(A, B) specifies that either (A, t_c) or (B, t_c) has to occur eventually. Formula *1 of 3*(A, B, C) specifies that either (A, t_c) or (B, t_c) or (C, t_c) has to eventually occur. Similarly, formula *1 of N* (A_1, \dots, A_N) specifies that one of the events $(A, t_c), \dots, (B, t_c)$ has to eventually occur. The formula for *2 of 3*(A, B, C) specifies that two of the events $(A, t_c), (B, t_c)$ or (C, t_c) has to eventually occur. Formulas for exclusive templates strengthen the choice. Formula for *exclusive 1 of 2*(A, B) specifies that either (A, t_c) or (B, t_c) has to eventually occur, but they can occur both. The formula for *exclusive 1 of 3*(A, B, C) specifies that one of the events $(A, t_c), (B, t_c)$ or (C, t_c) has to eventually occur, but the other two cannot occur at all. The formula for *exclusive 1 of N* (A_1, \dots, A_N) can be specified similarly so that it defines that one of the events $(A_1, t_c), \dots$ or (A_N, t_c) has to eventually occur, but the other $N-1$ events cannot occur at all. Finally, formula *exclusive 2 of 3*(A, B, C) specifies that two of the events $(A, t_c), (B, t_c)$ or (C, t_c) have to eventually occur and the third one cannot occur at all.

Table 5.4: LTL formulas for choice templates (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$)

name of formula	LTL expression
<i>1 of 2</i> (A, B)	$\diamond(A, t_c) \vee \diamond(B, t_c)$
<i>1 of 3</i> (A, B, C)	$\diamond(A, t_c) \vee \diamond(B, t_c) \vee \diamond(C, t_c)$
<i>1 of N</i> (A_1, \dots, A_N)	$\diamond(A_1, t_c) \vee \dots \vee \diamond(A_N, t_c)$
<i>2 of 3</i> (A, B, C)	$(\diamond(A, t_c) \wedge \diamond(B, t_c))$ $\vee (\diamond(B, t_c) \wedge \diamond(C, t_c))$ $\vee (\diamond(A, t_c) \wedge \diamond(C, t_c))$
<i>exclusive 1 of 2</i> (A, B) <i>exclusive 1 of 3</i> (A, B, C)	$(\diamond(A, t_c) \wedge !\diamond(B, t_c)) \vee (!\diamond(A, t_c) \wedge \diamond(B, t_c))$ $(\diamond(A, t_c) \wedge !\diamond(B, t_c) \wedge !\diamond(C, t_c)) \vee$ $(!\diamond(A, t_c) \wedge \diamond(B, t_c) \wedge !\diamond(C, t_c)) \vee$ $(\diamond(A, t_c) \wedge !\diamond(B, t_c) \wedge !\diamond(C, t_c))$
<i>exclusive 2 of 3</i> (A, B, C)	$(\diamond(A, t_c) \wedge \diamond(B, t_c) \wedge !\diamond(C, t_c))$ $\vee (\diamond(A, t_c) \wedge !\diamond(B, t_c) \wedge \diamond(C, t_c))$ $\vee (!\diamond(A, t_c) \wedge \diamond(B, t_c) \wedge \diamond(C, t_c))$

Graphical representations of ConDec templates presented in Figures 5.5, 5.6 and 5.7 are used to hide the (potentially) complex LTL formulas. Special symbols in the graphical representation of a template should illustrate the template's semantics. Figure 5.8 shows explanations of intuition behind the graphical notations of ConDec templates.

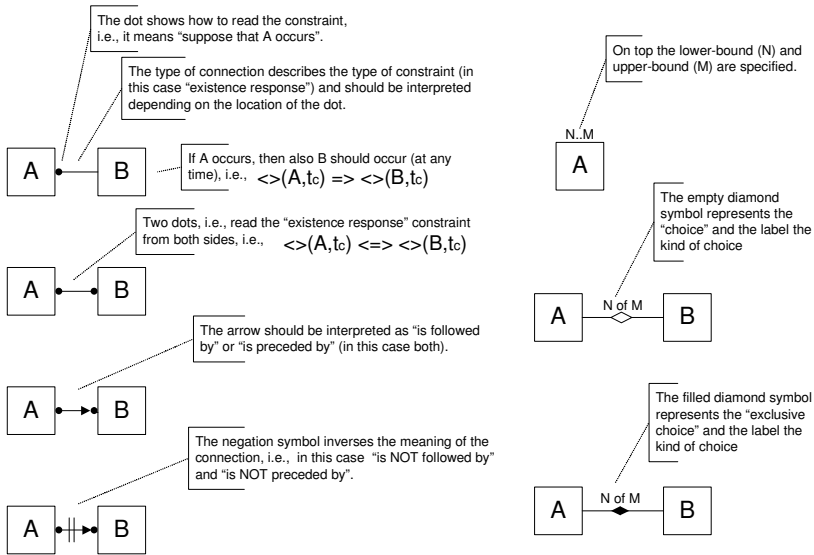


Figure 5.8: Explanation of the graphical notation

5.2.5 Branching of Templates

Each of the ConDec templates involves a specific number of activities. For example, templates $existence(A)$, $response(A,B)$ and $1\ of\ 3(A,B,C)$ involve one, two, and three activities, respectively. This means that, when a constraint is created based on a template, the constraint will involve as many real activities as predefined in the template. In Figure 5.3 we showed how a real activity replaces each of the template's parameters in a constraint. However, each constraint can easily be extended to deal with more activities than defined by its template. Consider, for example, the $response$ template that involves two activities A and B in Figure 5.9. In the simplest case, a $response$ constraint will involve two activities, each of which will replace one of the template's parameter. For example, activities $curse$ and $pray$ simply replace parameters A and B , respectively, in the graphical representation and LTL formula of the $response$ template (cf. 'plain' constraint in Figure 5.9). In some cases it might be necessary to assign more than two activities a parameter in a template. When a template parameter is replaced by more than one activity in a constraint, then we say that this parameter *branches*. An example of a branched $response$ constraint is shown in Figure 5.9: the parameter A is replaced by the activity $curse$ and parameter B is branched on activities $pray$ and $confess$. In case of branching, the parameter is replaced (1) by a multiple arcs to all branched activities in the graphical representation and (2) by a disjunction of branched activities in the LTL formula, as shown in the branched constraint in Figure 5.9. The semantics of branching can vary from template to template, depending on the LTL formula of the template. For

example, the branched constraint in Figure 5.9 specifies that each occurrence of *curse* should eventually be followed by at least one occurrence of activity *pray* or activity *confess*. Note that it is possible to branch all parameters, one parameter or none of the parameters.

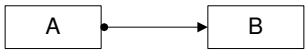

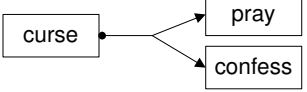
	GRAPHICAL	LTL FORMULA
TEMPLATE		$\Box ((A,t_e) \Rightarrow \Diamond (B,t_e))$
'PLAIN' CONSTRAINT		$\Box ((curse,t_e) \Rightarrow \Diamond (pray,t_e))$
BRANCHED CONSTRAINT		$\Box ((curse,t_e) \Rightarrow \Diamond ((pray,t_e) \vee (confess,t_e)))$

Figure 5.9: Branching the *response* template

The number of possible branches in ConDec constraints is unlimited. For example, it is possible to branch the parameter *B* in the *response* template to *N* alternatives, like shown in Figure 5.10.

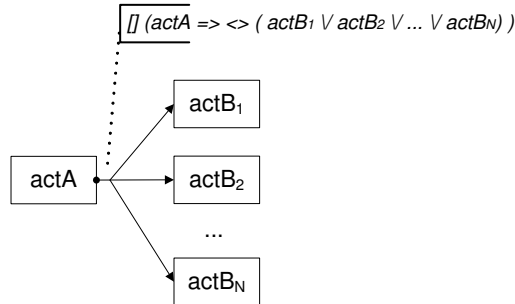


Figure 5.10: Branching the *response* template to multiple activities

5.3 ConDec Constraints

ConDec constraints are created from ConDec templates, such that the LTL formula and graphical representation of the template are associated with the constraint, as shown in Figure 5.11. Template ‘parameters’ (i.e., *A* and *B*) are replaced with ‘real’ activities (i.e., *perform surgery* and *prescribe rehabilitation*) both in the graphical and LTL specification of the constraint.

A finite representation of the set of traces that satisfy a ConDec constraint is retrieved from the LTL formula associated with the constraint. Every LTL formula can be translated into a non-deterministic *finite state automaton* (cf.

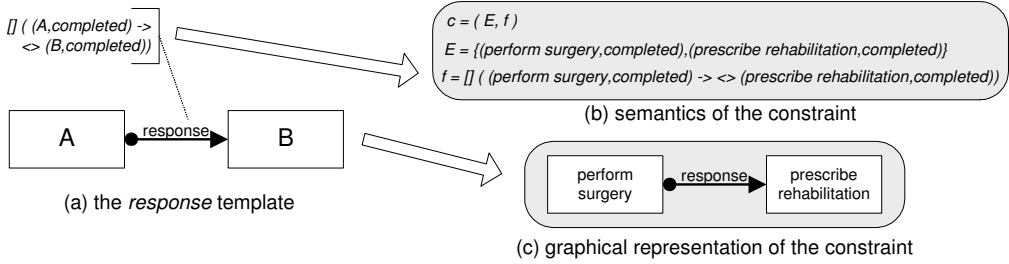


Figure 5.11: A ConDec constraint and its template

Definition 5.3.1) that represents *exactly all traces that satisfy the LTL formula* [74, 111–113, 158].

Definition 5.3.1. (Finite State Automaton FSA)

Finite state automaton *FSA* is a five tuple $\langle \mathcal{E}, S, T, S_0, S_F \rangle$ such that \mathcal{E} is the *alphabet*, S is the finite set of *states*, $T \subseteq S \times \mathcal{E} \times S$ it the *transition relation*, $S_0 \subseteq S$ is the set of *initial states*, and $S_F \subseteq S$ is the set of *accepting states*. \square

Figure 5.12 shows graphical representation of automaton created for constraint in Figure 5.11. This automaton has two states s_0 and s_1 where s_0 is both an initial (marked with an incoming arrow without a source) and accepting (marked with double border) state. Labeled directed arrows between states represent transitions. For example, transition $(s_0, (prescribe\ rehabilitation, t_c), s_0) \in T$ denotes that, if event $(prescribe\ rehabilitation, t_c)$ occurs when the automaton is in the state s_0 , then the automaton stays in state s_0 . In other words, we say that event $(prescribe\ rehabilitation, t_c)$ triggers this transition. The special label “-” denotes a transition that is triggered by any event $e \in \mathcal{E}$. Each transition is an output transition for the state from which it is triggered and an input transition for the state to which it leads. For example, transition $(s_0, -, s_1) \in T$ is an *output transition* for state s_0 and an *input transition* for state s_1 . We also say that s_0 is the *source* and s_1 is the *target* of this transition.

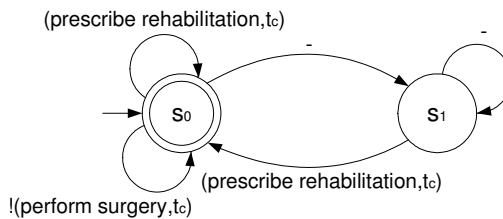


Figure 5.12: A finite state automaton FSA_f for constraint in Figure 5.11 (recall that $t_c \in \mathcal{T}$ is an event type such that $t_c = completed$)

Labels on automata transitions are considered to be boolean values of propositions [112, 158]. Consider, for example, three propositions a , b and c . Label

$!a \wedge !b \wedge c$ represents that (i.e., this transition will fire when) a and b do not hold and c holds, regardless whether other propositions hold or not. A transition labeled with $a \wedge !b$ will fire if a holds and b does not hold, regardless whether other propositions hold or not. Label $!a \wedge !b$ represents a transition that will fire if a and b do not hold, regardless whether other propositions hold or not.

The remainder of this section is structured as follows. First, in Section 5.3.1 we describe how we adjust the automata generated from LTL to specific properties of business processes, which were described in Section 5.1. Second, in Section 5.3.2 we describe how we deal with the fact that the generated automata are non-deterministic automata. Finally, in Section 5.3.3, we describe how we retrieve the set of all traces that satisfy a constraint (cf. Definition 4.1.4 in page 87) from the automaton generated for the constraint.

5.3.1 Adjusting to Properties of Business Processes

In order to apply LTL to execution of business processes, we adjust the regular LTL in two ways, as we described in Section 5.1: (1) we consider LTL for finite traces, and (2) we consider LTL for traces of single events. These two adjustments must also be reflected on the automata generated from LTL.

First, due to the *finite* semantics of execution traces of constraint models, we use the algorithm presented in [112, 113] to create a finite state automaton FSA from ConDec constraint $c = (E, f)$. We will use FSA_f to denote an automaton (1) created for LTL formula f using the algorithm presented in [113], and (2) adjusted for finite traces using the method presented in [112].

Second, due to the fact that classical LTL considers traces which can contain an arbitrary number of propositions at each trace element, automata generated for LTL formulas in [112, 113] might contain transitions that refer to such trace elements. For example, it is possible that a transition has label “(perform surgery, t_s) \wedge (perform reposition, t_s)” because events are treated as propositions. This transition is *blocked* (i.e., it will never fire) in the setting of traces of business processes, because events are triggered one at a time. In other words, a transition is blocked if and only if its label requires more than one event to hold. Naturally, blocked transitions are ignored in the context of execution traces of business processes because they can not be triggered by any element of a trace (i.e., by any event). In order to use these automata for our approach, we *unblock* them in such a way that: (1) all blocked transitions are removed from the automaton, (2) all states that are unreachable from initial states are removed from the automaton and (3) all states from which an accepting state is not reachable are removed from the automaton. Further in this thesis, we assume that all automata are unblocked, unless denoted differently. Definition 5.3.2 gives a formal specification of reachable states: the set of reachable states from state s of the automaton contains all states that can be reached with an arbitrary sequence of transitions from the state s . The unblocked version of an automaton

in specified in Definition 5.3.3.

Definition 5.3.2. (Reachable states R_s^{FSA})

Let $FSA = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be a finite state automaton and $s \in S$ be one state of FSA . State $s' \in S$ is reachable from state s if and only if it holds that $\exists \sigma \in \mathcal{E}^* : (s, \sigma[1], s_1) \in T \wedge (s_1, \sigma[2], s_2) \in T \wedge \dots \wedge (s_{|\sigma|-1}, \sigma[|\sigma|], s') \in T$. The set of states that are reachable from s is defined as $R_s^{FSA} = \{s' \in S \mid \exists \sigma \in \mathcal{E}^* : (s, \sigma[1], s_1) \in T \wedge (s_1, \sigma[2], s_2) \in T \wedge \dots \wedge (s_{|\sigma|-1}, \sigma[|\sigma|], s') \in T\}$. \square

Definition 5.3.3. (Unblocked finite state automaton)

Let $FSA = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be a finite state automaton and $T^B \subseteq T$ be a set of all blocked transitions in T . Automata $FSA^{UB} = \langle \mathcal{E}, S^{UB}, T^{UB}, S_0^{UB}, S_F^{UB} \rangle$ where $S^{UB} \subseteq S$, $T^{UB} \subseteq T$, $S_0^{UB} \subseteq S_0$ and $S_F^{UB} \subseteq S_F$ is the unblocked version of FSA if and only if it holds that

- all blocked transitions are removed, i.e., $T^{UB} = T \setminus T^B$, and
- all states are reachable from the initial state(s), i.e., S^{UB} and S_0^{UB} are the biggest subsets of S such that it holds that $\forall s \in S^{UB} \setminus S_0^{UB} : \exists s^0 \in S_0^{UB} : s \in R_{s^0}^{FSA^{UB}}$, and
- form all states an accepting state is reachable, i.e., S^{UB} and S_F^{UB} are the biggest subsets of S such that it holds that $\forall s \in S^{UB} \setminus S_F^{UB} : R_s^{FSA^{UB}} \cap S_F^{UB} \neq \emptyset$.

\square

In addition to unblocking automata, when presenting automata in figures in this thesis, we will simplify labels on transitions in the following manner: if, due to its label, a transition can be triggered only by a single event, then its label will be replaced by a label containing only that event. For example, a transition with the label $e_1 \wedge !e_2$ can be triggered only by event e_1 and, therefore, in this thesis we will label it only with e_1 . On the other hand, transition with the label $!e_1 \wedge !e_2$ can be triggered with multiple events (i.e., all events except e_1 and e_2), and, therefore, its label will remain the same.

5.3.2 Dealing with the Non-Determinism

Automata created for LTL formulas are *non-deterministic* automata, i.e., one state can have multiple output transitions that are triggered by the same event, but that have different target states. Consider, for example, state s_0 in the automaton in Figure 5.12. If event $(prescribe\ rehabilitation, t_c) \in \mathcal{E}$ occurs at this state, all three output transitions $(s_0, (prescribe\ rehabilitation, t_c), s_0)$, $(s_0, !(perform\ surgery, t_c), s_0)$ and $(s_0, -, s_1)$ can be triggered. A run of a finite trace on non-deterministic automata represents the execution of the trace on the automata and it transfers automata from one *set of possible states* to another *set of possible states*, as specified in Definition 5.3.4. We call such a run a non-deterministic run or *nd-run*.

Definition 5.3.4. (Non-deterministic run (*nd-run*))

Let $FSA = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be a finite state automaton and $\sigma \in \mathcal{E}^*$ be a trace. If finite sequence $\widetilde{FSA}(\sigma) = \langle (S_0, \sigma[1], S_1), (S_1, \sigma[2], S_2), \dots, (S_{|\sigma|-1}, \sigma[|\sigma|], S_{|\sigma|}) \rangle$ such that $\forall_{1 \leq i \leq |\sigma|} : S_i = \{s \in S \mid \exists s' \in S_{i-1} : (s', \sigma[i], s) \in T\}$ and $\forall_{1 \leq i \leq |\sigma|} : S_i \neq \emptyset$ exists, then we say that $\widetilde{FSA}(\sigma)$ is the *nd-run* of trace σ on the automaton FSA . We use \widetilde{FSA} to denote the set of all traces $\sigma \in \mathcal{E}^*$ such that $\widetilde{FSA}(\sigma)$ exists.

If it holds that $\sigma \in \widetilde{FSA}$, then we use S_σ^{FSA} to denote the last set of states in $\widetilde{FSA}(\sigma)$, i.e., $S_\sigma^{FSA} = S_{|\sigma|}$. \square

Table 5.5 shows the *nd-run* of a trace $\sigma \in \mathcal{E}^*$ containing eight events on the automaton FSA shown in Figure 5.12. The first row refers to the initial state of the automaton, i.e., the empty trace, and each row underneath this row refers to one element (i.e., event) of the trace. The first column contains events from the trace σ . The second column shows all transitions that can be triggered by the event given the current set of possible states (this set is shown in the third column of the previous row). Finally, the third column shows the new set of possible states after the event occurred. As Table 5.5 shows, the *nd-run* of trace σ starts in the set of initial states $S_0 = \{s_0\}$ and ends in the set of possible states $S_\sigma^{FSA} = \{s_0, s_1\}$.

Table 5.5: A non-deterministic run on the automaton FSA shown in Figure 5.12 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$)

examine=*examine patient*, surgery=*perform surgery*, rehabilitation=*prescribe rehabilitation*.

$\sigma \in \mathcal{E}^*$	transitions	new states
<i>initial state</i>		$S_0 = \{s_0\}$
$\sigma[1] = (\text{examine}, t_s)$	$(s_0, !(surgery, t_c), s_0)$ $(s_0, -, s_1)$	$S_1 = \{s_0, s_1\}$
$\sigma[2] = (\text{examine}, t_c)$	$(s_0, !(surgery, t_c), s_0)$ $(s_0, -, s_1)$ $(s_1, -, s_1)$	$S_2 = \{s_0, s_1\}$
$\sigma[3] = (\text{surgery}, t_s)$	$(s_0, !(surgery, t_c), s_0)$ $(s_0, -, s_1)$ $(s_1, -, s_1)$	$S_3 = \{s_0, s_1\}$
$\sigma[4] = (\text{surgery}, t_c)$	$(s_0, -, s_1)$ $(s_1, -, s_1)$	$S_4 = \{s_1\}$
$\sigma[5] = (\text{examine}, t_s)$	$(s_1, -, s_1)$	$S_5 = \{s_1\}$
$\sigma[6] = (\text{examine}, t_c)$	$(s_1, -, s_1)$	$S_6 = \{s_1\}$
$\sigma[7] = (\text{rehabilitation}, t_s)$	$(s_1, -, s_1)$	$S_7 = \{s_1\}$
$\sigma[8] = (\text{rehabilitation}, t_c)$	$(s_1, -, s_1)$ $(s_1, (\text{rehabilitation}, t_c), s_0)$	$S_\sigma^{FSA} = \{s_0, s_1\}$

If a trace brings the automata to an accepting state, then the automata accepts this trace. In case of a non-deterministic run, we say that the automata FSA accepts trace $\sigma \in \mathcal{E}^*$ if and only if the non-deterministic run of σ leaves

FSA in a set of possible states S such that at least one of the possible states is accepting, as specified in Definition 5.3.5.

Definition 5.3.5. (Acceptance)

Let $FSA = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be a finite state automaton and $\sigma \in \mathcal{E}^*$ be a trace. We say that FSA accepts trace σ if and only if it holds that $(\sigma \in \overline{FSA}) \wedge (S_\sigma^{FSA} \cap S_F \neq \emptyset)$. The language of FSA , $\mathcal{L}(FSA) \subseteq \mathcal{E}^*$, consists of all traces accepted by FSA . \square

5.3.3 Retrieving the Set of Satisfying Traces

If FSA_f is an automaton created for some well-formed LTL formula f , then the language of this automaton $\mathcal{L}(FSA_f)$ represents exactly all traces $\sigma \in \mathcal{E}^*$ that satisfy the formula f (i.e., $\sigma \models f$) [74, 111, 112, 158]. Hence all traces that satisfy the LTL formula f are represented by the automaton FSA_f generated from f (cf. Property 5.3.6).

Property 5.3.6. $(\forall \sigma \in \mathcal{E}^* : (\sigma \models f) \Leftrightarrow (\sigma \in \mathcal{L}(FSA_f)))$

Let f be a well formed LTL formula and $FSA_f = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be the automaton generated for formula f , then it holds that $\forall \sigma \in \mathcal{E}^* : (\sigma \models f) \Leftrightarrow (\sigma \in \mathcal{L}(FSA_f))$.

Proof. The automaton FSA construction in [111, 112, 158]. \square

Due to the fact that the language $\mathcal{L}(FSA_f)$ of automaton FSA_f generated from the LTL formula f of constraint $(E, f) \in \mathcal{C}$ accepts exactly the traces that satisfy the LTL formula f , it holds that the automaton FSA_f represents the set of traces that satisfy the constraint (E, f) , as shown in Property 5.3.7.

Property 5.3.7. $(\mathcal{E}_{\models(E,f)}^* = \mathcal{L}(FSA_f))$

Let $c \in \mathcal{C}$ be a constraint where $c = (E, f)$ such that f is a well formed LTL formula over E . If $FSA_f = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ is an automaton generated for formula f , then it holds that $\mathcal{E}_{\models c}^* = \mathcal{L}(FSA_f)$.

Proof. It holds that $\forall \sigma \in \mathcal{E}^* : (\sigma \models f) \Leftrightarrow (\sigma \in \mathcal{L}(FSA_f))$ (cf. Property 5.3.6), i.e., $\forall \sigma \in \mathcal{E}^* : (f(\sigma) = \mathbf{true}) \Leftrightarrow (\sigma \in \mathcal{L}(FSA_f))$ (cf. Definition 5.1.1). It also holds that $\mathcal{E}_{\models c}^* = \{\sigma \in \mathcal{E}^* \mid \sigma \models c\}$ where $(\sigma \models c) \Leftrightarrow f(\sigma) = \mathbf{true}$ (cf. Definition 4.1.4). Therefore, it holds that $\mathcal{E}_{\models(E,f)}^* = \mathcal{L}(FSA_f)$. \square

Automata generated from LTL formulas are consistent with the two requirements of sets of accepting traces for constraints (and constraint models). First, note that, although it is built for constraint on two particular activities (i.e., $E = \{(perform\ surgery, t_c), (prescribe\ rehabilitation, t_c)\}$), the automaton in Figure 5.12 can run traces that can contain events involving activities other than *perform surgery* and *prescribe rehabilitation*, i.e., it can contain events $e \notin E$ that are not in the namespace. Transitions $(s_0, !(surgery, t_c), s_0)$, $(s_0, -, s_1)$ and

$(s_1, -, s_1)$ can trigger other events like, for example, events $(examine, t_s)$ and $(examine, t_c)$. Indeed, a trace constraining these two events can run on the automaton, as shown in Table 5.5. Moreover, these two events can be replaced by any other event not involving activities *perform surgery* and *prescribe rehabilitation*, e.g., by events $(drink\ coffee, t_s)$ and $(drink\ coffee, t_c)$. As explained in Chapter 4, this is an important property of sets of accepting traces that enables easy ad-hoc change (cf. Section 4.5).

Second, trace σ from Table 5.5 is accepted by the automaton FSA_f in Figure 5.12 for constraint $c = (E, f)$ in Figure 5.11 because it leaves the automata in the set of possible traces $S_\sigma^{FSA} = \{s_0, s_1\}$ where s_0 is an accepting state. Trace σ contains many events that are not in the namespace E of the constraint. In fact, if any of the events in σ that are not in the namespace $e \notin E$ would be replaced by other events not in the name space $e' \notin E$, the automaton would still accept such changed trace. This is in line with the requirement of Definition 4.1.4 on page 87, which says that if a trace σ satisfies the constants, then all traces from the projection $\sigma \downarrow^E$ satisfy the constraint and vice versa.

Figure 5.13 shows another example of a ConDec constraint $c = (E, f)$. This constraint is created from the *precedence* template (cf. Section 5.2.2), and, therefore, it has the template's graphical representation and LTL formula. This constraint specifies that activity *perform surgery* cannot be executed before activity *perform X ray*, i.e., formula f specifies that events $(perform\ surgery, t_s)$ and $(perform\ surgery, t_c)$ cannot occur before event $(perform\ X\ ray, t_c)$.

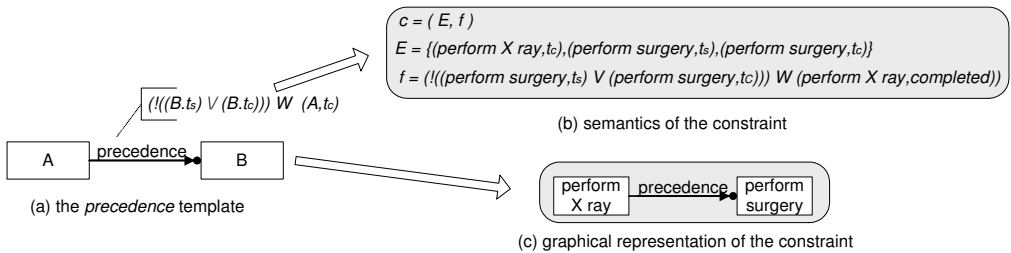


Figure 5.13: An example of a *precedence* constraint (recall that $t_s, t_c \in \mathcal{J}$ are event types such that $t_s = started$ and $t_c = completed$)

Figure 5.14 shows the finite state automaton FSA_f generated from the LTL formula for the precedence constraint c in Figure 5.13. Language $\mathcal{L}(FSA_f)$ of the automaton represents exactly all traces that satisfy constraint c . Indeed, the automaton prevents occurrence of events $(perform\ surgery, t_s)$ and $(perform\ surgery, t_c)$ before event $(perform\ X\ ray, t_c)$ since transition $(s_1, -, s_1)$ is the only transition that can trigger $(perform\ surgery, t_s)$ and $(perform\ surgery, t_c)$ and state s_1 can be reached only via transition $(s, (perform\ X\ ray, t_c), s_1)$.

For illustration, in Table 5.6 we present non-deterministic runs (*nd-run*) of two traces on the finite state automaton FSA_f in Figure 5.14, which is generated for the constraint $c = (E, f)$ in Figure 5.13. Each trace starts at

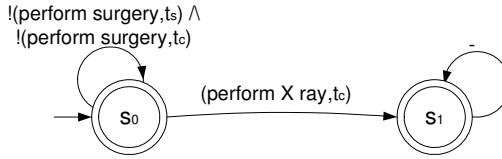


Figure 5.14: A finite state automaton FSA_f for constraint in Figure 5.13 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

the *initial state* of the automaton. Further, for each event in the trace the new set of possible states of the automaton is given. Trace σ_1 is accepted by the automaton FSA_f and, therefore, σ_1 satisfies constraint from Figure 5.13, i.e., events $(perform\ surgery, t_s)$ and $(perform\ surgery, t_c)$ are preceded by event $(perform\ X\ ray, t_c)$ in σ_1 . Trace σ_2 is not accepted by the automaton FSA_f (i.e., it is not possible to trigger event $(perform\ surgery, t_s)$ from state s_0) and, therefore, σ_2 does not satisfy constraint from Figure 5.13, i.e., event $(perform\ surgery, t_s)$ is not preceded by event $(perform\ X\ ray, t_c)$ in σ_2 .

Table 5.6: Non-deterministic runs of two traces on automaton in Figure 5.14 generated for the constraint $c = (E, f)$ in Figure 5.13 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

examine=*examine patient*, surgery=*perform surgery*, X ray=*perform X ray*.

	$\sigma_1 \in \mathcal{E}_{\neq c}^*$		$\sigma_2 \notin \mathcal{E}_{\neq c}^*$	
i	$\sigma_1[i]$	new states	$\sigma_2[i]$	new states
	<i>initial state</i>	$S_0 = \{s_0\}$	<i>initial state</i>	$S_0 = \{s_0\}$
1	$(examine, t_s)$	$S_1 = \{s_0\}$	$(examine, t_s)$	$S_1 = \{s_0\}$
2	$(examine, t_c)$	$S_2 = \{s_0\}$	$(examine, t_c)$	$S_2 = \{s_0\}$
3	$(X\ ray, t_s)$	$S_3 = \{s_0\}$	$(surgery, t_s)$	$\{\}$
4	$(X\ ray, t_s)$	$S_4 = \{s_0, s_1\}$		
5	$(examine, t_s)$	$S_5 = \{s_0, s_1\}$		
6	$(examine, t_c)$	$S_6 = \{s_0, s_1\}$		
7	$(surgery, t_s)$	$S_7 = \{s_1\}$		
8	$(surgery, t_c)$	$S_{\sigma_1}^{FSA} = \{s_1\}$		

5.4 ConDec Models

ConDec constraint models are presented graphically to users: (1) activities are presented as labeled rectangles and (2) ConDec constraints are presented as graphical representations of their templates, i.e., special lines and symbols between activities. Figure 5.15 shows one ConDec model consisting of three activities (i.e., *curse*, *pray* and *bless*) and two mandatory constraints. The constraint between activities *curse* and *pray* is based in the *response* template, i.e., it specifies that each occurrence of event $(curse, t_c)$ has to be eventually followed by

at least one occurrence of event $(pray, t_c)$. The constraint on the activity $pray$ is based on the $1..*$ template, i.e., it specifies that there has to be at least one occurrence of event $(pray, t_c)$. As Figure 5.15 shows, LTL formulas are associated to constraints but they are hidden in the model, i.e., each constraint is represented as the graphical representation of its template. Note that both constraints in Figure 5.15 are *mandatory*, i.e., they are represented as *full lines*. *Optional* constraints are also represented as graphical representations of their templates, but as *dashed lines*. Further in this chapter, we will show a ConDec model with an optional constraint.

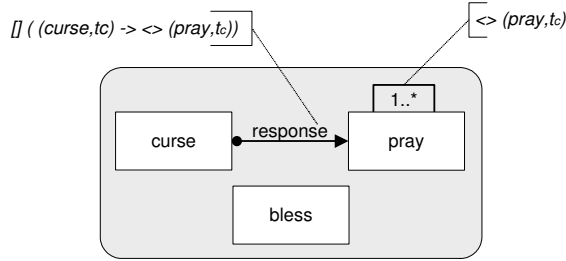


Figure 5.15: A ConDec model (recall that $t_c \in \mathcal{T}$ is an event type such that $t_c = completed$)

Due to the fact that all mandatory and optional constraints in a ConDec model are ConDec constraints, i.e., *LTL constraints*, every ConDec model is an *LTL constraint model*. The set of satisfying traces of any LTL constraint model (e.g., a ConDec model) is determined based on the LTL formulas associated with mandatory constraints from the model. The mandatory formula of such a model is defined as a conjunction of LTL formulas for all mandatory constraints, as specified in Definition 5.4.1. For example, the mandatory formula for ConDec model cm in Figure 5.15 is $f_{cm} = (\Box((curse, t_c) \Rightarrow \Diamond(pray, t_c))) \wedge (\Diamond(pray, t_c))$.

Definition 5.4.1. (Mandatory formula f_{cm})

Let $cm \in \mathcal{U}_{cm}$ be a constraint model where $cm = (A, C_M, C_O)$ such that all mandatory and optional constraints are LTL constraints, then the mandatory formula for model cm is defined as

$$f_{cm} = \begin{cases} \mathbf{true} & \text{if } C_M = \emptyset; \\ \bigwedge_{(E,f) \in C_M} f & \text{otherwise.} \end{cases}$$

□

Because the mandatory formula of a model corresponds to the conjunction of all mandatory constraints, the language of the automaton generated for this formula represents the set of satisfying traces for the model, as shown by Property 5.4.2.

Property 5.4.2. ($\mathcal{L}_{=cm}^* = \mathcal{L}(FSA_{f_{cm}})$)

Let $cm \in \mathcal{U}_{cm}$ be a ConDec constraint model such that $cm = (A, C_M, C_O)$ and

$FSA_{f_{cm}}$ be a finite state automaton such that f_{cm} is a mandatory formula for cm , then $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}})$.

Proof. If it holds that $C_M = \emptyset$, then it holds that $f_{cm} = \mathbf{true}$ (cf. Definition 5.4.1). Therefore, it holds that $\forall \sigma \in \mathcal{E}^* : \sigma \models f_{cm}$ (cf. operator \mathbf{true} in Definition 5.1.1), i.e., $\mathcal{L}(FSA_{f_{cm}}) = \mathcal{E}^*$ (cf. Property 5.3.6). Further, because it holds that $C_M = \emptyset$, it also holds that $\mathcal{E}_{\models cm}^* = \mathcal{E}^*$ (cf. Definition 4.2.2 on page 90). Therefore, it holds that $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}}) = \mathcal{E}^*$.

If it holds that $C_M \neq \emptyset$, then, on the one hand, it holds that $f_{cm} = \bigwedge_{(E,f) \in C_M} f$ (cf. Definition 5.4.1). Therefore, it also holds that $(\sigma \in \mathcal{L}(FSA_{f_{cm}})) \Leftrightarrow (\forall (E,f) \in C_M : f(\sigma) = \mathbf{true})$ (cf. operator \wedge in Definition 5.1.1 and Definition 5.3.5). On the other hand, it holds that $(\sigma \in \mathcal{E}_{\models cm}^*) \Leftrightarrow (\forall (E,f) \in C_M : f(\sigma) = \mathbf{true})$ (cf. definitions 4.1.4 on page 87 and 4.2.2 on page 90). Therefore, it holds that $(\sigma \in \mathcal{L}(FSA_{f_{cm}})) \Leftrightarrow (\sigma \in \mathcal{E}_{\models cm}^*)$, i.e., it holds that $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}})$. \square

The automaton $FSA_{f_{cm}}$ generated for the mandatory formula f_{cm} for an LTL constraint model $cm \in \mathcal{U}_{cm}$ (e.g., a ConDec model) is a *finite representation* of the set of satisfying traces $\mathcal{E}_{\models cm}^*$ of the model cm . This approach makes checking if some trace $\sigma \in \mathcal{E}^*$ satisfies model cm a trivial operation – it is enough to check if automaton $FSA_{f_{cm}}$ accepts trace σ . Figure 5.16 shows the automaton $FSA_{f_{cm}}$ generated for the mandatory formula $f_{cm} = (\square((curse, t_c) \Rightarrow \diamond(pray, t_c))) \wedge (\diamond(pray, t_c))$ of the ConDec model cm in Figure 5.15.

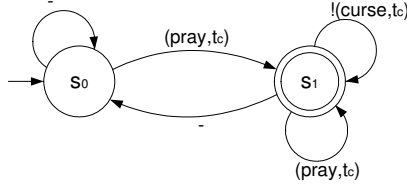


Figure 5.16: Automaton $FSA_{f_{cm}}$ that accepts the satisfying traces $\mathcal{E}_{\models cm}^*$ for ConDec model cm in Figure 5.15 (recall that $t_c \in \mathcal{T}$ is an event type such that $t_c = \mathit{completed}$)

The automaton in Figure 5.16 has one initial (i.e., s_0) and one accepting (i.e., s_1) state. The language of this automaton represents all traces that satisfy the ConDec model cm in Figure 5.15, i.e., it represents all traces that satisfy both mandatory constraints from the model. First, it is clear that only traces that contain at least one occurrence of event $(pray, t_c)$ can satisfy cm , because the accepting set of states $\{s_0, s_1\}$ can be reached if and only if event $(pray, t_c)$ occurs. Therefore, only traces that satisfy constraint $1..^*$ can satisfy this model. Second, if event $(curse, t_c)$ occurs, the automaton's state is a non-accepting set of states (i.e., $\{s_0\}$), and the automaton will remain in this set of states until the first occurrence of event $(pray, t_c)$. In other words, only traces that satisfy constraint *response* can satisfy this model.

Table 5.7 shows three non-deterministic runs on the automaton $FSA_{f_{cm}}$ shown in Figure 5.16, which is generated for the mandatory formula of the model cm in Figure 5.15. On the one hand, traces σ_1 and σ_2 are accepted by this automaton, and, therefore, these traces satisfy the model cm . In these two traces each occurrence of event $(curse, t_c)$ is followed by an occurrence of event $(pray, t_c)$ (i.e., constraint *response* is satisfied) and there is one occurrence of event $(pray, t_c)$ (i.e., constraint $1..*$ is satisfied). Therefore, traces σ_1 and σ_2 satisfy the model cm . On the other hand, trace σ_3 is not accepted by the automaton, and, therefore, this trace does not satisfy the model cm . The occurrence of event $(curse, t_c)$ is not followed by an occurrence of event $(pray, t_c)$ in trace σ_3 (i.e., constraint *response* is not satisfied) and there is one occurrence of event $(pray, t_c)$ (i.e., constraint $1..*$ is satisfied). Therefore, trace σ_3 does not satisfy the model cm .

Table 5.7: Examples of (non-)accepting traces for automaton $FSA_{f_{cm}}$ in Figure 5.16 generated for model cm in Figure 5.15 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

	$\sigma_1 \in \mathcal{E}_{\models cm}^*$		$\sigma_2 \in \mathcal{E}_{\models cm}^*$		$\sigma_3 \notin \mathcal{E}_{\models cm}^*$	
i	$\sigma_1[i]$	new states	$\sigma_2[i]$	new states	$\sigma_3[i]$	new states
	<i>initial</i>	$\{s_0\}$	<i>initial</i>	$\{s_0\}$	<i>initial</i>	$\{s_0\}$
1	$(bless, t_s)$	$\{s_0\}$	$(bless, t_s)$	$\{s_0\}$	$(pray, t_s)$	$\{s_0, s_1\}$
2	$(bless, t_c)$	$\{s_0\}$	$(bless, t_c)$	$\{s_0\}$	$(pray, t_c)$	$\{s_0, s_1\}$
3	$(curse, t_s)$	$\{s_0\}$	$(curse, t_s)$	$\{s_0\}$	$(curse, t_s)$	$\{s_0\}$
4	$(curse, t_c)$	$\{s_0\}$	$(curse, t_c)$	$\{s_0\}$	$(curse, t_c)$	$\{s_0\}$
5	$(bless, t_s)$	$\{s_0\}$	$(become\ holy, t_s)$	$\{s_0\}$	$(bless, t_s)$	$\{s_0\}$
6	$(bless, t_c)$	$\{s_0\}$	$(become\ holy, t_c)$	$\{s_0\}$	$(bless, t_c)$	$\{s_0\}$
7	$(curse, t_s)$	$\{s_0\}$	$(curse, t_s)$	$\{s_0\}$		
8	$(curse, t_c)$	$\{s_0\}$	$(curse, t_c)$	$\{s_0\}$		
9	$(pray, t_s)$	$\{s_0\}$	$(pray, t_s)$	$\{s_0\}$		
10	$(pray, t_c)$	$\{s_0, s_1\}$	$(pray, t_c)$	$\{s_0, s_1\}$		

Note that trace σ_2 is accepted by FSA_{cm} (and satisfies cm) although σ_2 contains events $(become\ holy, t_s)$ and $(become\ holy, t_c)$, i.e., it contains events on activities that are not in the model cm (cf. Figure 5.15). As discussed in Chapter 4, this is a desirable property of constraint languages, which facilitates ad-hoc change. For example, if activity *become holy* was originally in the model, it possible that it was executed and later removed from the model.

5.5 ConDec Model: Fractures Treatment Process

By using graphical notations of templates for representation of constraints, ConDec models can be easily represented to people not familiar with LTL. Figure 5.17 shows a ConDec model cm^{FT} for the Fractures Treatment process (cf. Example 4.3.2 on page 96). Labels c_1, \dots, c_6 mark constraints from the model cm^{FT} .

All six constraints are created from ConDec templates. The first five constraints are mandatory, i.e., presented as *full lines*. The sixth constraint is optional and, therefore, presented as a *dashed line*. Constraint c_4 is created from the *precedence* template, but it is branched on several activities. This means that, although the *precedence* template has two parameters A and B , in this model we replace parameter A with activity *perform X ray* and we branch parameter B with activities *apply cast*, *perform reposition* and *perform surgery*. By doing this, we specify that none of the activities *apply cast*, *perform reposition* or *perform surgery* can be executed before activity *perform X ray* is executed.

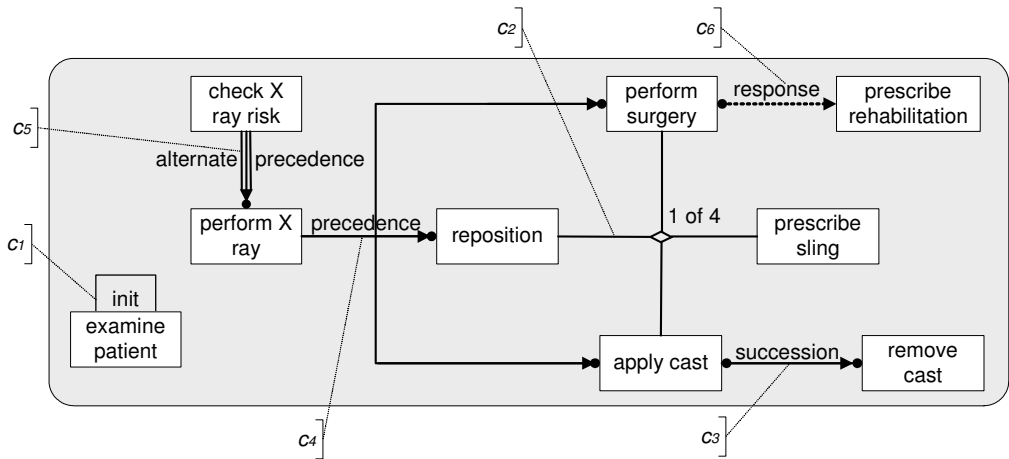


Figure 5.17: ConDec model for the Fractures Treatment

Although the ConDec model in Figure 5.17 presents constraints graphically, each of these constraints has a hidden LTL formula that is inherited from the constraint's template. As we explained before, when a constraint is created between activities in a ConDec model, then these activities replace the formal parameters in the template. Constraint c_1 is created from the *init* existence template (cf. Section 5.2.1) and constraint c_2 is based on the *1 of 4* choice template (cf. Section 5.2.4). Constraints c_3 , c_4 , c_5 and c_6 are created from *succession*, *precedence*, *alternate precedence* and *response* relation templates, respectively (cf. Section 5.2.2). Table 5.8 shows LTL formulas for all constraints in the ConDec model in Figure 5.17. Constraint c_4 is a special case, i.e., here the second template parameter B is branched on activities *apply cast*, *perform reposition* and *perform surgery*, i.e., event (B, t_s) is replaced with the disjunction of events $(\text{apply cast}, t_s)$, $(\text{perform reposition}, t_s)$ and $(\text{perform surgery}, t_s)$ and (B, t_c) is replaced with the disjunction of events $(\text{apply cast}, t_c)$, $(\text{perform reposition}, t_c)$ and $(\text{perform surgery}, t_c)$.

Table 5.8: LTL formulas for constraints in ConDec model in Figure 5.17 (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$)

constraint c_i	LTL formula f_i
$c_1 = (E_1, f_1)$	$f_1 = ((\text{examine patient}, t_s) \vee (\text{examine patient}, t_x))$ $U(\text{examine patient}, t_c)$
$c_2 = (E_2, f_2)$	$f_2 = (\diamond(\text{apply cast}, t_c) \vee (\diamond(\text{prescribe sling}, t_c))$ $\vee (\diamond(\text{perform reposition}, t_c) \vee (\diamond(\text{perform surgery}, t_c)))$
$c_3 = (E_3, f_3)$	$f_3 = (\square(\text{apply cast}, t_c) \Rightarrow \diamond(\text{remove cast}, t_c))$ $\wedge !(((\text{remove cast}, t_s) \vee (\text{remove cast}, t_c))W(\text{apply cast}, t_c))$
$c_4 = (E_4, f_4)$	$f_4 = !(((\text{apply cast}, t_s) \vee (\text{perform reposition}, t_s)$ $\vee (\text{perform surgery}, t_s)) \vee ((\text{apply cast}, t_c) \vee (\text{perform reposition}, t_c)$ $\vee (\text{perform surgery}, t_c)))W(\text{perform X ray}, t_c)$
$c_5 = (E_5, f_5)$	$f_5 = (((\text{perform X ray}, t_s) \vee (\text{perform X ray}, t_c))$ $W(\text{check X ray risk}, t_c) \wedge \square(\text{perform X ray}, t_c) \Rightarrow$ $\circ(!((\text{perform X ray}, t_s) \vee (\text{perform X ray}, t_c))$ $W(\text{check X ray risk}, t_c))))$
$c_6 = (E_6, f_6)$	$f_6 = (\square(\text{perform surgery}, t_c) \Rightarrow \diamond(\text{prescribe rehabilitation}, t_c))$

5.6 Execution of ConDec Instances

Execution of instances of LTL constraint models, and therefore ConDec instances, is based on the LTL specifications of constraints and, more precisely, the automaton generated from the mandatory formula of the model. In sections 5.6.1, 5.6.2, and 5.6.3 we show how this automaton is used to determine the state of the instance (cf. Section 4.4.1), enabled events (cf. Section 4.4.2), and states of constraints (cf. Section 4.4.3), respectively.

5.6.1 Instance State

The automaton $FSA_{f_{cm}}$ generated for the mandatory formula of instance's model and the instance trace σ is used to easily determine the state of an instance $ci = (\sigma, cm)$ at any moment of execution. Property 5.6.1 shows that the instance is *satisfied* if and only if the trace σ is accepted by the automaton $FSA_{f_{cm}}$. In other words, if the *nd*-run of σ on $FSA_{f_{cm}}$ leaves the $FSA_{f_{cm}}$ in a set of possible states that contains an accepting state, then the instance state is *satisfied*. Recall that $\omega(ci) = \text{satisfied}$ if $\sigma \in \mathcal{E}_{\models cm}^*$ for some instance $ci(\sigma, cm)$ (cf. Definition 4.4.2).

Property 5.6.1. ($\omega((\sigma, cm)) = \text{satisfied}$ if and only if $\sigma \in \mathcal{L}(FSA_{f_{cm}})$)

Let $ci \in \mathcal{U}_{ci}$ be an instance of a LTL constraint model $cm \in \mathcal{U}_{cm}$ where $ci = (\sigma, cm)$ and let $FSA_{f_{cm}}$ be the automaton generated for the mandatory formula f_{cm} . It holds that $\omega(ci) = \text{satisfied}$ if and only if it holds that $\sigma \in \mathcal{L}(FSA_{f_{cm}})$.

Proof. Since $(\omega(ci) = \text{satisfied}) \Leftrightarrow (\sigma \in \mathcal{E}_{\models cm}^*)$ (cf. Definition 4.4.2) and $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}})$ (cf. Property 5.4.2), we know that $(\omega(ci) = \text{satisfied}) \Leftrightarrow (\sigma \in \mathcal{L}(FSA_{f_{cm}}))$. \square

An instance $ci = (\sigma, cm)$ is in the *temporarily violated* state if σ is not accepted by the automaton $FSA_{f_{cm}}$ generated for the mandatory formula but the trace σ can be run on $FSA_{f_{cm}}$, as Property 5.6.2 shows. In other words, if the *nd*-run of σ on $FSA_{f_{cm}}$ leaves the $FSA_{f_{cm}}$ in a set of possible states that does not contain an accepting state, then the instance state is *temporarily violated*.

Property 5.6.2. $(\omega(\overline{(\sigma, cm)})) = \text{temporarily violated}$ if and only if $\sigma \notin \mathcal{L}(FSA_{f_{cm}})$ and $\sigma \in \overline{FSA_{f_{cm}}}$

Let $ci \in \mathcal{U}_{ci}$ be an instance of a LTL constraint model $cm \in \mathcal{U}_{cm}$ where $ci = (\sigma, cm)$. Let $FSA_{f_{cm}} = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be the automaton generated for f_{cm} and $\overline{FSA_{f_{cm}}}(\sigma)$ be the non-deterministic run of σ on $FSA_{f_{cm}}$. It holds that $\omega(ci) = \text{temporarily violated}$ if and only if it holds that $\sigma \notin \mathcal{L}(FSA_{f_{cm}})$ and $\sigma \in \overline{FSA_{f_{cm}}}$ ².

Proof. If it holds that $\sigma \notin \mathcal{L}(FSA_{f_{cm}})$, then it holds that $\sigma \notin \mathcal{E}_{\models cm}^*$ because $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}})$ (cf. Property 5.4.2).

If it holds that $\sigma \in \overline{FSA_{f_{cm}}}$, then it holds that $S_{\sigma}^{FSA_{f_{cm}}} \neq \emptyset$ and it holds that $\forall s \in S_{\sigma}^{FSA_{f_{cm}}} : R_s^{FSA_{f_{cm}}} \cap S_F \neq \emptyset$ (cf. Definition 5.3.3). Therefore, it holds that $\exists \gamma \in \mathcal{E}^*$ such that automaton $FSA_{f_{cm}}$ accepts trace $\sigma + \gamma$ (cf. Definitions 5.3.5 and 5.3.5), i.e., $\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{L}(FSA_{f_{cm}})$ and it holds that $\exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*$ (cf. Property 5.4.2).

Because it holds that $\sigma \notin \mathcal{E}_{\models cm}^* \wedge \exists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*$, it also holds that $\omega(ci) = \text{temporarily violated}$. \square

An instance $ci = (\sigma, cm)$ is in the *violated* state if σ cannot be run on the automaton $FSA_{f_{cm}}$ generated for the mandatory formula, as shown in Property 5.6.3. In other words, if the *nd*-run of σ on $FSA_{f_{cm}}$ does not exist, then the instance state is *violated*.

Property 5.6.3. $(\omega(\overline{(\sigma, cm)})) = \text{violated}$ if and only if $\sigma \notin \overline{FSA_{f_{cm}}}$

Let $ci \in \mathcal{U}_{ci}$ be an instance of a LTL constraint model $cm \in \mathcal{U}_{cm}$ where $ci = (\sigma, cm)$. Let $FSA_{f_{cm}}$ be the automaton generated for f_{cm} and $\overline{FSA_{f_{cm}}}(\sigma)$ be the non-deterministic run of σ on $FSA_{f_{cm}}$. It holds that $\omega(ci) = \text{temporarily violated}$ if and only if it holds that $\sigma \notin \overline{FSA_{f_{cm}}}$

Proof. If it holds that $\sigma \notin \overline{FSA_{f_{cm}}}$, then it holds that $\sigma \notin \mathcal{L}(FSA_{f_{cm}})$ and $\nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in FSA_{f_{cm}}$ (cf. definitions 5.3.4 and 5.3.5). Therefore, it also holds that $\sigma \notin \mathcal{E}_{\models cm}^*$ and $\nexists \gamma \in \mathcal{E}^* : \sigma + \gamma \in \mathcal{E}_{\models cm}^*$, i.e., $\omega(ci) = \text{temporarily violated}$. \square

Figure 5.18 shows (a) a ConDec model cm and (b) the automaton created for the mandatory formula f_{cm} of the model cm . The model has four

²Recall that $\overline{FSA_{f_{cm}}}$ uses the unblocked automaton from which “dead ends” are removed (cf. definitions 5.3.2 and 5.3.3)

activities and three constraints. Constraint *response* specifies that every occurrence of event $(curse, t_c)$ has to be eventually followed by at least one occurrence of event $(pray, t_c)$. Constraint $1..*$ specifies that $(pray, t_c)$ has to occur at least once. Constraint *precedence* specifies that events $(become\ holy, t_s)$ and $(become\ holy, t_c)$ cannot occur before the first occurrence of event $(pray, t_c)$. In other words, constraints in this model specify that (1) if one *curse*s, then one has to *pray* at least once afterwards, (2) one has to *pray* at least once, and (3) one cannot *become holy* until one has *prayed*. Note that automaton in Figure 5.18(b) is a simplified version of the original automaton generated for f_{cm} using the algorithm presented in [112,113]: for simplicity we show a smaller automaton with the same language as the language of the original automaton.

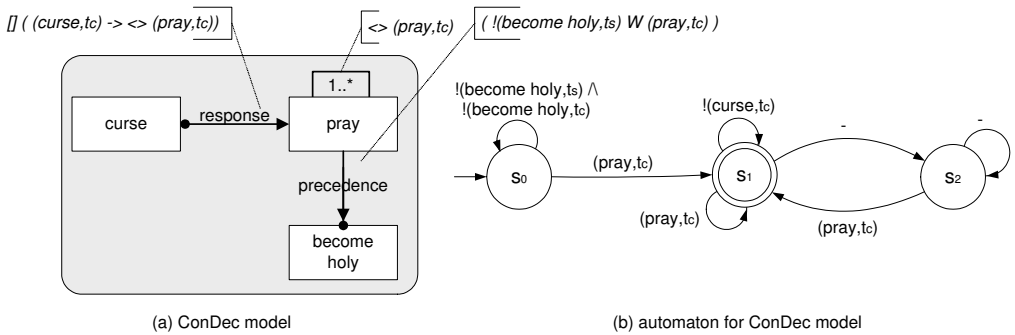


Figure 5.18: A ConDec model (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

Figure 5.19 shows how a *satisfied* instance $ci = (\sigma, cm)$ of the ConDec model cm in Figure 5.18(a) changes states depending on its execution on the automaton $FSA_{f_{cm}}$ (cf. Figure 5.18(b)). This instance is correct because its state is never *violated*. Initially, the automaton is in the set of possible states $\{s_0\}$ and, therefore the instance is *temporarily violated*. The state of the instance is initially *temporarily violated* because none of the possible states is accepting (i.e., s_0 is not accepting) but an accepting state (i.e., s_1) is reachable from one of the possible states (i.e., s_1 is reachable from s_0). On the other hand, the instance state is *satisfied* every time when the set of possible states contains at least one accepting state (i.e., s_1). The states of the instance in Figure 5.19 reflect the mandatory constraint of the model in Figure 5.18. The instance is *temporarily violated* until the first occurrence of event $(pray, t_c)$ (the $1..*$ constraint). The instance becomes *temporarily violated* again after the occurrence of event $(curse, t_c)$ and becomes *satisfied* only after a new occurrence event $(pray, t_c)$ (the *response* constraint). The *precedence* constraint is fulfilled because events $(become\ holy, t_s)$ and $(become\ holy, t_c)$ occur only after event $(pray, t_c)$.

Figure 5.20 shows how a *violated* instance $ci = (\sigma, cm)$ of the ConDec model cm in Figure 5.18(a) changes states depending on its execution on the automaton

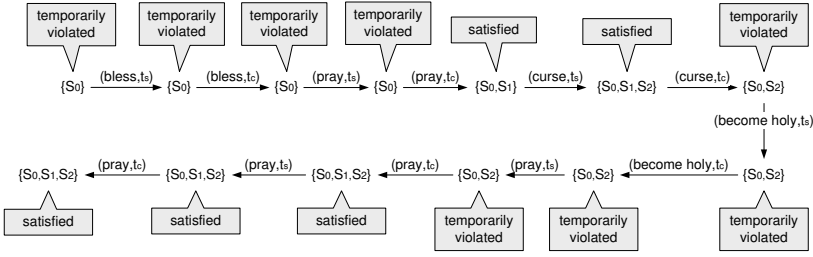


Figure 5.19: A *satisfied* instance of the model in Figure 5.18 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \textit{started}$ and $t_c = \textit{completed}$)

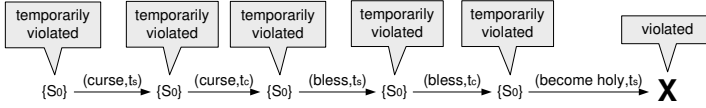


Figure 5.20: A *violated* instance of the model in Figure 5.18 (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \textit{started}$ and $t_c = \textit{completed}$)

$FSA_{f_{cm}}$ (cf. Figure 5.18(b)). Execution of event $(\textit{become holy}, t_s)$ brings this instance to the *violated* state, because there is no non-deterministic run of this trace on $FSA_{f_{cm}}$. Therefore, the trace is not accepted by $FSA_{f_{cm}}$ and an accepting state is not reachable anymore. The reason for the violation is the fact that the precedence constraint cannot be fulfilled in the future, i.e., $\textit{become holy}$ is executed before the activity \textit{pray} .

5.6.2 Enabled Events

Enabled events are events that can be triggered during the executing of an instance of a constraint model, such that the instance does not become *violated*, as described in Section 4.4.2. Event $e \in A \times \mathcal{T}$ is enabled in instance $ci = (\sigma, cm)$ of an LTL constraint model (e.g., a ConDec model) if in the current set of possible states of the automaton generated for the mandatory formula f_{cm} there exists an output transition that can be triggered by the event e , as shown in Property 5.6.4. Consider, for example, the model and its automaton in Figure 5.18. Event $(\textit{become holy}, t_s)$ is not enabled in instances of this model as long as the instance's automaton stays in the set of possible states $\{s_0\}$ because none of the transitions $\neg(\textit{become holy}, t_s) \wedge \neg(\textit{become holy}, t_c)$ or (\textit{pray}, t_c) can trigger event $(\textit{become holy}, t_s)$. On the other hand, all other events involving activities in the model can be triggered via one or both transitions.

Property 5.6.4. (Enabled event)

Let $ci \in \mathcal{U}_{ci}$ be an instance of a LTL constraint model $cm \in \mathcal{U}_{cm}$ where $ci = (\sigma, cm)$, $cm = (A, C_M, C_O)$. Let $FSA_{f_{cm}} = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be the automaton generated for f_{cm} . Event $e \in A \times \mathcal{J}$ is enabled in ci (denoted by $ci[e]$) if and only if it holds that $\sigma \in \widetilde{FSA}_{f_{cm}}$ and $\exists s \in S_{\sigma}^{FSA_{f_{cm}}} \exists s' \in S : (s, e, s') \in T$.

Proof. If it holds that $(s, e, s') \in T$, then it holds that $\sigma + \langle e \rangle \in \widetilde{FSA}_{f_{cm}}$ (cf. definitions 5.3.3 and 5.3.4). Then, it either holds that $\sigma + \langle e \rangle \in \mathcal{L}(FSA_{f_{cm}})$, i.e., $\omega((\sigma + \langle e \rangle), cm) = \textit{satisfied}$ (cf. Property 5.6.1), or it holds that $\sigma + \langle e \rangle \notin \mathcal{L}(FSA_{f_{cm}})$, i.e., $\omega((\sigma + \langle e \rangle), cm) = \textit{temporarily violated}$ (cf. Property 5.6.2). In other words, it holds that $\omega((\sigma + \langle e \rangle), cm) \neq \textit{violated}$ and it holds that $ci[e]$. \square

5.6.3 States of Constraints

States of constraints in an instance can provide useful information for users who are executing the instance (cf. Section 4.4.3). Properties 5.6.1, 5.6.2 and 5.6.3 can be used to monitor states of all constraints in an instance $ci(\sigma, cm)$: for LTL formula of each constraint an automaton is created and analyzed given the trace σ to determine the state of the constraint. This method for monitoring states of constraints is used in our DECLARE system presented in Chapter 6.

5.7 Ad-hoc Change of ConDec Instances

As we discussed in Section 4.5, an ad-hoc change of constraint instances is successful if the change does not bring the instance in the *violated* state (cf. Figure 4.4). Automata generated for ConDec models enable easy implementation of ad-hoc change of ConDec instances. As shown in Property 5.7.1, ad-hoc change of a ConDec instance is successful if the instance trace can be ‘replayed’ on the mandatory automaton of the new model. Recall that Δ is the ad-hoc instance change function (cf. Definition 4.5.1 on page 107). Ad-hoc change of an instance ci to model cm is successful (i.e., $(ci, cm') \in \textit{dom}(\Delta)$) if and only if the changed model does not bring the instance into the *violated* state.

Property 5.7.1. (Instance (σ, cm) is successfully changed to (σ, cm') if and only if $\sigma \in \widetilde{FSA}_{f_{cm'}}$)

Let $ci \in \mathcal{U}_{ci}$ be an instance of a LTL constraint model where $ci = (\sigma, cm)$ and let $cm' \in \mathcal{U}_{cm}$ be a constraint model. Let $FSA_{f_{cm'}}$ be the automaton generated for $f_{cm'}$. It holds that $((\sigma, cm), cm') \in \textit{dom}(\Delta)$ if and only if it holds that $\sigma \in \widetilde{FSA}_{f_{cm'}}$.

Proof. If it holds that $\sigma \in \widetilde{FSA}_{f_{cm'}}$, then it holds that $\omega((\sigma, cm)) \neq \textit{violated}$ (cf. Property 5.6.3). Therefore, it holds that $((\sigma, cm), cm') \in \textit{dom}(\Delta)$ (cf. Definition 4.5.1). \square

Consider, for example, a *temporarily violated* instance $ci = (\sigma, cm)$ in Figure 5.21. The instance model cm is shown in Figure 5.21(a). This model consists of several activities and a *response* constraint between the activities *curse* and *pray*. This constraint specifies that, if *curse* is completed, then *pray* has to be also completed afterwards. The automaton $FSA_{f_{cm}}$ generated for the mandatory formula of this model is shown in Figure 5.21(b). Figure 5.21 (c) shows the instance trace σ , i.e., it shows the *nd*-run of the trace on the automaton in Figure 5.21(b) and it shows the corresponding instance states. Next, we will show two examples of ad-hoc change of the instance in Figure 5.21 – one example of a successful ad-hoc change and one example of an impossible (i.e., unsuccessful) ad-hoc change.

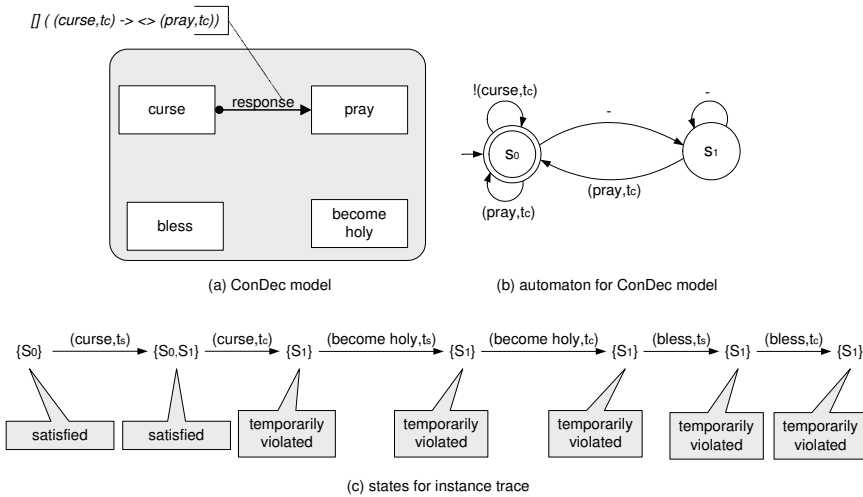


Figure 5.21: A ConDec instance $ci = (\sigma, cm)$ (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$)

Figure 5.22 shows an example of a successful ad-hoc change of the instance $ci = (\sigma, cm)$ in Figure 5.21. Figure 5.22(a) shows the new model cm_S for the instance ci : activity *bless* is removed from the original instance model instance and constraint $1..*$ is added to the original instance model (cf. Figure 5.21(a)). The new $1..*$ constraint specifies that activity *pray* has to be executed at least once. Figure 5.22(b) shows the automaton $FSA_{f_{cm_S}}$ generated for the mandatory formula of the new model cm_S . Figure 5.22(c) shows the *nd*-run of σ on the automaton $FSA_{f_{cm_S}}$. In other words, the instance trace σ can be replayed on the new automaton $FSA_{f_{cm_S}}$ in Figure 5.22(b) and, although the instance is *temporarily violated*, this is a valid ad-hoc change.

Note that, even though the activity *bless* is removed from the model after it was executed (trace σ contains events $(bless, t_s)$ and $(bless, t_c)$) the ad-hoc change in Figure 5.21 is successful. This is due to the property that a set of satisfying traces of a model can contain activities that are *not* in the model (cf.

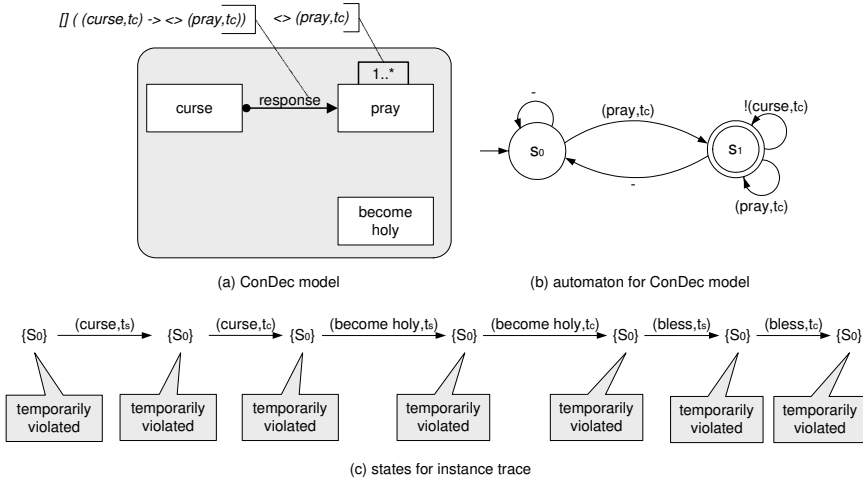


Figure 5.22: Ad-hoc change of ConDec instance ci in Figure 5.21 is *successful* (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$)

Chapter 4). The only consequence of removing activity *bless* from the model is the fact that it will not be possible to execute this activity in the future (cf. enabled events in Definition 4.4.3 and execution rule in Definition 4.4.4).

Figure 5.23 shows an example of an impossible (unsuccessful) ad-hoc change of the instance $ci = (\sigma, cm)$ in Figure 5.21. Figure 5.23(a) shows the new model cm_F for the instance ci : activity *bless* is removed from the original instance model and constraints $1..^*$ and *precedence* are added to the original instance model (cf. Figure Figure 5.21(a)). Constraint $1..^*$ specifies that activity *pray* has to be executed at least once while *precedence* specifies that activity *become holy* can be executed only after the activity *pray*. Figure 5.22(b) shows the automaton $FSA_{f_{cm_F}}$ generated for the mandatory formula of the new model cm_F . Figure 5.23(c) shows that the *nd*-run of σ on the automaton $FSA_{f_{cm_F}}$ *does not exist*, i.e., this change would *violate* the instance. In other words, the instance trace σ can not be replayed on the new automaton $FSA_{f_{cm_F}}$ in Figure 5.22(b) because it is not possible to execute event $(\text{become holy}, t_s)$ from the state s_0 . Therefore, the ad-hoc change from Figure 5.23 is not possible.

5.8 Verification of ConDec Models

In Section 4.6 we described two errors that can occur in constraint models: dead events and conflicts. These errors can be detected in ConDec and any other LTL-based models by analyzing automata generated from LTL formulas.

An event is dead in ConDec model if none of the transitions of the automaton generated for the mandatory formula of the model can trigger the event, as shown in Property 5.8.1.

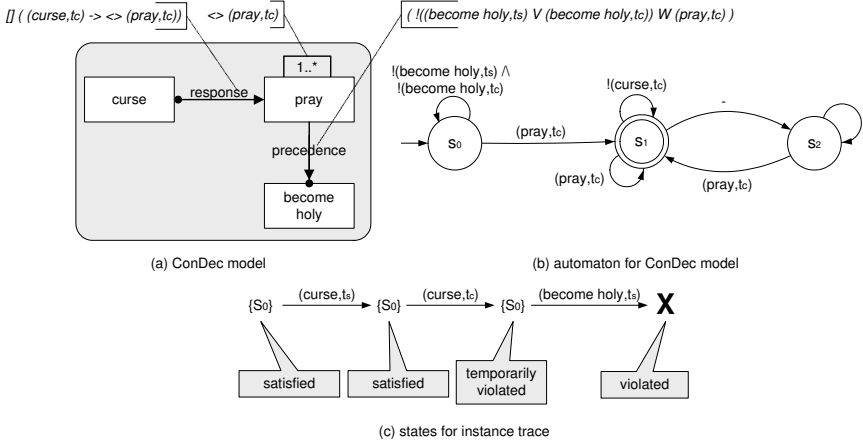


Figure 5.23: Ad-hoc change of ConDec instance ci in Figure 5.21 is *not successful* (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$)

Property 5.8.1. (A dead event cannot be triggered by any transition)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model and $FSA_{f_{cm}} = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be the automaton generated for f_{cm} and $e \in \mathcal{E}$ be an event. It holds that $e \in \Pi_{DE}(cm)$ (i.e., e is a *dead* in model cm , cf. Definition 4.6.1), if and only if $\nexists_{s,s' \in S} (s, e, s') \in T$.

Proof. If it holds that $\nexists_{s,s' \in S} (s, e, s') \in T$, then it holds that $\forall \sigma \in \widetilde{FSA}_{f_{cm}} : e \notin \sigma$ (cf. Definition 5.3.4). Therefore, it holds that $\forall \sigma \in \mathcal{L}(FSA_{f_{cm}}) : e \notin \sigma$ (cf. Definition 5.3.5). Because $\mathcal{E}_{\models cm}^* = \mathcal{L}(FSA_{f_{cm}})$ (cf. Property 5.4.2), it further holds that $\forall \sigma \in \mathcal{E}_{\models cm}^* : e \notin \sigma$, i.e., $e \in \Pi_{DE}(cm)$ (cf. Definition 4.6.1). \square

Figure 5.24(a) shows the ConDec model for the model in Example 4.6.3 on page 111. Due to the fact that event $(\text{become holy}, t_c)$ has to occur at least once (i.e., constraint $1..*$ on become holy) and events $(\text{become holy}, t_c)$ and (curse, t_c) cannot occur both (i.e., constraint *not co-existence*), event (curse, t_c) is dead in this model. This dead event can be easily detected by analyzing the automaton in Figure 5.24(b), which is generated for the mandatory formula of the model: none of the transitions in the automaton can trigger event (curse, t_c) .

A ConDec model has a conflict if the automaton generated for the mandatory formula of the model is empty. Property 5.8.2 shows that the automaton generated for the mandatory formula of the model with a conflict has no states.

Property 5.8.2. (The automaton is empty for a model with a conflict)

Let $cm \in \mathcal{U}_{cm}$ be a constraint model and $FSA_{f_{cm}} = \langle \mathcal{E}, S, T, S_0, S_F \rangle$ be the automaton generated for f_{cm} . It holds that $\mathcal{E}_{\models cm}^* = \emptyset$ (i.e., cm has a conflict), if and only if $S = \emptyset$.

Proof. If it holds that $S = \emptyset$, then it holds that $\mathcal{L}(FSA_{f_{cm}}) = \emptyset$ (cf. Definition 5.3.5 and the algorithm to generate the automata [111, 112, 158]). Because

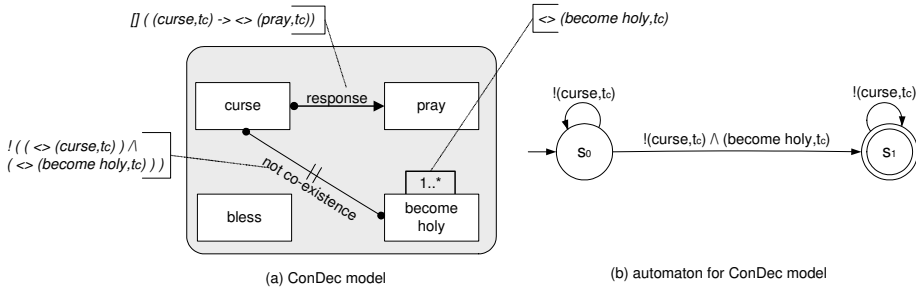


Figure 5.24: A ConDec model where event $(curse, t_c)$ is dead (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

$\mathcal{E}_{\models_{cm}}^* = \mathcal{L}(FSA_{f_{cm}})$ (cf. Property 5.4.2), it further holds that $\mathcal{E}_{\models_{cm}}^* = \emptyset$ (cf. Definition 4.6.1). \square

Figure 5.25(a) shows the ConDec model for the model in Example 4.6.7 on page 4.6.7. Due to the fact that each of the events $(become\ holy, t_c)$ and $(curse, t_c)$ have to occur at least once (i.e., constraints $1..*$ on $become\ holy$ and $curse$) and events $(become\ holy, t_c)$ and $(curse, t_c)$ cannot occur both (i.e., constraint *not co-existence*) this model has a conflict. In other words, there is not trace that can satisfy this model (i.e., even the empty trace does not satisfy the model). The conflict in this model can be easily detected by analyzing the automaton generated for the mandatory formula of the model: the automaton has no states (cf. Figure 5.25(b)).

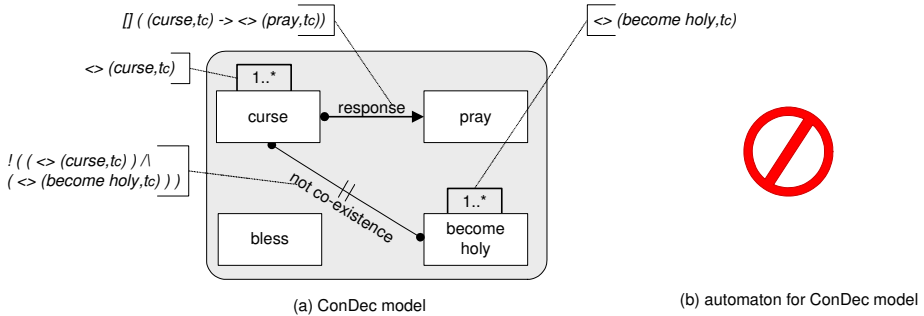


Figure 5.25: A ConDec model with a conflict (recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = started$ and $t_c = completed$)

The cause of an error (dead event or conflict) in a ConDec model $cm = (A, C_M, C_O)$ can be found by searching through the powerset (all subsets) of the mandatory constraints. For each subset $C \subseteq C_M$ an automaton FSA_f is generated for the conjunction of all constraints in the subset, i.e., $f = \bigwedge_{(E,f) \in C} f$ and this automaton is analyzed for errors in the same way as for the whole model (cf. Properties 5.8.1 and 5.8.2). The smallest subset of mandatory constraints for which the error is detected is the cause of the error. Clearly, all verification

and ad-hoc change concepts presented in Chapter 4 can be realized for ConDec and any other LTL-based language.

Sometimes it is necessary to check if models are compatible with each other. For example, if two or more models share some activities but have different constraints, it might happen that the composition of these models contains inconsistencies, i.e., the models are incompatible. We showed in Section 4.6.3 that constraints models can be incompatible with respect to a dead activity and with respect to a conflict. Compatibility analysis is performed by verifying the original and merged (combined) models against dead activities and conflicts. Compatibility of ConDec or any other LTL-based model can be analyzed using the merging procedure described in Definition 4.6.10 and applying the verification techniques for ConDec models presented in Section 5.8.

5.9 Activity Life Cycle and ConDec

ConDec templates do not consider any particular model of a life cycle of activities. Although the three event types (i.e., *started*, *completed* and *cancelled*) are used in the templates, the actual life cycle model (cf. Figure 4.1 on page 84) where activities are first *started* and afterwards *completed* or *cancelled* is not considered in templates. In other words, LTL formulas for ConDec templates allow for an arbitrary order of event types. Consequently, mandatory formulas (cf. Definition 5.4.1) do not consider the activity life cycle. This means that the language of the automaton $FSA_{f_{cm}}$ generated for the mandatory formula of model cm contains traces where activities can be, e.g., *completed* before they are *started* or even without ever being *started*. For example, trace $\sigma = \langle (curse, started), (curse, completed), (pray, completed) \rangle$ is accepted by the automaton in Figure 5.16 on page 143. This means that, even though trace σ is not possible because activity *pray* is *completed* without being *started* before, this trace will be contained in the set of traces that satisfy the ConDec model presented in Figure 5.15 on page 142.

5.9.1 Possible Problems

The absence of an explicit activity life cycle in ConDec can lead to serious errors in execution and verification of ConDec models. In particular, errors might occur when determining *enabled events*, *constraint state* and *instance state* during execution and when discovering *dead activities* and *conflicts* during verification. Moreover, the same holds for any LTL-based language because LTL itself does not impose an activity life cycle. Figure 5.26 shows an illustrative example of an LTL-based constraint model that can cause errors related to the activity life cycle. This model consists of activities x and y and a constraint *error* specifying that activity A cannot be *started* but it must be *completed* (i.e., formula

($\neg \diamond(x, \text{started})$) \wedge ($\diamond(x, \text{completed})$)). The automaton generated for the mandatory formula of this model is presented in Figure 5.26(b). Although none of the transitions in the automaton can be triggered by event $(x, \text{started})$, the accepting state can be reached only by triggering event $(x, \text{completed})$.

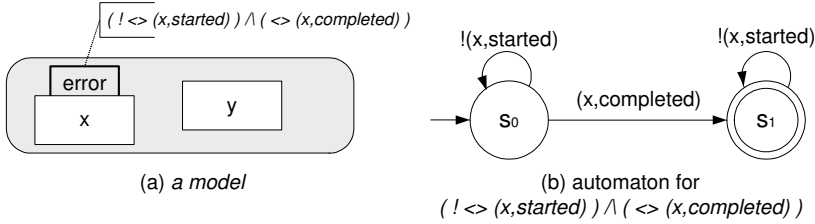


Figure 5.26: A ConDec model with a conflict

Four types of errors might occur in LTL-based languages: (1) while determining which events are enabled in an instance, (2) while determining instance and constraint states, (3) during model verification against dead events, and (4) during model verification against conflicts. Each of these problems is described below using the example shown in Figure 5.26.

Enabled events: Events referring to *completion* or *cancellation* of an activity might be enabled (and available for execution) although the activity was not *started* yet. For example, as long as the automaton in Figure 5.26(b) remains in its initial state s_0 , events $(x, \text{completed})$, $(y, \text{started})$ and $(y, \text{completed})$ will be enabled. In other words, events $(x, \text{completed})$ and $(y, \text{completed})$ will be enabled at the beginning of the instance execution, i.e., before any of the two activities is *started*.

Constraint and instance state: The state of an instance or constraint might be incorrect because an accepting state is reachable *only* by violating the activity life cycle. For example, as long as an instance remains in the initial state s_0 of the automaton in Figure 5.26(b) the instance state is *temporarily violated*, i.e., the accepting state s_1 is reachable via transition $(s_0, (x, \text{completed}), s_1)$ (cf. Property 5.6.2). In other words, this automaton can reach an accepting state only by triggering event $(x, \text{completed})$, while event $(x, \text{started})$ can never be triggered. However, due to the activity life cycle, an occurrence of event $(x, \text{completed})$ must be preceded by an occurrence of event $(x, \text{started})$. Therefore, transition $(s_0, (x, \text{completed}), s_1)$ will never be triggered and the accepting state s_1 will never be reached, i.e., the actual state of the instance is *violated*. Note that the same holds for the state of the *error* constraint.

Dead events: Existing dead event might not be discovered because the event can be triggered *only* by violating the activity life cycle. Only event $(x, \text{started})$ will be discovered as a dead event in the model in Figure 5.26(a)

because this is the only event that cannot be triggered by any of the transitions of the automaton in Figure 5.26(b) (cf. Property 5.8.1). However, due to the activity life cycle and the fact that event $(x, started)$ is dead, event $(x, completed)$ can also never occur in traces that satisfy the model. Thus, although none of the traces that satisfy the model and the activity life cycle contains event $(x, completed)$, verification procedure is not able to discover this event as a dead event (cf. Definition 4.6.1).

Conflicts: Existing conflict might not be discovered because an accepting state is reachable *only* by violating the activity life cycle. According to Property 5.8.2, there is no conflict the model in Figure 5.26(a) because the automaton in Figure 5.26(b) is not empty. However, each trace in the language of the automaton contains event $(x, completed)$ and does not contain event $(x, started)$. Therefore, none of these traces actually complies with the activity life cycle model, i.e., there is no trace that satisfies the model and the model actually has a conflict.

The first problem, i.e., *enabling wrong events*, can easily be solved by a correctly implemented workflow management system. For example, while executing an instance $ci = (\sigma, cm)$ of a model $cm = (A, C_M, C_O)$, the DECLARE prototype (cf. Chapter 6) will enable events $(a, completed)$ and $(a, cancelled)$ if and only if (1) $a \in A$, (2) there exists at least one occurrence of event $(a, started)$ that has not been *completed* or *cancelled* yet, and (3) events $(a, completed)$ and $(a, cancelled)$ are enabled according to Property 5.6.4. Also, event $(a, started)$ will be enabled only if a transition that can be triggered by $(a, completed)$ is reachable from the current set of possible states of the mandatory automaton. Problems referring to instance and constraint state, dead events and conflicts can, to some extent, be ‘softened’ using one of the three approaches described in the next section.

5.9.2 Available Solutions

In order to prevent violation of the activity life cycle in ConDec, for each activity $a \in A$ in a model $cm = (A, C_M, C_O)$ we would need to find a way to specify in the mandatory automaton of the model that each occurrence of events $(a, completed)$ or $(a, cancelled)$ must be preceded by an unique occurrence of event $(a, started)$. In other words, we would need to specify that an activity can be *completed* and *cancelled* exactly as many times as it has been *started* before. For example, it is not possible to *start* activity a once and then *complete* it twice, i.e., it is not possible that *one* occurrence of event $(a, started)$ is followed by *two* occurrences of event $(a, completed)$. Unfortunately, it is not possible to specify in LTL that an event can occur exactly as many times as another event occurred before because it is not possible to ‘count’ how many times an event (in this case event $(a, started)$) occurred [74]. Therefore, the activity life cycle *cannot* be *fully* integrated into

the ConDec language. Instead, three alternative *partial* solutions can be used to minimize possible problems. The first two solutions aim at imposing the *precedence* or *alternate precedence* activity life cycle requirements. These two solutions integrate to some extent the activity life cycle into the model $cm = (A, C_M, C_O)$ itself by (1) creating an additional ‘partial life cycle’ LTL formula for each activity $a \in A$ in the model, (2) adding these formulas to the mandatory formula (as a conjunction) and (3) then generating the automaton for the model. The same procedure must be applied when creating automata used to determine state of each of the constraints. In the third solution we make sure that LTL formulas for *templates are specified carefully*, so that errors are avoided as much as possible. However, these three solutions are only partial, i.e., they do minimize errors but do not guarantee that errors will be completely avoided. Moreover, these solutions come at a cost: they use larger LTL formulas and, therefore, the automata generation becomes less efficient [74, 111, 112, 158].

The *precedence* activity life cycle requirement. For each activity we can specify a *precedence* life cycle requirement: the activity cannot be *completed* or *cancelled* before it is *started*. For each activity $a \in A$ in a model $cm = (A, C_M, C_O)$ an LTL formula similar to the *precedence* template can be generated: $(!(a, t_c) \vee (a, t_x))W(a, t_s)$. These formulas are added as a conjunction to the mandatory formula and formulas used to monitor states of constraints. Figure 5.27 shows an example of the LTL formula for the *precedence* life cycle requirement generated for some activity $a \in A$ and the automaton generated for this formula. It is clear that this solution only partially imposes the activity life cycle, i.e., an activity must be *started* at least once before it can be *completed* or *cancelled*, but once it was *started* it can be *completed* and *cancelled* an arbitrary number of times. For example, it is possible to start the activity once (i.e., one occurrence of event (a, t_s)) and then complete it twice (i.e., two occurrences of event (a, t_c)).

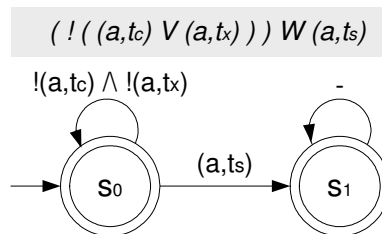


Figure 5.27: The *precedence* activity life cycle requirement (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \textit{started}$, $t_c = \textit{completed}$ and $t_x = \textit{cancelled}$)

The *alternate precedence* activity life cycle requirement. For each activity we can specify an *alternate precedence* life cycle requirement: the activity

cannot be *completed* or *cancelled* before it is *started* and after the activity is *completed* it cannot be *completed* or *cancelled* again until it is *started* again. This requirement can be represented with LTL formula similar to the *alternate precedence* template: $((!(a, t_c) \vee (a, t_x)))W(a, t_s) \wedge (\Box((a, t_c) \Rightarrow \bigcirc((!(a, t_c) \vee (a, t_x)))W(a, t_s)))$. Such formulas are generated for all activities in the model and added as a conjunction to the mandatory formula and formulas used to monitor states of constraints. Figure 5.28 shows an example of the LTL formula for the *alternate precedence* life cycle requirement generated for some activity $a \in \mathcal{A}$ and the automaton generated for this formula. This solution overcomes the shortcoming of the previous one (cf. Figure 5.27), i.e., it is not possible to *complete* an activity more times that it was *started*. However, this solution introduces another shortcoming: now it is not possible to *concurrently* execute one activity multiple times for the same instance. For example, it is not possible to first *start* an activity twice and then complete it twice (e.g., represented with trace $\langle (a, t_s), (a, t_s), (a, t_c), (a, t_c) \rangle$).

$$((!(a,t_c) \vee (a,t_x))) W(a,t_s) \wedge \Box((a,t_c) \rightarrow \bigcirc((!(a,t_c) \vee (a,t_x))) W(a,t_s))$$

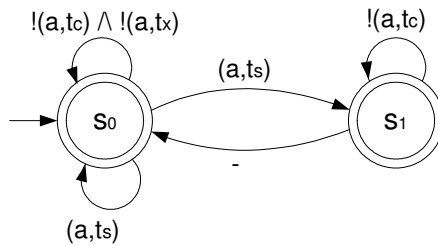


Figure 5.28: The *alternate precedence* activity life cycle requirement (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$)

Carefully defining templates. As the third alternative solution we propose carefully defining templates. For example, the *precedence* template can be defined with formula $!(b, t_s)W(a, t_c)$ to specify that activity b cannot be *started* until activity a is *completed*. This formula is simple, but can cause problems because its automaton is generated such that *completing* and *cancelling* activity b is possible while *starting* the same activity is prohibited, as Figure 5.29(a) shows. To avoid this problem, we can use a safer (but larger) formula for the precedence template to prevent *starting*, *completing* and *cancelling* activity b before *completing* activity a , i.e., we can use formula $(((b, t_s) \vee (b, t_c) \vee (b, t_x)))W(a, t_c)$ (cf. Figure 5.29(b)). On the other hand, the *existence* template has formula $\diamond(a, t_c)$ and it specifies that activity a has to be *completed* at least once. This allows for situations where a is *completed* before being *started* (i.e., event (a, t_s) does not precede event (a, t_c)), which also violates the activity life cycle. Clearly, formula $\diamond((a, t_s) \wedge \diamond(a, t_c))$ would be more appropriate for the *existence* template. In

general, it is advisable that all templates are defined such that (1) if *starting* an activity is prohibited, then *completing* and *cancelling* the activity is also prohibited, and (2) if *completing* or *cancelling* an activity is expected, then *starting* the activity is expected before.

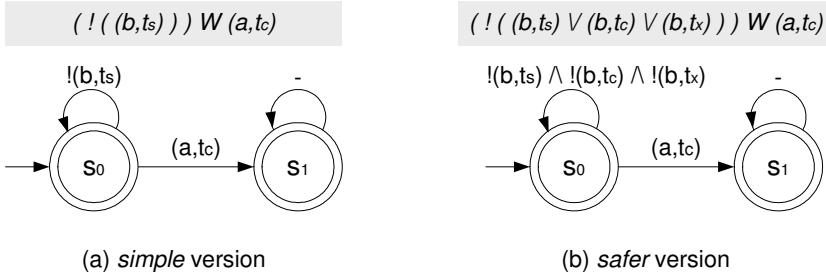


Figure 5.29: Two formulas for the *precedence* template (recall that $t_s, t_c, t_x \in \mathcal{T}$ are event types such that $t_s = \text{started}$, $t_c = \text{completed}$ and $t_x = \text{cancelled}$)

If the precedence or alternate precedence requirement is applied to a constraint model, then the requirement is also enforced in all constraints in the model. However, relying on carefully defined templates does not enforce the (alternate) precedence requirement. Consider, for example, the *precedence template* presented in Figure 5.29 and the *precedence requirement* presented in Figure 5.27³. The ‘safer’ formula indeed prevents starting, completing and cancelling activity a before completion of activity a . However, it still allows for completing activities a and b before starting them, i.e., the *precedence* requirement for activities a and b is omitted. For example, trace $\langle (a, t_c), (b, t_c) \rangle$ is accepted by the automaton in Figure 5.29(b) and, thus, satisfies this formula. Carefully defining templates indeed requires attention when defining every template. For example, using the ‘safer’ version of the precedence template only makes sense if, in all other templates, whenever event (a, t_s) is prohibited, event (a, t_c) is also prohibited. Having this in mind, the safer version of the *not response* template would be $\Box((a, t_c) \Rightarrow !(\Diamond((b, t_s) \vee (b, t_c) \vee (b, t_x))))$.

5.10 Summary

In this chapter we showed how LTL can be used to specify constraints in constraint models. We presented an example of an LTL-based language called ConDec. This language consists of a set of constraint templates - constructs that are used to create constraints in ConDec models. Due to the fact that templates are based on LTL formulas and graphical representation, it is easy to change or remove existing and add new templates to the ConDec language. This makes

³The same holds for the comparison of the *precedence template* and the *alternate precedence requirement*.

ConDec an ‘open’ language that can evolve over time. Similar languages can be created using different templates, which are specific to an application area (e.g., DecSerFlow [37,38] for process models of web services domain and CIGDec [176] for medical processes). Graphical representation of templates and constraints hides the underlying LTL formula and makes ConDec models easier to understand.

The set of satisfying traces of ConDec constraints and ConDec models has a finite representation – the automaton generated for the underlying LTL formula(s). These automata enable execution of ConDec instances, i.e., the state of the instance and enabled events are determined based on the run of the instance trace on the automaton.

The generated automata are also used for ad-hoc change of ConDec instances. If the trace can be replayed on the automaton for the new model, the change is successful. Otherwise, the change is rejected. Note that many problems described in literature (e.g., the “dynamic change bug” and other problems for procedural languages [101,201]) can be avoided, thus making “change easy” [184].

Finally, the generated automata are used for verification of ConDec models. If an event cannot be triggered by any of the transitions, then this event is dead. If the automaton is empty, i.e., it does not have any state, then the model has a conflict. These verification techniques can also be used for the compatibility analysis of ConDec models.

Unfortunately, we were not able to find a way to *fully* incorporate the activity life cycle model in ConDec and other LTL-based languages. This can cause serious problems during execution of instances and verification of models. Three available partial solutions can minimize the problems, but impose larger LTL formulas. Larger LTL formulas decrease the efficiency of automata generation [74,111,112,158] and negatively influence performance of a workflow management system supporting the LTL approach.

Although we presented just one LTL-based language (i.e., ConDec) all the techniques presented in this chapter rely on LTL specification of constraints and, therefore, can be applied to any LTL-based language. In fact, the same ideas could be applied to other temporal logics. This illustrates that the framework presented in the previous chapter is truly generic.

The DECLARE prototype presented in Chapter 6 supports LTL-based constraint language (e.g., ConDec). All principles presented in this chapter are implemented in DECLARE.

Note that LTL is just one example of a suitable language for the specification of constraints. For example, other types of logics can also be used. As discussed in Section 5.1, CTL can also be used for the constraint specification. Another alternative is to use operators from Interval Algebra (IA) (i.e., *before*, *meets*, *during*, *overlaps*, *starts*, *finishes*, *after*, etc.) [50] for constraint specification, and translation of IA networks to Point Algebra (PA) networks [60] for verification and execution of instances [162,163]. Note that LTL, CTL and IA consider time

implicitly (i.e., via their temporal operators) [50, 60, 74]. Using languages like, e.g., Extended Timed Temporal Logic [63] and LogLogics [123], would enable usage of explicit time in constraints (e.g., the *response* template can be extended with a deadline: *activity A must be executed after activity B within N time units*). While timed automata can be used for verification and execution of constraints specified in Extended Timed Temporal Logic, a mature efficient software support of the LogLogics-based language would still need to be developed.

Chapter 6

DECLARE: Prototype of a Constraint-Based System

In this chapter we introduce DECLARE - a prototype based on the constraint-based approach presented in Chapter 4. Although DECLARE currently supports LTL-based languages like, for example, the ConDec language (cf. Chapter 5), it is possible to extend the prototype with other suitable constraint-based languages. In fact, DECLARE also supports DecSerFlow [37,38] and CIGDec [176]. DECLARE is an open-source tool implemented in Java [171], distributed under the GNU General Public License [14], and it can be downloaded from <http://declare.sf.net>.

The remainder of this chapter is organized as follows. In Section 6.1 the prototype's architecture is described. How to define constraint-based languages and templates is described in Section 6.2 and the development of models in Section 6.3. Section 6.4 describes instance execution and Section 6.5 ad-hoc change in DECLARE. Verification of DECLARE models is described in Section 6.6. Tool's simple resource and data perspectives are described in sections 6.7 and 6.8, respectively. Using data elements for defining conditions on constraints is described in Section 6.9 and extending the tool to support other languages in Section 6.10. Section 6.11 shows how DECLARE can be combined with other approaches. Finally, Section 6.12 summarizes this chapter.

6.1 System Architecture

The architecture of the DECLARE system is shown in Figure 6.1. The core of the system consists of the following basic components: *Designer*, *Framework* and *Worklist*. DECLARE can be used in combination with two other tools. First, by combining DECLARE and the workflow management system YAWL [11, 23, 32, 210, 212] constraint-based models can be combined with procedural models. Second, the ProM tool [8, 27] can be used for process mining [28] and run-time

mining and recommendations. The *Framework* component creates event logs that contain information about instances (e.g., which activities were executed, by whom, when, etc.). Constraint templates and constraint models can be exported to ProM and used for verification of past executions recorded in these event logs. Also, ProM can provide run-time recommendations for users (e.g., which activity should be executed next) based on past executions. In Chapter 7 we will describe in more detail how DECLARE and ProM can be used for process mining and recommendations.

6.2 Constraint Templates

An arbitrary number of constraint-based languages can be defined in DECLARE. For example, Figure 6.2 shows how the ConDec language (cf. Chapter 5) is defined in the *Designer* tool. A tree with the language templates is shown under the selected language. On the panel on the right side of the screen the selected template is presented graphically.

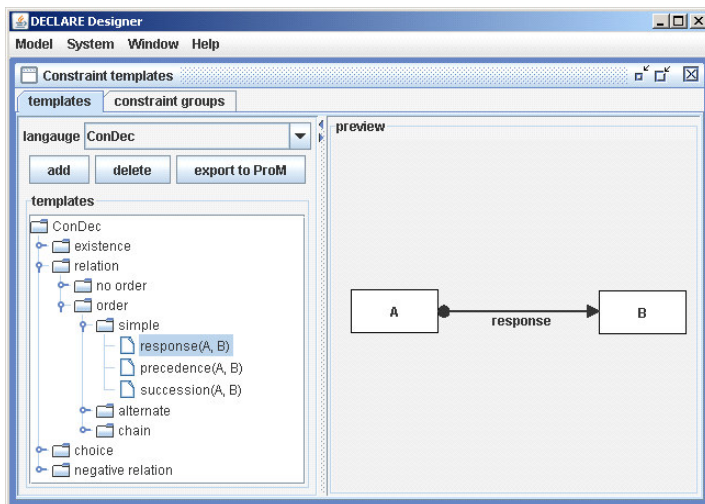


Figure 6.2: Defining a language

An arbitrary number of templates can be created for each language. Figure 6.3 shows a screen-shot of the *Designer* while defining the *response* template. First, the template name and additional display are entered. Next, it is possible to define an arbitrary number of parameters in the template. The *response* template has two parameters: *A* and *B*. For each parameter it is specified if it can be *branched* or not. When creating a constraint from a template in a model, an activity replaces each of the template's parameters. If a parameter is branchable, then it is possible to replace the parameter with more activities. In this case, the parameter will be replaced by a *disjunction* of selected activities

in the formula (cf. Section 5.2.5 on page 133). The graphical representation of the template is defined by selecting the kind of symbol that will be drawn next to each parameter and the style of the line. Figure 6.3 shows that the *response* template is graphically represented by a single line with a filled circle next to the first activity (*A*), and a filled arrow symbol next to the second activity (*B*). Furthermore, a textual description and an LTL formula are given.

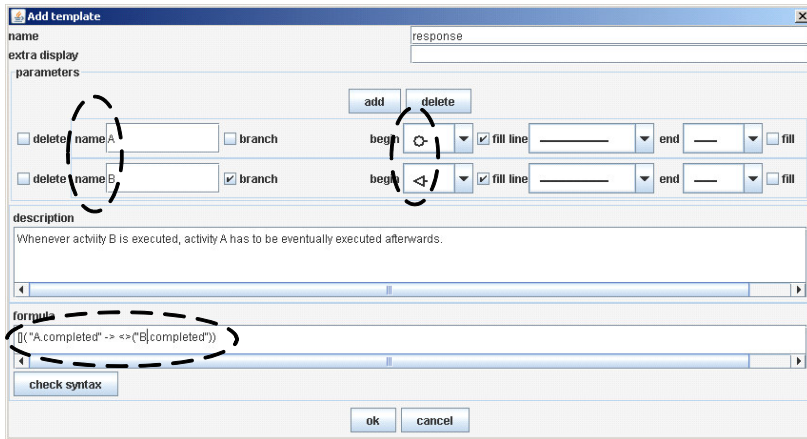


Figure 6.3: Constraint template “response”

DECLARE uses the life cycle of activities presented in Figure 4.1 in Chapter 4. The formula for the *response* template in Figure 6.3 is $\square(\text{“A.completed”} \Rightarrow \diamond(\text{“B.completed”}))$. Events (*A, started*), (*A, completed*) and (*A, cancelled*) are denoted as “A.started”, “A.completed” and “A.cancelled” in DECLARE templates, respectively. A shorter way to denote event (*A, completed*) in DECLARE is by using only “A”.

As we described in Chapter 4, a constraint model consists of mandatory and optional constraints. Optional constraints are not obligatory, i.e., users can violate them. However, allowing users to violate an optional constraint without any warning would totally hide the existence of the constraint. Therefore, when the user is about to violate an optional constraint, DECLARE first issues a warning about the violation to the user. Then, the user can decide to proceed and violate the optional constraint, or to abort and not to violate the constraint. The violation warning contains some information about the constraint. A part of this information is the group to which the optional constraint belongs to. Groups that can be used for optional constraints are defined on the system level in the *Designer*. Each group has a name and a short description, as illustrated by Figure 6.4. For example, it should be easier to decide to violate a constraint belonging to the *Hospital Policy*, then a constraint that belongs to the *Medical Policy*.

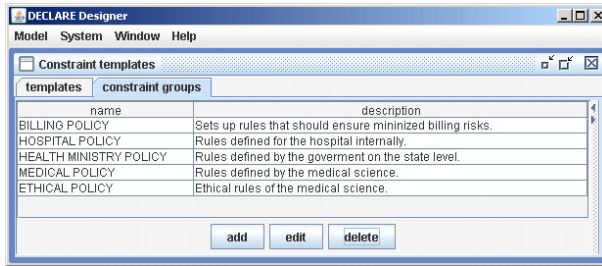
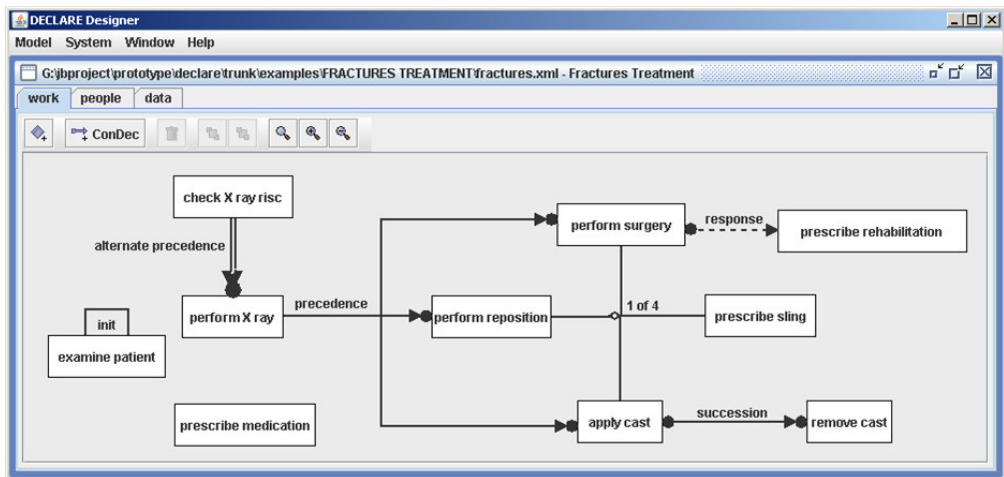


Figure 6.4: Constraint groups

6.3 Constraint Models

Constraint models can be developed in the *Designer* tool for each of the languages defined in the system. For example, we want to use ConDec templates described in Section 5.2 for constraints in the Fractures Treatment model presented in Figure 5.17 on page 145 and Table 5.7 on page 145, and, therefore, we create this model in the *DECLARE Designer* as a ConDec model. Figure 6.5 shows the Fractures Treatment model in *DECLARE*. Activities are presented as labeled rectangles and constraints as special lines between activities.

Figure 6.5: The Fractures Treatment model in *DECLARE*

Each constraint in the model in Figure 6.5 is created using a ConDec template. For example, the constraint between activities *perform surgery* and *prescribe rehabilitation* is created by applying the *response* template, as shown in Figure 6.6. The template is selected in the top left corner of the screen. Underneath the template all its parameters are shown. Activities are assigned to parameters by selecting one or more (in case of branching) activities from the

model. On the right side of the screen some additional information can be given. First, constraint can have an arbitrary name, although the constraint initially gets its name from the template. Second, a constraint can have a condition involving some data element from the model (the handling of data elements and conditional constraints in DECLARE will be explained in sections 6.8 and 6.9, respectively). For example, condition “age < 80” on a constraint would mean that the constraint should hold only if the data element *age* has a value less than 80. Third, for each constraint it must be specified if the constraint is mandatory or optional. If a constraint is optional, then some additional information has to be provided: (1) a group (cf. Section 6.2), (2) the importance level on a scale from 1 (for low importance) to 10 (for high importance), and (3) some local message that should be displayed. This additional information for an optional constraint is presented in warnings when a user is about to violate this constraint.

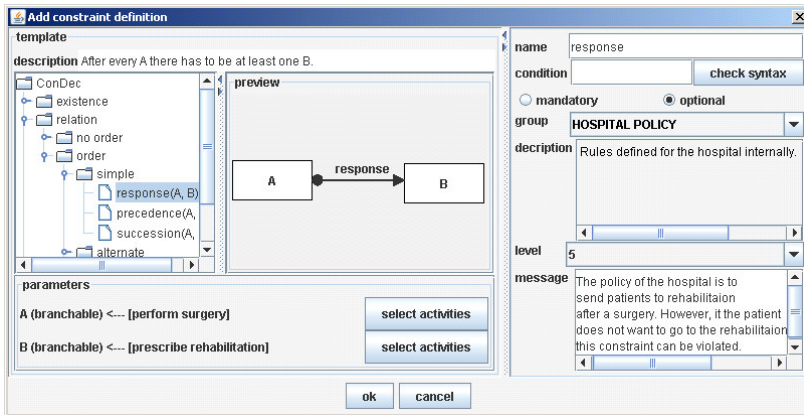
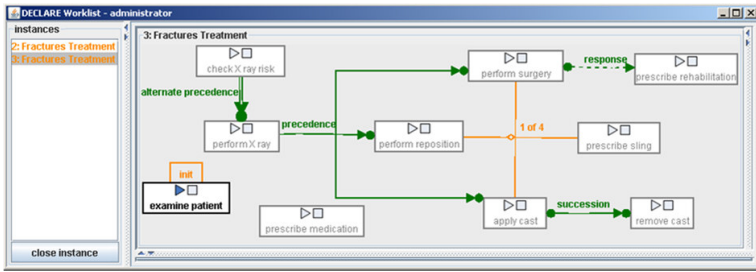


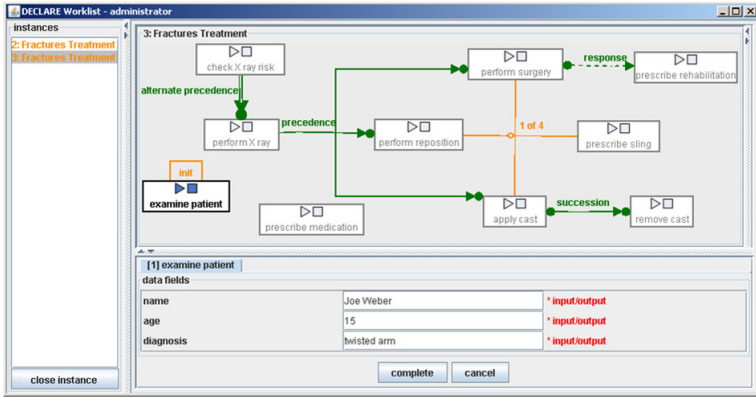
Figure 6.6: Defining a constraint in DECLARE

6.4 Execution of Instances

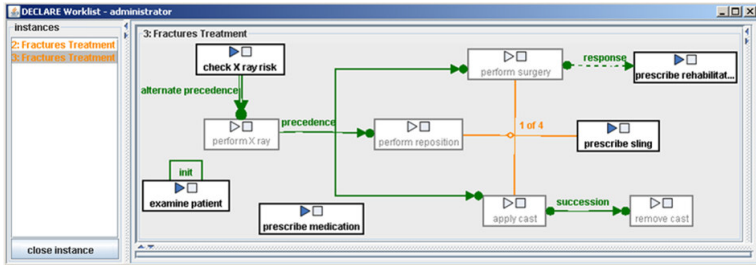
DECLARE determines enabled events, the state of an instance and states of constraints using the approach presented in Section 5.6. Each instance is launched (i.e., created) in the *Framework* tool and users can work on instances via their *Worklists* (cf. Figure 6.1). All instances that a user can work on are presented in the user’s *Worklist*. Figure 6.7 shows several screen-shots of a *Worklist*. All available instances are shown in the list on the left side of the screen. In Figure 6.7, there are two instances of the Fractures Treatment model presented in Figure 6.5 (the list with header ‘instances’ in the upper left corner): ‘2: Fractures Treatment’ and ‘3: Fractures Treatment’. The model of the selected instance is shown on the right side of the screen. After the user starts an activity by double-clicking it, the activity will be opened in the ‘activity panel’ under the model (cf. Figure 6.7(b)).



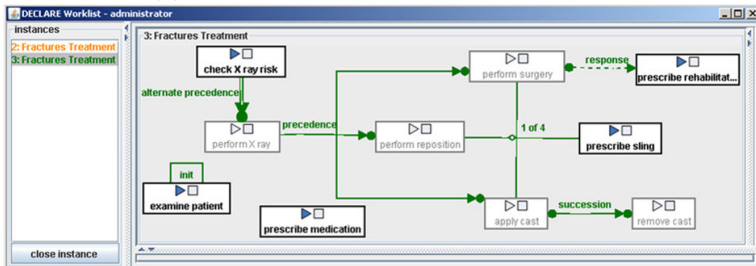
(a) the initial state



(b) after starting *examine patient*



(c) after completing *examine patient*



(d) after starting and completing *prescribe sling*

Figure 6.7: Execution of a Fractures Treatment instance

Although the structure of the process model is the same as in the *Designer* (cf. Figure 6.5), the *Worklist* uses some additional symbols and colors to present to users the current state of the instance (cf. sections 4.4.1 and 5.6.1), enabled events (cf. sections 4.4.2 and 5.6.2), and states of constraints (cf. sections 4.4.3 and 5.6.3). First, each instance in Figure 6.7 has a color, which represents the state of the instance: green for a *satisfied* instance, orange for a *temporarily violated* instance and red for *violated* instance. Second, each activity contains ‘start’ (play) and ‘complete’ (stop) icons, that indicate if users can start/complete the activity at the moment by triggering events *started* or *completed*. The initial state of the process instance in Figure 6.7(a) shows that it is only possible to start activity *examine patient*, because the corresponding symbol is enabled. Starting and completing any of the other activities is not possible, as indicated by the disabled icons. In addition, all currently disabled activities are colored grey¹. This initial state of the process instance is influenced by the *init* constraint on the activity *examine patient*, i.e., this activity is the first activity to be executed and, therefore, the only enabled event is (*examine patient*, *started*). Third, each constraint is colored to indicate its state: (1) *satisfied* – the constraint is represented by a *green* color, (2) *temporarily violated* – the constraint is represented by a *orange* color, and (3) *violated* – the constraint is represented by a *red* color. Figure 6.7(a) shows the initial states of constraints in a Fractures Treatment instance. Constraints *init* and *1 of 4* are *temporarily violated* (i.e., *orange*), while all other constraints in the instance are *satisfied* (i.e., *green*).

Figure 6.7(b) shows the instance after starting activity *examine patient*. This activity is now “open” in the ‘activity panel’ on the bottom of the screen. Data elements that are used in this activity are presented in the ‘activity panel’ (data elements will be explained in Section 6.8). In this case, three data elements are available – patient *name*, *age* and *diagnosis*. In this way, users can manipulate data elements while executing activities. The activity can be *completed* or *cancelled* by clicking on the buttons *complete* or *cancel* on the ‘activity panel’, respectively.

The state of the instance after completing activity *examine patient* is shown in Figure 6.7(c). After the occurrence of the event (*examine patient*, *completed*) the *init* constraint is *satisfied*. This has two consequences: (1) this constraint becomes *green* and (2) it is now possible to start activities *check X ray risk*, *prescribe rehabilitation*, *prescribe medication* and *prescribe sling*. Activity *perform X ray* is still disabled due to the *alternate precedence* constraint, activities *perform reposition*, *perform surgery* and *apply cast* are disabled due to the *precedence* constraint and activity *remove cast* is disabled due to the *succession* constraint.

Figure 6.7(d) shows the state of the instance after starting and completing activity *prescribe sling*. Execution of event (*prescribe sling*, *completed*) results in a state that satisfies the *1 of 4* constraint and, therefore, this constraint becomes

¹Note that we say that an *activity* $a \in \mathcal{A}$ is *enabled* if and only if event (a , *started*) is enabled.

green. Because all mandatory constraints in the instance are now *satisfied*, the instance itself also becomes *satisfied* (i.e., *green*).

In some cases, triggering an event or closing the instance can violate optional constraint(s). For example, consider an instance of the Fractures Treatment model where activity *perform surgery* was executed and the user tries to close the instance without executing activity *prescribe rehabilitation*. Closing the instance at this point would violate the optional constraint *response* (specification of this constraint is shown in Figure 6.6). Instead of automatically closing the instance, DECLARE first issues a warning associated with the optional constraint, as shown in Figure 6.8. The user can now decide based on the information presented in the warning whether to close the instance and violate the constraint or not. Note that, in case of mandatory constraints, this is not possible, i.e., if a mandatory constraint would not be *satisfied*, the whole instance would not be *satisfied*, and it would not be possible to close the instance.

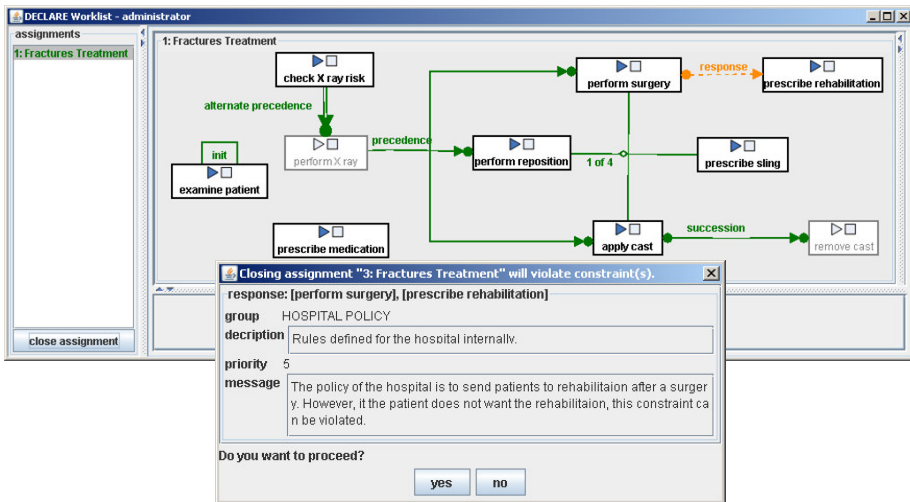


Figure 6.8: Warning: closing the instance violates the optional constraint *response*

6.5 Ad-hoc Change of Instances

Instances in DECLARE can be changed in an ad-hoc manner (cf. Section 4.5) by adding and removing activities and constraints. DECLARE fully supports the approach for ad-hoc change described in Section 5.7. In other words, after the change, DECLARE creates an automaton for the mandatory formula of the new model. If the instance trace can be ‘replayed’ on this automaton, the ad-hoc change is accepted. If not, the error is reported and the instance continues its execution based on the old model. Naturally, when applying an ad-hoc change, it is also possible to verify the new model against dead activities and conflicts

but these errors will not prevent the change. Actually, the procedure for ad-hoc change is very similar to the procedure for starting instances in DECLARE, as Figure 6.9 shows. It is possible to perform the basic model verification in both cases. The only difference is in the execution of the automaton. When an instance is started, the execution of the automaton begins from the initial state. In case of an ad-hoc change, DECLARE first makes an attempt to ‘replay’ the current trace of the instance on the new automaton, i.e., the new model is verified against the current trace. If this is possible, the ad-hoc change is successful and the execution continues from the current set of possible states of the new automaton, i.e., the instance state, enabled events, and states of constraints are determined based on the new automaton and the current trace (cf. Section 5.6). If this is not possible, the ad-hoc change failed and the instance must continue using the old model.

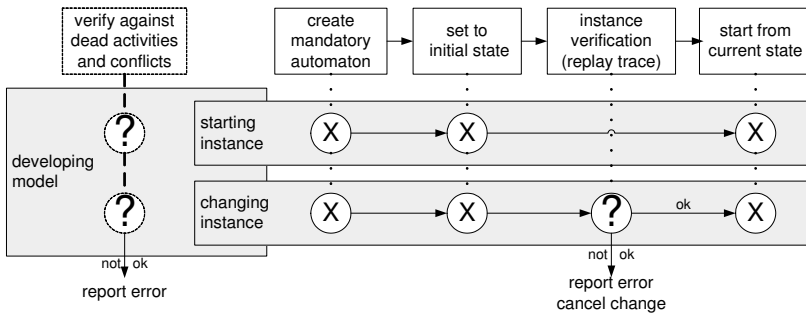
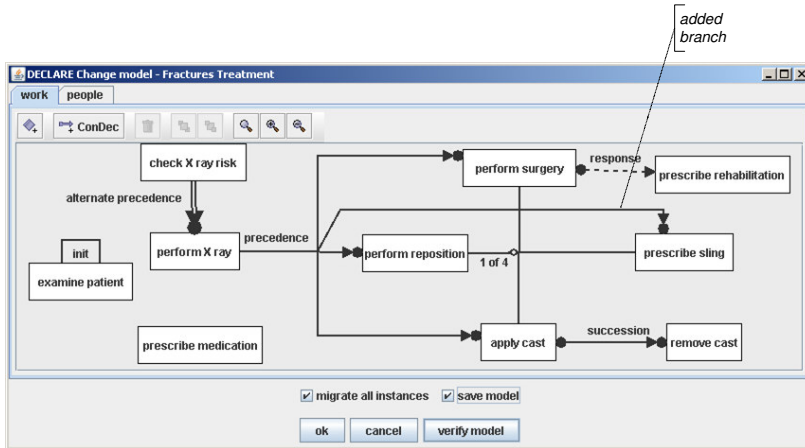


Figure 6.9: Procedure for starting and changing instances in DECLARE

Besides changing an instance, DECLARE offers two additional options: *migration* of all instances and changing the original constraint model (cf. Figure 6.9). First, it is possible to request a migration of all instances, i.e., that the ad-hoc change is applied to all running instance of the same constraint model [202]. DECLARE performs migration by applying the same procedure for ad-hoc change to all instances of the same constraint model, i.e., only instances with traces that can be replayed on the new automaton are changed. Second, it is possible to also change the original constraint model. In this case, all instances created in the future will be based on the new model.

Consider, for example, two Fractures Treatment instances $ci_1 = (\sigma_1, cm^{FT})$ and $ci_2 = (\sigma_2, cm^{FT})$ where $\sigma_1 = \langle (examine\ patient, t_s), (examine\ patient, t_c) \rangle$ and $\sigma_2 = \langle (examine\ patient, t_s), (examine\ patient, t_c), (prescribe\ sling, t_s), (prescribe\ sling, t_c) \rangle$. Figure 6.10(a) shows a DECLARE screen-shot of ad-hoc change of instance ci_1 where activity *prescribe sling* is added as a new branch in the *precedence* constraint. As a consequence of adding this branch, events $(prescribe\ sling, t_s)$, $(prescribe\ sling, t_c)$, and $(prescribe\ sling, t_x)$ can now be executed only after the event $(perform\ X\ ray, t_c)$ (cf. Table 5.2 on page 129). In addition, both the migration and the change of the model are requested. Fig-

ure 6.10(b) shows the DECLARE report for the requested ad-hoc change. The migration is applied to two currently running instances of the Fractures Treatment model, i.e., to instances ci_1 and ci_2 . The change is successfully applied to instance ci_1 and the change failed for instance ci_2 , due to the violation of *precedence* constraint (because event (*prescribe sling*, t_c) already occurs before event (*perform X ray*, t_c) in trace σ_2).



(a) changed instance

instances	result	instance id	instance name
1 error		2	2: Fractures Treatment
ok		1	1: Fractures Treatment

Errors for :2: Fractures Treatment	
result	constraints
name	activities
HISTORY	precedence [perform X ray], [perform reposition, apply cast, perform surgery, prescribe sling]

(b) DECLARE report for ad-hoc change

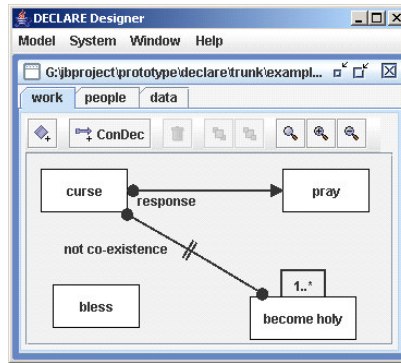
Figure 6.10: Ad-hoc change in DECLARE

6.6 Verification of Constraint Models

DECLARE uses the methods presented in Section 5.8 to detect dead events and conflicts (cf. Section 4.6) and their causes in models. Information about dead events is used to detect the so-called dead activities. An activity in a model is dead if this activity can never be *started* and/or *completed*.

Recall the example of a model with a dead event (*curse*, t_c) from Figure 5.24 on page 154. Figure 6.11(a) shows this model in DECLARE and Figure 6.11(b)

shows the DECLARE verification report for the model: activity *curse* is dead due to constraints $1..*$ and *not co-existence*.



(a) the model

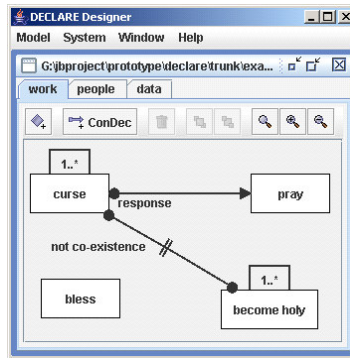
result		constraints		
name		constraint	activities	condition
DEAD ACTIVITY	[curse]	1..*	[become holy]	
		not co-existence	[curse], [become holy]	

(b) activity *curse* is dead

Figure 6.11: A DECLARE model with a dead activity *curse*

The example in Figure 5.25 on page 154 has a conflict. Figure 6.12(a) shows this model in DECLARE. In Section 5.8 we presented one error in this model, i.e., the conflict. However, this model also contains two dead events. Verification in DECLARE detects all errors in a model. figures 6.12(b), 6.12(c) and 6.12(d) show the verification report in DECLARE for the model in Figure 6.12, where three errors are detected. First, Figure 6.12(b) shows that the conflict caused by constraints $1..*$ on *curse*, $1..*$ on *become holy* and *not co-existence*. Second, Figure 6.12(c) shows that activity *become holy* is dead due to constraints $1..*$ on *curse* and *not co-existence*. Third, Figure 6.12(d) shows that activity *curse* is dead due to constraints $1..*$ on *become holy* and *not co-existence*. The conflict in this model is, actually, caused by forcing the execution of the two dead activities, i.e., activities *curse* and *become holy* are dead and there is a $1..*$ constraint on each of these activities.

Detailed verification reports in DECLARE aim at helping model developers to understand error(s) in the model. The goal is to assist the resolution of such problems. As discussed in Section 4.6, errors can be eliminated from a model by removing at least one constraint from the group of constraints that together cause the error. For example, if one of the constraints $1..*$ on *become holy* or *not co-existence* would be removed from the DECLARE model in Figure 6.11(a), activity *curse* would no longer be dead. Also, by removing at least one of the



(a) the model

Verification result - 3 errors were detected.		
result		
name		
CONFLICT		
DEAD ACTIVITY	[become holy]	
DEAD ACTIVITY	[curse]	

constraints		
constraint	activities	condition
1..*	[curse]	
1..*	[become holy]	
not co-existence	[curse], [become holy]	

(b) a conflict

Verification result - 3 errors were detected.		
result		
name		
CONFLICT		
DEAD ACTIVITY	[become holy]	
DEAD ACTIVITY	[curse]	

constraints		
constraint	activities	condition
1..*	[curse]	
not co-existence	[curse], [become holy]	

(c) activity *become holy* is dead

Verification result - 3 errors were detected.		
result		
name		
CONFLICT		
DEAD ACTIVITY	[become holy]	
DEAD ACTIVITY	[curse]	

constraints		
constraint	activities	condition
1..*	[become holy]	
not co-existence	[curse], [become holy]	

(d) activity *curse* is dead

Figure 6.12: A DECLARE model with a conflict

constraints $1..*$ on *curse*, $1..*$ on *become holy* or *not co-existence* from the model in Figure 6.12(a) would remove the conflict in this model.

6.7 The Resource Perspective

The resource perspective specifies which users can execute which activities in instances, as described in Section 3.1.2. The resource perspective of DECLARE is intentionally not designed to resemble the resource perspective of any of the existing workflow management systems. Instead, it is inspired by self-managed work teams, which are described in Section 2.3. A self-managed work team is

responsible for a meaningful piece of work, i.e., for the team's assignment. This style of work assumes that team members have a high degree of knowledge and responsibility for their assignment. Therefore, the team is an autonomous unit and its members are able to fully make their own (local) decisions about how to execute the assignment.

The resource perspective is defined in four steps in DECLARE. First, on the system level, *users* and their *system roles* can be specified in the *Designer* component. Figure 6.13(a) shows eight system roles and Figure 6.13(b) shows six users. An arbitrary number of system roles can be assigned to each user. In Figure 6.13(b) we can see that system roles *nurse* and *anesthesiologist* are assigned to user *Marry Stone*.

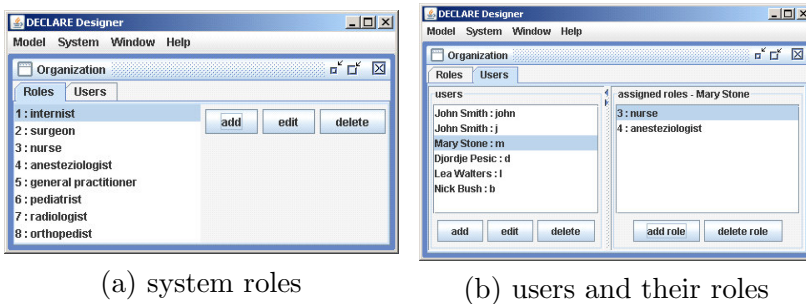


Figure 6.13: Setting up the system level in DECLARE

Second, in addition to system roles, *model roles* can be defined for each model. Figure 6.14 shows five model roles in the Fractures Treatment model. Each of the model roles is associated with one of the system roles. For example, an *orthopedist* on the system level can have a *leading* role in one model, and an *advisory* role in another model.

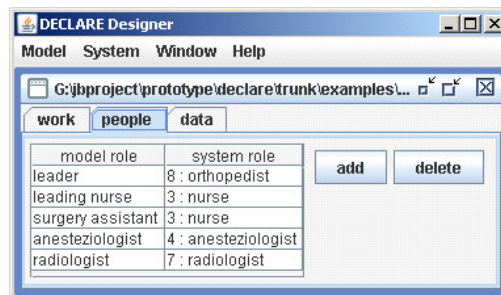


Figure 6.14: Roles in the Fractures Treatment model from Figure 6.5

Third, for each activity in a model, one or more model roles *authorized* to execute this activity can be specified. For example, it might be the case that the activity *examine patient* in the Fractures Treatment model should be executed

only by a *leader*, as shown in Figure 6.15. If several model roles are authorized to execute an activity, then the activity can be executed by any user that has at least one of the authorized roles. If authorized model roles are not defined for an activity, then this activity can be executed by any instance participant.

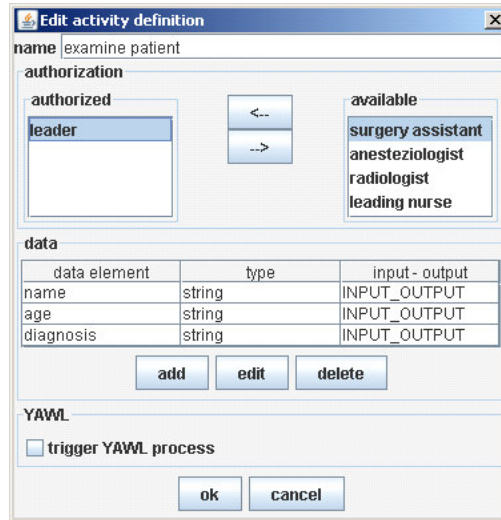


Figure 6.15: Assigning model roles to activities

Finally, when an instance is launched (created) in DECLARE, users that will participate in the instance execution are selected, as shown in Figure 6.16. For each model role an arbitrary number of users that have the system role associated with the model role can be selected. For example, a *surgery assistant* can be any user who has the *nurse* role on the system level (cf. Figure 6.14). Because users *Marry Stone* and *Lia Walters* have the *nurse* system role, they can be assigned as *surgery assistants* in this instance. Users assigned to model roles in an instance are called *instance participants*. An instance can be accessed only by its participants, i.e., an instance is shown only in *Worklists* of its participants. Moreover, an activity in an instance can be executed only by participants that have the role(s) authorized to execute this activity.

Figure 6.17 shows an illustrative example of how resource perspective of the Fractures Treatment model is specified in DECLARE. For the purpose of simplicity, we use only a part of the Fractures Treatment model, i.e., only two model roles (i.e., *leader* and *radiologist*) and two activities (i.e., *examine patient* and *perform X ray*). The first three steps of defining resource perspective are shown in Figure 6.17(a). First, two system roles (i.e., *orthopedist* and *radiologist*) are assigned to four users, i.e., *John Smith* and *Jimmy Travolta* are *orthopedists* and *Jane Travic* and *Nick Bush* are *radiologists*. Second, model roles *leader* and *radiologist* are defined in the model, such that the model role *leader* can be assigned

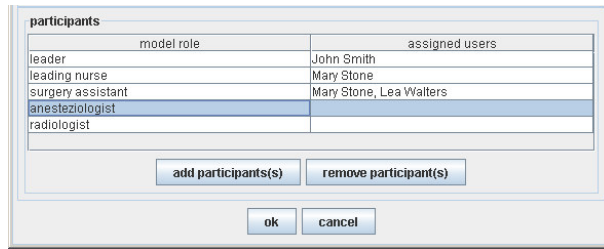


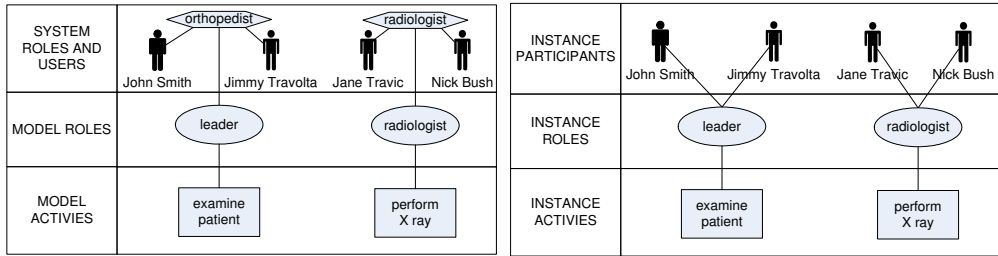
Figure 6.16: Selecting participants for an instance

only to users with the system role *orthopedist* and the model role *radiologist* only to users with the system role *radiologist*. Third, only model *leaders* are authorized to execute activity *examine patient* and only model *radiologists* are authorized to execute activity *perform X ray*. Figures 6.17(b), (c) and (d) show three examples of instances with different users allocated to them. Only users with corresponding system roles can be assigned to model roles in an instance as participants. For example, participants of the instance in Figure 6.17(b) are *orthopedists* *John Smith* and *Jimmy Travolta* as instance *leaders* and *radiologists* *Jane Travic* and *Nick Bush* as instance *radiologists*. An instance is accessible only by its participants. For example, the instance in Figure 6.17(b) will be shown in *Worklists* of all four users, the instance in Figure 6.17(c) will be shown in *Worklists* of *John Smith*, *Jane Travic* and *Nick Bush*, and the instance in Figure 6.17(d) will be shown in *Worklists* of *John Smith* and *Jane Travic*. An activity can be executed only by instance participants with the model role authorized for this activity. For example, only *orthopedists* *John Smith* and *Jimmy Travolta* are allowed to execute activity *examine patient* in the instance in Figure 6.17(b) because they are *leaders* in this instance. This is not the case for the instances in Figures 6.17(c) and (d): only *John Smith* is a leader in these instances and, therefore, only he can execute activity *examine patient* in these two instances.

6.8 The Data Perspective

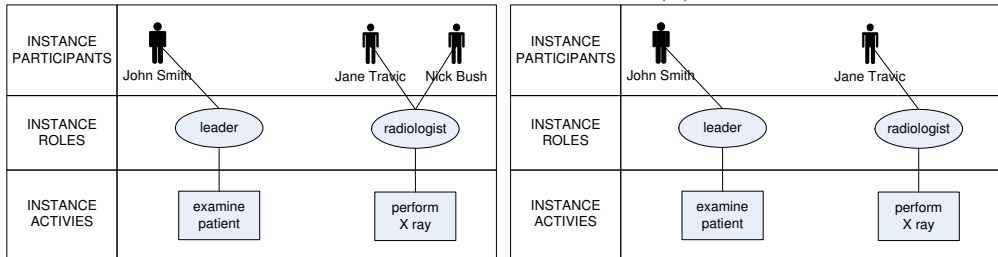
The data perspective defines the way in which information is handled while executing instances of process models. It defines which data elements are available and how these data elements can be accessed, as described in Section 3.1.3. Therefore, defining the data perspective in DECLARE involves (1) defining available data elements on the model level and (2) defining available data elements and their accessibility on the activity level.

Each instance in DECLARE carries its own data. For example, an instance of the Fractures Treatment model should contain the patient name, age and the diagnosis. This kind of instance-related information is stored in data elements of



(a) users, roles and a model

(b) instance ci_1



(c) instance ci_2

(d) instance ci_3

Figure 6.17: The resource perspective in three instances

the instance’s process model and is clearly instance-specific. An arbitrary number of data elements can be defined in each model in DECLARE. A data element has a name, type (e.g., string, integer, etc.) and possibly an initial value. Figure 6.18 shows three data elements in the Fractures Treatment model: patient *name*, *age* and *diagnosis*. These data elements are exclusively owned by instances of the model, i.e., each instance of the Fracture Treatment model will have its own patient *name*, patient *age* and *diagnosis*.

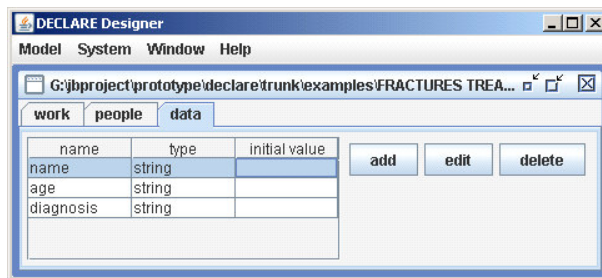


Figure 6.18: Data elements in the Fractures Treatment model from Figure 6.5

As described above, each instance owns its own set of data elements defined in instance’s model. Users can access values of instance’s data elements while executing instance’s activities. Therefore, an arbitrary number of model’s data elements can be available in each activity of the model. Figure 6.19 (the bottom

of the screen) shows how available data elements are assigned to an activity in the *Designer* tool, while developing a process model. First, any of the data elements defined in the model can be assigned to an activity. Figure 6.19 shows that data elements *name*, *age* and *diagnosis* are available in activity *examine patient*. Second, for each of the available data elements the type of accessibility must be defined. Recall that, in Section 3.1, we presented the two types of data accessibility: (1) the value of an input data element can be accessed but not edited in the activity and (2) the value of an output data element can be edited in the activity. DECLARE uses a similar approach: data elements can have *input*, *output* or *input-output* accessibility in activities. The first two types of accessibility correspond to the two types described in Section 3.1, while an *input-output* data element is both *input* and *output* in the activity. Note that the definition of available data elements and their accessibility influences the way users can manipulate data in instances. Users can manipulate the activity data elements while executing an activity in a *Worklist*, as shown in Figure 6.7(b) on page 169.

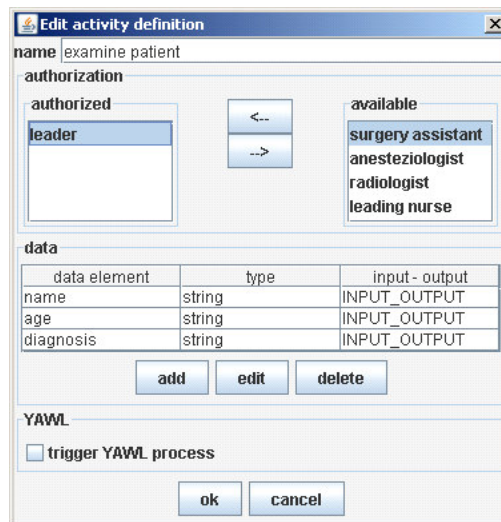


Figure 6.19: Adding an optional constraint

6.9 Conditional Constraints

A *condition* can be defined for any constraint in the special field on the screen for defining a constraint, as shown in Figure 6.20. A condition is an expression involving data elements from a process model. Consider, for example, data elements in the Fractures Treatment process presented in Figure 6.18. These data elements can be used to define conditions on any of the constraints in the

Fractures Treatment model. For example, it might be the case that the hospital changed the policy about *prescribing rehabilitation* after *performing surgery*, and that, from now on, rehabilitation *must* be prescribed after surgery to all patients who are *less than eighty years old*². This would require the *response* constraint between activities *perform surgery* and *prescribe rehabilitation* to be a mandatory constraint with condition $age < 80$, as shown in Figure 6.20. If a constraint has a condition, then this condition is displayed in the model above the constraint. We refer to a constraint with a condition as to a *conditional constraint*.

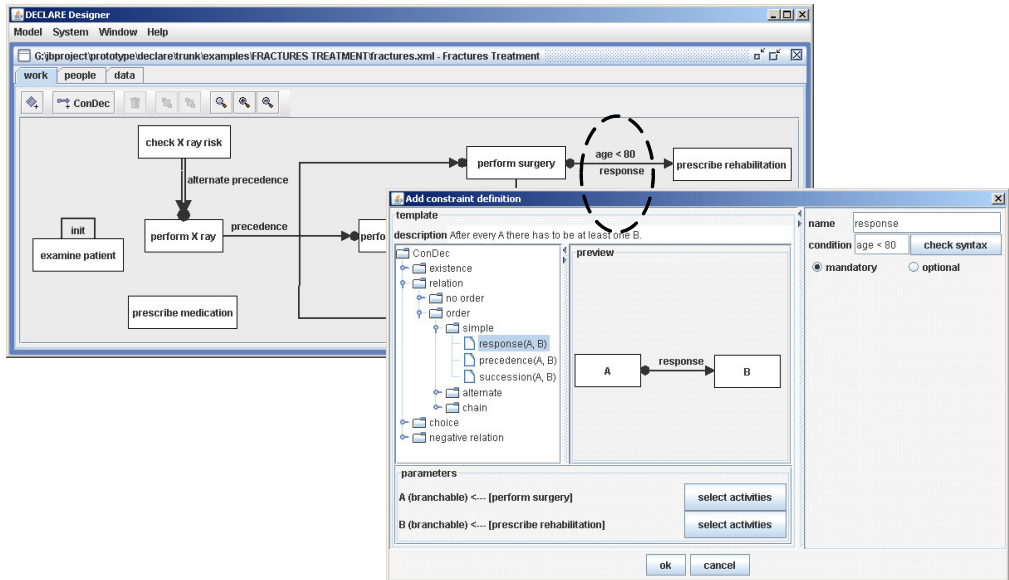


Figure 6.20: Condition on a constraint

At any point during instance execution a condition value is either **true** (i.e., the related constraint is applicable) or **false** (i.e., the related constraint is not applicable). The value of a condition depends on the value of data elements involved in the condition. For example, if data element *age* has value 30 in an instance, then the value of condition $age < 80$ is **true**, and the *response* constraint between activities *perform surgery* and *prescribe rehabilitation* is applicable in the instance (cf. Figure 6.20). If, on the other hand, *age* has value 82 in an instance, then the value of this condition is **false**, and this constraint is not applicable in the instance. If the condition on a constraint is **false**, then this constraint is presented with a light *gray* color in *Worklists* (instead of the color representing constraint's state).

DECLARE handles a conditional constraint in an instance in a special way, depending on the value of the condition, as Table 6.1 shows. If the value of the

²Note that, originally, the *response* constraint between *perform surgery* and *prescribe rehabilitation* was defined as optional (cf. Figure 6.5 on page 167).

condition is **true**, then the related constraint is applicable, i.e., the state of the constraint is monitored and presented with the right color in *Worklists* and, if the constraint is mandatory, it is included in the mandatory formula/automaton (cf. Definition 5.4.1 on page 142). If the value of the condition is **false**, then the related constraint is not applicable, i.e., the state of the constraint is not monitored, the constraint is ‘grayed-out’ in *Worklists* and, even if the constraint is mandatory, it is discarded from the mandatory formula.

Table 6.1: Conditional constraints

condition	constraint	constraint state	mandatory constraint
true	applicable	monitor (present state color)	consider
false	not applicable	do not monitor (<i>gray-out</i>)	discard

As users execute activities in their *Worklists*, instance’s data elements change values (cf. Figure 6.7 on page 169). Thus, it might happen that a condition changes its value several times during the execution of the instance. As Table 6.2 shows, DECLARE handles a change of the value of a condition as a special type of ad-hoc change (cf. Section 6.5). We refer to this kind of ad-hoc instance change as to a *conditional change*. If the value of a condition changes from **false** to **true**, then the related constraint is added to the instance. If the value of a condition changes from **true** to **false**, then the related constraint is removed from the instance.

Table 6.2: Conditional change as an ad-hoc change

condition value		instance
old	new	ad-hoc change
false	true	add constraint
true	false	remove constraint

A conditional change can be handled in a similar manner like an ad-hoc change, as Figure 6.21 shows³. However, conditional change can be handled using two strategies: the *unsafe* or the *safe* strategy. Both strategies start with creating a new mandatory automaton, setting it to its initial state and then ‘replaying’ the trace of the instance on the new automaton.

If the *unsafe* strategy is used, then the execution of the activity that causes the conditional change is accepted, regardless the new state of the instance. In other words, the *unsafe* strategy allows conditional changes that bring the instance in the state *violated*. The violation of the instance can be caused when the value of the condition on a *mandatory* constraint changes from **false** to **true**. This is because the related mandatory constraint is added to the instance,

³See Figure 6.9 for a similar diagram explaining ad-hoc change.

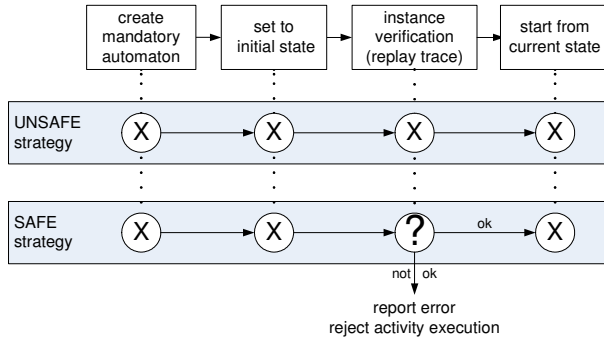


Figure 6.21: Two strategies for conditional change

which might discard the current trace from the set of traces that satisfy the new model of the instance (cf. Property 4.2.5 on page 94).

The *safe* strategy resembles the ad-hoc change more than the *unsafe* strategy, because this strategy does not allow a conditional change that would bring the instance into the *violated* state. If executing an activity would change value(s) of data element(s) in such a way that the instance becomes *violated*, then this error is reported (just like in the ad-hoc change) and the execution of this activity is rejected.

Currently, DECLARE uses the *unsafe* strategy for conditional change. However, the *safe* strategy can also be easily implemented using existing techniques for ad-hoc change presented in Section 6.5.

Conditional mandatory constraints can cause problems during verification of models in DECLARE. This is because (1) conditions are independent from the formal specifications of constraints and (2) the verification procedure in DECLARE does not take conditions on mandatory constraints into account, i.e., all constraints are treated as unconditional during the verification. This can cause DECLARE to detect and report a verification error even if, due to condition(s), the error does not exist. For example, consider a model where a set of mandatory constraints causes an verification error (i.e., a dead event or a conflict), and two constraints c_1 and c_2 from this set with conditions $age < 20$ and $age > 50$, respectively. DECLARE will not take these conditions into account during verification, i.e., it will detect this error and the set of mandatory constraints that causes it. However, because these two conditions can never evaluate to **true** at the same time, at any point of time at least one of these two constraints will be discarded from the instance. Therefore, the detected verification error does not actually exist (cf. Property 4.6.4 on page 111 and Property 4.6.8 on page 114). In order to overcome this problem to some extent, DECLARE presents conditions of all constraints that cause a detected error (cf. figures 6.11 and 6.12 on pages 174 and 175, respectively).

6.10 Defining Other Languages

The ConDec language (cf. Section 5.2) is not the only language that can be used in DECLARE. In this section we show how another languages can be supported by DECLARE. Section 6.10.1 describes how other LTL-based languages can be defined (cf. DecSerFlow [37,38], CIGDec [176], etc.). Besides LTL-based languages, the prototype can be extended to support languages based on other formalizations, as described in Section 6.10.2.

For illustration purposes we use a simple, hypothetical, language called the ‘Simple Language’. This language uses several procedural control-flow patterns [35,213]. As Figure 6.22 shows, the ‘Simple Language’ has five templates, i.e., *sequence*, *parallel split*, *synchronization*, *exclusive choice* and *simple merge*, such that each of them is illustrates a control-flow pattern [35,213].

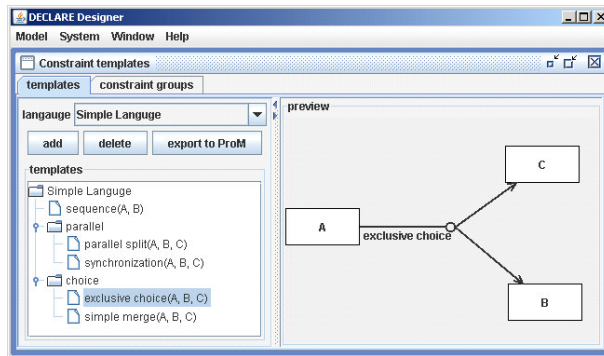


Figure 6.22: Defining the ‘Simple Language’

Templates from the ‘Simple Language’ can be used to specify relations between activities in models. For example, the *Handle Complaint* process (cf. Figure 3.3 on page 52) can be modeled in the ‘Simple Language’, as Figure 6.23 shows. Note that, due to usage of identical control-flow patterns/templates, the *Handle Complaint* models shown Figures 3.3 and 6.23 have identical semantics, i.e., they represent the same process.

Parameters, graphical representation and formal specification must be specified for each of the templates in the ‘Simple Language’. In sections 6.10.1 and 6.10.2 we describe how the templates of the ‘Simple Language’ can be defined using LTL or some other formalism, respectively.

6.10.1 Languages Based on LTL

Because DECLARE is by default able to support LTL-based languages, defining such a language is trivial in the prototype. This is done in two steps. First, the name for the new language must be given. Second, all templates of the new language must be added. Further, it is important to define the appropriate

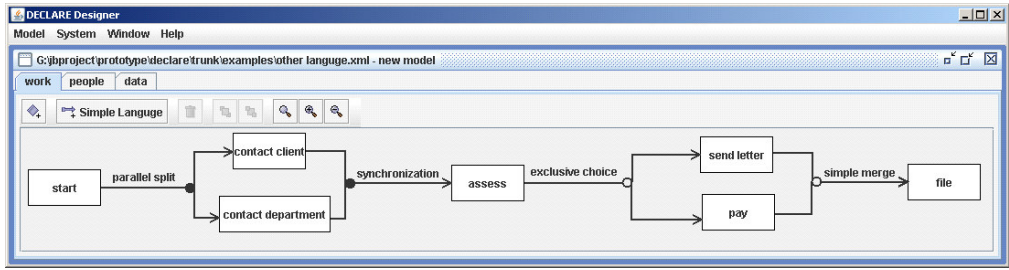


Figure 6.23: A ‘Simple Language’ model

graphical representation and LTL formula for each template. Figure 6.24 shows how the *exclusive choice* template of the ‘Simple Language’ can be defined in LTL. Note that the given LTL formula specifies the semantics of the template. In this case, the *exclusive choice* template specifies that (1) activities *B* and *C* cannot be *started* before *A* is *completed*, (2) *B* or *C* must be *completed* after *A* is *completed* and (3) *B* and *C* cannot both be *completed*. Note that we selected a particular semantics for each of the five patterns. For simplicity, we did not consider the more advanced use of these five patterns (loops, etc.)

Figure 6.24: The *exclusive choice* template in LTL

6.10.2 Languages Based on Other Formalizations

Although it currently uses LTL for constraint specification (cf. Chapter 8), the DECLARE prototype can be extended to support other languages. In this case, defining the language, its templates and models remains the same like in the LTL based languages, e.g., ConDec. The only difference is that, the new formalization must be used when defining formulas of templates. This can be an arbitrary

textual formalization. For example, Figure 6.25 shows how the *exclusive choice* template of the ‘Simple Language’ can be defined using a hypothetical formal specification in the field *formula*.

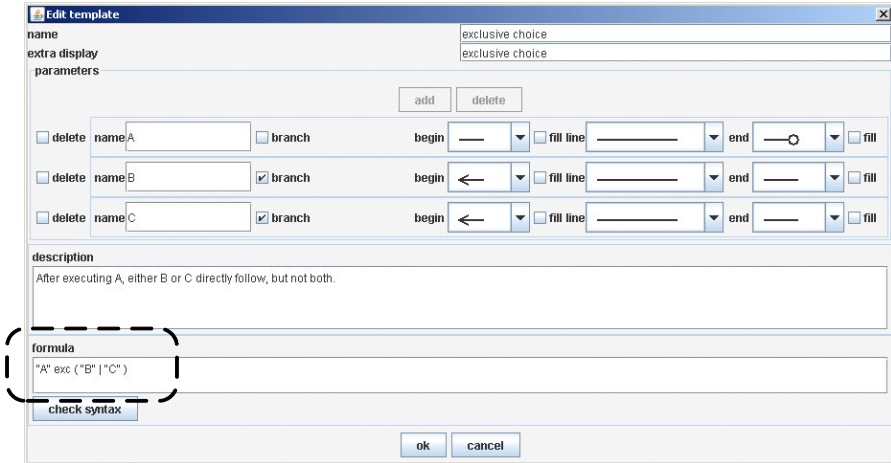


Figure 6.25: The *exclusive choice* template in a hypothetical formalization

As shown in Figures 6.22, 6.25 and 6.23, DECLARE already supports developing non-LTL-based languages and their models. However, in order to be able to verify models and execute instances of a new language, some extensions of the prototype are necessary. Implementing a non-LTL-based language in DECLARE requires extending the prototype with several Java classes [171], as shown in Figure 6.26. Figure 6.26(a) shows classes *LTLInstanceExecutionHandler* and *SLInstanceExecutionHandler*, which are responsible for execution of LTL and the ‘Simple Language’ instances, respectively. These classes must implement methods of the *IInstanceExecutionHandler*, which are responsible for (1) execution of the next event (method *next*), (2) retrieving enabled events (method *enabled*), (3) information about possible violation of optional constraints (method *violatesConstraints*), (4) retrieving the state of constraints (method *constraintState*), (5) retrieving the instance state (method *instanceState*), and (6) performing ad-hoc instance change (method *reset*). Figure 6.26(b) shows classes that are used for the verification of models and traces in ad-hoc change of LTL and the ‘Simple Language’ instances. Classes *LTLVerification* and *SLVerification* extend *ModelVerification* and are responsible for verification of LTL and the ‘Simple Language’ models. Classes *LTLHistoryVerification* and *SLHistoryVerification*, which extend *HistoryVerification*, verify the new model against current trace during ad-hoc change of LTL and the ‘Simple Language’ instances.

As Figure 6.26 shows, in order to implement a new language in DECLARE, it is necessary to create three classes: (1) a class that implements the *IInstanceExecutionHandler* is responsible for execution of instances, (2) a class that extends

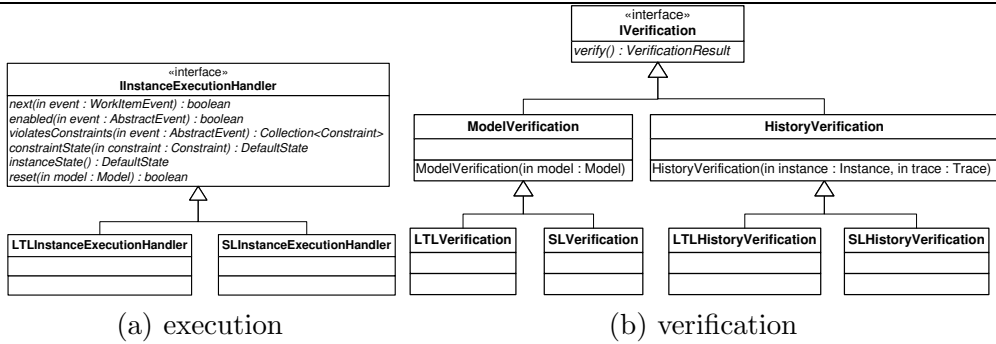


Figure 6.26: Implementation of a new language in DECLARE

ModelVerification for verification of models and (3) a class that extends *HistoryVerification* that is responsible for the verification of ad-hoc instance change. For example, classes *SLInstanceExecutionHandler*, *SLVerification*, and *SLHistoryVerification* handle models and instances of the ‘Simple Language’. In addition to the classes shown in Figure 6.26, it is advisable to also implement classes that check syntax of the formal specification of templates in the new language, i.e., the content of the *formula* field shown in Figure 6.25.

6.11 Combining the Constraint-Based and Procedural Approach

The nature of a business process determines the appropriate approach with respect to workflow technology. On the one hand, flexible approaches (e.g., constraint-based approach), where users control the work, are appropriate for turbulent business processes. For example, the right treatment for each patient in the Fractures Treatment process (cf. Figure 6.5) depends on the specific injury. Therefore, this process must be flexible enough to allow the medical staff to use their expertise and experience and provide the best treatment for each patient. On the other hand, some business processes might require that a strict procedure is followed for each instance. For example, a blood analysis in a medical laboratory must always be conducted following a prescribed procedure, in order to deliver reliable results. The procedural approach where the system controls the work is more appropriate for business processes that must strictly follow some prescribed procedure.

As discussed in Chapter 1, none of the two approaches is sufficient in its own. On the contrary, organizations often need to combine flexible and procedural approaches because procedural processes (e.g., laboratory analysis) and flexible processes (e.g., performing surgeries) are often integrated in organizations. In

other words, a business process is often composed of both flexible and procedural subprocesses. Therefore, it is remarkable that commercial workflow management systems tend to support *either* one *or* the other approach. Moreover, there are many ICT approaches that use sophisticated methods for developing interfaces between different systems (e.g., Service Oriented Architectures [52]). Similar concepts can be applied to workflow technology in order to allow for arbitrary decompositions of process models developed in various languages and enacted by various systems. Figure 6.27 shows an illustrative example of how various processes can be combined in arbitrary decompositions. Instead of being executed as simple (manual or automatic) unit of work, an activity can be decomposed into a process modeled in an arbitrary language.

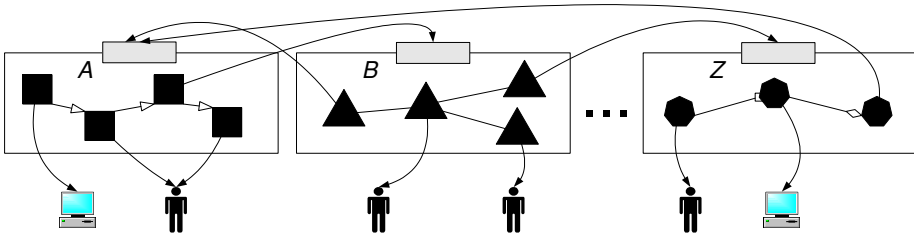


Figure 6.27: Decomposing processes using various modeling languages: A, B, \dots, Z

6.11.1 Decomposition of declare and YAWL Processes

YAWL is a workflow management system that supports the procedural approach developed in a collaboration between the Eindhoven University of Technology and the University of Queensland [11, 23, 32, 210, 212]. YAWL aims at supporting most of the workflow patterns [10, 35, 208] (cf. Section 3.1). The YAWL system consists of three typical components of a workflow management system (cf. Figure 3.1 on page 48), as Figure 6.28 shows. First, the YAWL Editor can be used to create process models. Second, the YAWL engine manages execution of instances. Finally, each user uses its YAWL Worklist to execute activities in running instances.

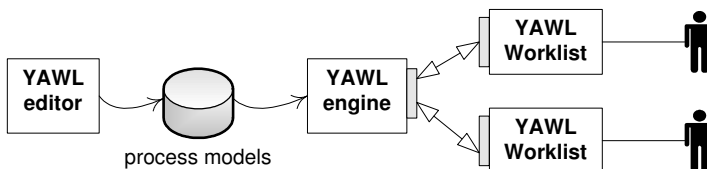


Figure 6.28: Workflow management system YAWL

The architecture of YAWL enables easy interaction between YAWL and other applications, i.e., the so-called custom YAWL services. Figure 6.29(a) shows the

interface between YAWL and a custom service. First, YAWL can delegate an activity to the service, instead of the YAWL Worklist (cf. activity marked with letter *S* in Figure 6.29(a)). Second, a custom service can request launching of a new instance in YAWL. In both cases relevant instance and activity data elements are exchanged between YAWL and the service. The YAWL Worklist is a custom YAWL service. By default, all activities in YAWL models are delegated to the YAWL Worklist, i.e., all activities are by default executed by users in their YAWL worklists. If an activity should be delegated to another custom service, then this must be explicitly specified in the YAWL model.

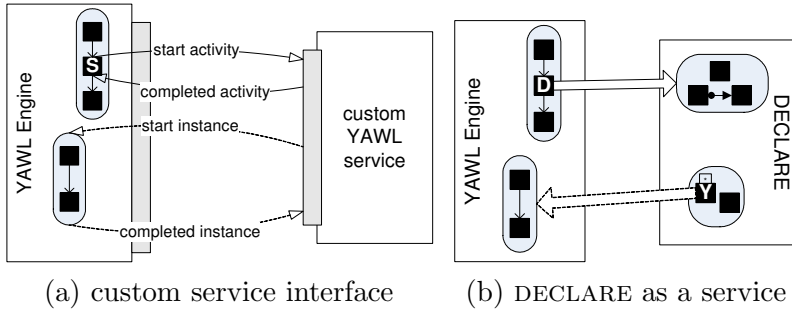


Figure 6.29: DECLARE as a custom YAWL service

DECLARE *Framework* can act as a custom YAWL service. This enables arbitrary decompositions of DECLARE and YAWL models, as shown in Figure 6.29(b). First, it is possible that a YAWL activity triggers execution of a DECLARE instance: when the YAWL activity becomes enabled DECLARE will launch its instance. YAWL will consider the completion of the launched DECLARE instance as a completion of its activity. Second, a DECLARE activity can trigger execution of a YAWL instance. Note that users execute ‘standard’ activities of DECLARE and YAWL instances in a default manner in DECLARE and YAWL worklists, as Figure 6.30 shows.

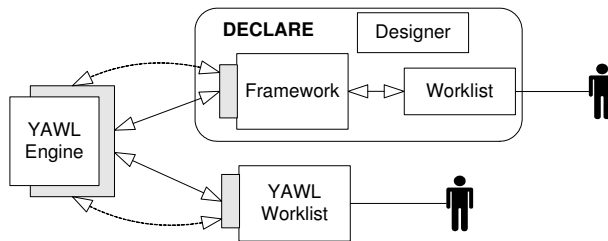


Figure 6.30: Interface between DECLARE and YAWL

Arbitrary decompositions of DECLARE and YAWL models allow for integrating the constraint-based and the procedural approach on different abstraction levels within one business process. This way the designer is not forced to make

a binary choice between flexible and inflexible processes. Instead, an integration can be achieved, where parts of the processes that need a high degree of flexibility are supported by constraint-based DECLARE models and parts of the processes that need centralized control of the system are supported by YAWL models. Consider, for example, the decomposition of a part of a health care process shown in Figure 6.31. On the highest decomposition level, the main process is modeled using a procedural YAWL model. Each visit starts with *opening the file* and a quick *preliminary examination*. If the *preliminary examination* shows existence of an injury, the patient is accepted for the *fractures treatment*. Finally, each visit must be *archived*. On the second level, the DECLARE Fractures Treatment model (cf. Section 6.3) offers a high degree of flexibility to medical experts that treat the injury. Finally, activity *perform surgery* is decomposed into another YAWL model. Note that, further decomposition can also be achieved. For example, activities *prepare* and *schedule* could consist of several smaller steps and, thus, also need to be decomposed into YAWL or DECLARE subprocesses.

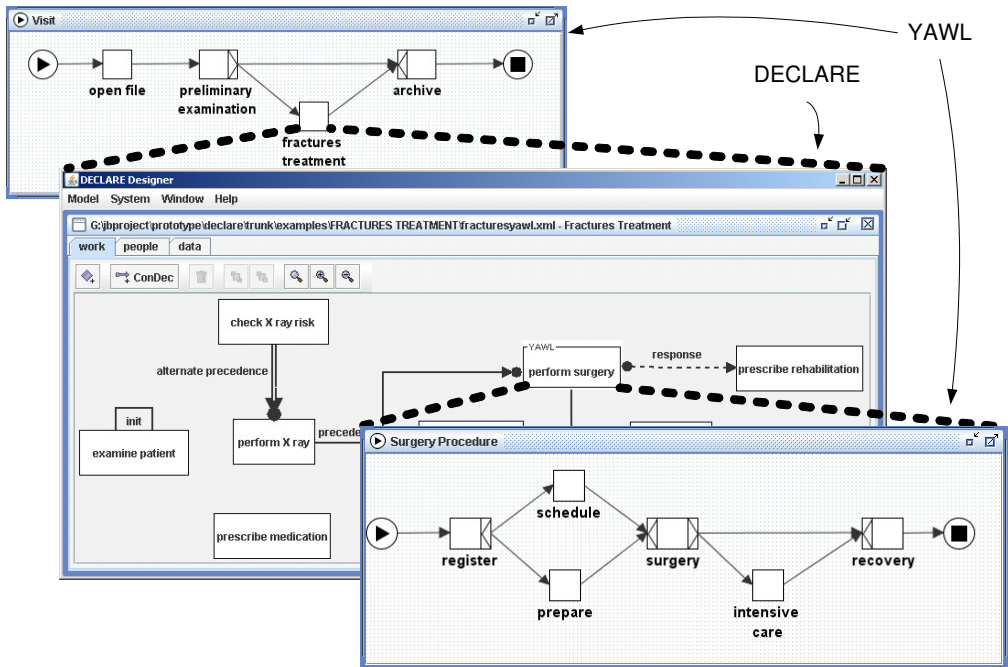


Figure 6.31: An example: decomposition into DECLARE and YAWL process models in the health care domain

In both YAWL and DECLARE models activities are by default offered to users to manually execute them. If an activity should be delegated to an external application, then this must be explicitly defined in the process model. Figure 6.32 shows the definition of the *perform surgery* activity from the Fractures Treatment model shown Figure 6.31: this activity is decomposed into the *Surgery Procedure*

YAWL model. Therefore, activity *perform surgery* is graphically presented with a special “YAWL” symbol in the Fractures Treatment model in Figure 6.31. Note that, although activity *perform surgery* will not be executed manually by a user, model role *leader* is authorized for it. This means that instance participants with the role *leader* are authorized to decide *when* this activity (i.e., the referring YAWL model) can be executed. However, when an participant starts the activity, it will not be opened in the ‘activity panel’ in the *Worklist*. Instead, it will be automatically delegated to YAWL, which will launch a new instance of its Surgery Procedure model.

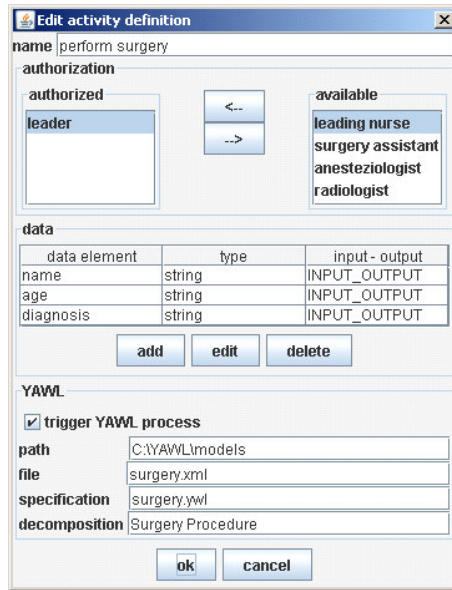


Figure 6.32: DECLARE activity *perform surgery* launches a YAWL instance

Similarly, in a YAWL process model it can be specified that activity should be delegated to a custom YAWL service, e.g., DECLARE. Figure 6.33 shows a YAWL instance where activities D_1 , D_2 and D_3 are delegated to DECLARE. In the general scenario, DECLARE users must manually select which DECLARE model should be executed for each YAWL request. For example, DECLARE users can select to execute *Model B* for activity D_1 . If the decomposed YAWL activity contains an input data element with name ‘delaremodel’ (cf. sections 3.1 and 6.8), then DECLARE automatically launches a new instance of the referring model. For example, activity D_2 launches a new instance of *Model A* in DECLARE. If the specified model cannot be found, DECLARE users must manually select a DECLARE process models to be executed. For example, users can select to execute an instance of model *Model C* for activity D_3 . Because the ‘delaremodel’ data element is also an output data element in activity D_3 , DECLARE will return the name of the executed model to YAWL. In this manner YAWL users are informed

about the subprocess that was executed for the decomposed YAWL activity.

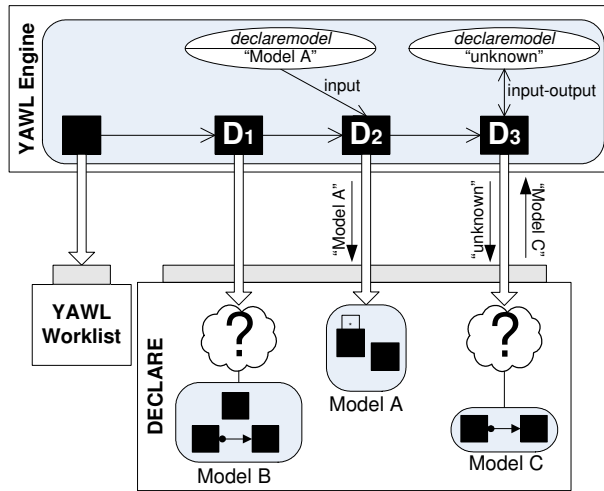


Figure 6.33: YAWL activities D_1 , D_2 and D_3 launch DECLARE instances

6.11.2 Dynamic Decompositions

Decompositions of DECLARE and YAWL models are dynamic, i.e., the decomposition structure can be changed at run-time. To some extent, this enables building process models/instances ‘on the fly’.

As described in Section 6.5, DECLARE models can be changed during execution. During the so-called ad-hoc change, constraints and activities can be added or removed from an instance. Moreover, because altering definitions of constraints and activities can be part of the change, this may influence the instance decomposition in multiple ways. Examples of some possible scenarios are shown in Figure 6.34. Manipulating constraints (cf. Figure 6.34(a)) may influence execution of activities in several ways: (1) execution of a decomposed activity is no longer possible, (2) execution of a decomposed activity becomes a necessity, or (3) the moment at which a decomposed activity can be executed changes. Changes involving activities can also influence the decomposition of an instance: (1) the YAWL subprocess can be changed for a decomposed activity (cf. Figure 6.34(b)), (2) a decomposed activity can be removed from the instance (cf. Figure 6.34(c)), (3) a decomposed activity can be changed into a ‘standard’ one (cf. Figure 6.34(d)), (4) a ‘standard’ activity can be changed into a decomposed one (cf. Figure 6.34(e)), and (5) a new decomposed activity can be added to the instance (cf. Figure 6.34(f)).

YAWL also contributes to dynamic decompositions because decomposition of a YAWL model might vary between instances of this model. As shown in Figure 6.35, it is possible to request execution of an instance of a specific DECLARE

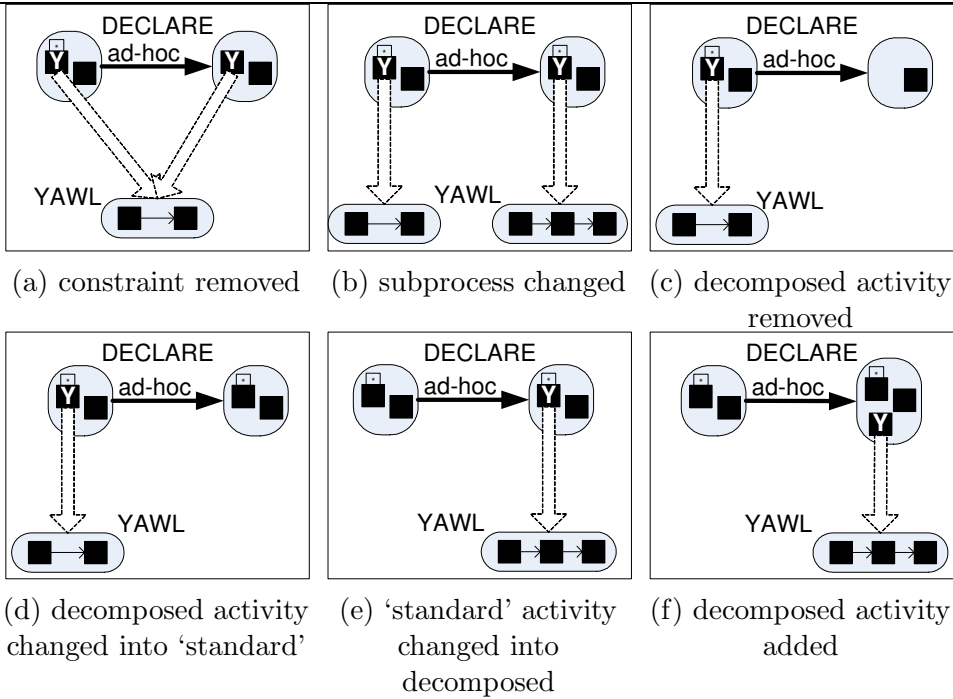


Figure 6.34: Ad-hoc change of decompositions in DECLARE

model, by using the special 'declaremodel' data element. Depending on the value of this data element, the activity can be decomposed into different DECLARE models, as Figure 6.35 shows. Although all three instances refer to the same YAWL model, the activity *D* is decomposed into *Model A*, *Model B* or a manually selected *Model C*. Thus, by assigning different values to the 'declaremodel' data element, YAWL users can determine at run-time to which DECLARE model an activity should be decomposed.

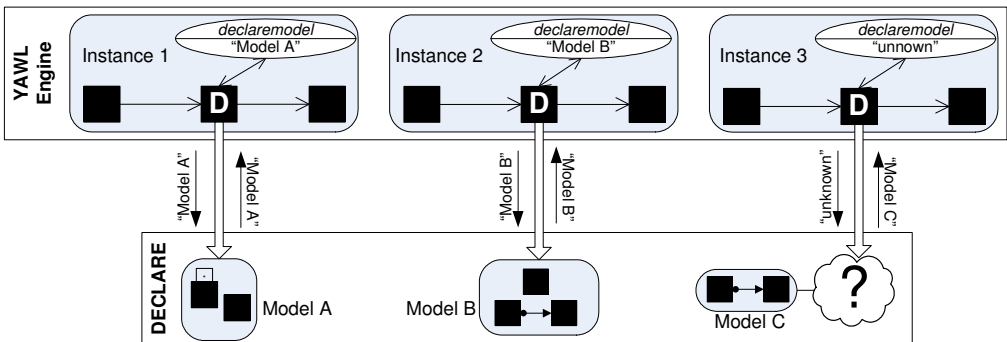


Figure 6.35: Decompositions of three instances of one process model in YAWL

6.11.3 Integration of Even More Approaches

YAWL is a service oriented system and it allows for integration of an arbitrary number of custom services [11,23,32]. For example, the *Worklet Dynamic Process Selection Service* (*Worklet Service*) is also a custom YAWL service [41, 44, 45]. The Worklet Service dynamically substitutes a YAWL activity with a new instance of a contextually selected YAWL process, i.e., *worklet* [41, 44, 45]. Figure 6.36 shows the architecture of the Worklet Service [41, 44, 45]. The decision about which worklet to select for a YAWL activity is made based on the *activity data* and the *worklet repository*. The worklet repository consists of existing *YAWL process models* that can be selected as worklets, *ripple down rules (RDRs)* used to select a worklet based on the activity data, and process and audit *logs*. The RDR Editor is used to create new or alter existing RDRs based on the logs. The RDR editor may communicate with the worklet service in order to override worklet selections based on the rule set additions. In this manner, the Worklet Service allows for dynamic compositions of YAWL instances of various YAWL process models based on the instance data and the rules induced from past executions.

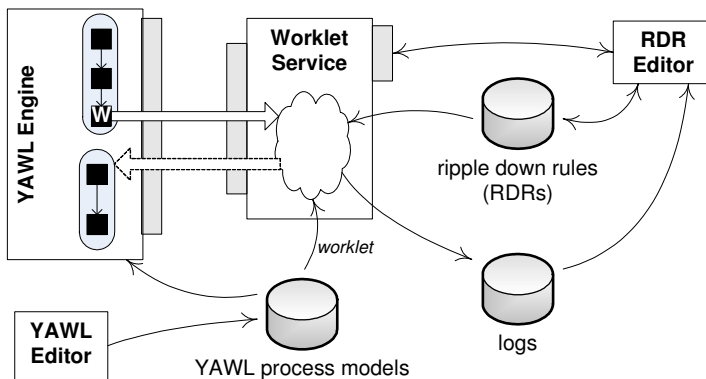


Figure 6.36: The worklet service in YAWL

The service oriented architecture of the YAWL system enables integration of multiple workflow approaches. Any application that can act as a YAWL custom service can join the integration. For example, approaches of the procedural YAWL, constraint-based DECLARE and dynamic Worklet Service can easily be combined, as Figure 6.37 shows. Thick lines represent activities delegated to external applications and thin lines activities offered to users for execution. Due to this integration, the selection of the appropriate approach is not an *exclusive* choice. Instead, an organization can combine multiple approaches on various abstraction (i.e., decomposition) levels. This way the initial idea expressed in Figure 6.27 is realized.

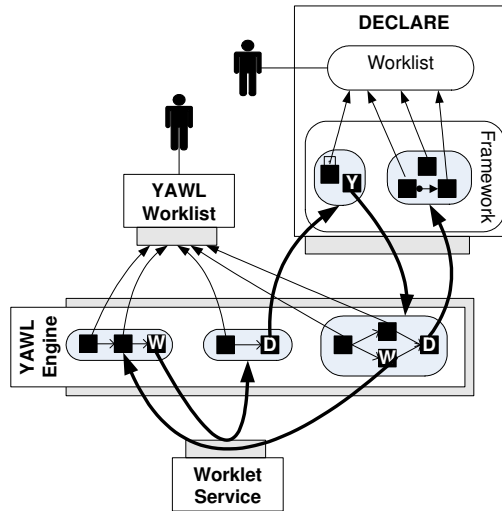


Figure 6.37: The worklet service in YAWL

6.12 Summary

DECLARE is a prototype of a workflow management system based on the constraint approach presented in Chapter 4. An arbitrary constraint-based language can be defined by creating constraint templates. These templates are used in DECLARE models to create constraints. Instances of DECLARE models can be launched and executed. DECLARE supports advanced features like model verification and ad-hoc change of instances. Although it focuses on the control-flow perspective (cf. Section 3.1.1), DECLARE also provides a basic support for the resource and data perspectives. Currently, DECLARE supports LTL-based constraint languages (cf. Chapter 5). However, the tool is implemented in such a way that it can be extended to support other languages suitable for the constraint-based approach. This was demonstrated in Section 6.10.

Integration of DECLARE and the YAWL system allows for combining of various approaches. For example, procedural YAWL processes can be combined with constraint-based DECLARE processes. In this manner, organizations do not need to make a ‘binary choice’ and can combine different approaches. As shown, various approaches can be integrated in a single business process. Dynamic compositions of DECLARE and YAWL processes allow for definition the exact execution of an instance at run-time. This provides *flexibility by underspecification* [125, 226–228] (cf. Section 3.2.2).

DECLARE has more to offer than the features presented in this section. Chapter 7 describes how DECLARE templates and models can be used for *process mining* [28] and generation of run-time recommendations [258] for users with the help of the ProM tool [8, 27].

Chapter 7

Using Process Mining for the Constraint-Based Approach

The idea of process mining is to discover, monitor and improve ‘real’ business processes (i.e., not assumed processes) by extracting knowledge from event logs [28]. Figure 7.1 shows the role of process mining in context of Business Process Management. Process mining can provide different types of analysis, as shown with thick lines in Figure 7.1. First, there are many process mining techniques that enable *discovery of process models* from event logs. Second, techniques for *conformance checking* aim at comparing a given process model with a given event log and judging to which extent these two conform to each other. Third, process mining techniques can be used for log-based verification of processes against some (un)wanted properties by analyzing event logs. Fourth, execution of processes can be supported with *recommendations* generated based on event logs.

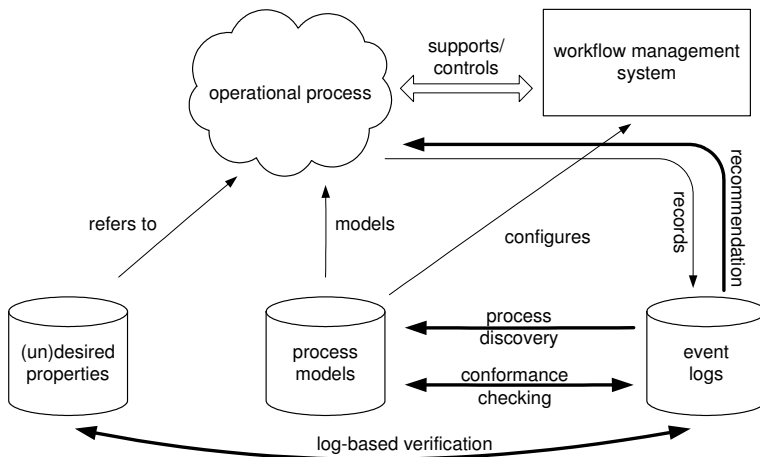


Figure 7.1: Process mining [89]

In this chapter we will describe how process mining can be used in the context of our constraint-based approach (cf. Chapter 4). In Section 7.1 we present the ProM framework [8, 91], a process mining tool that includes a wide variety of plug-ins that support various types of mining techniques. In Section 7.2 we present the LTL Checker, which is a ProM plug-in able to verify event logs against properties specified in LTL [74]. In Section 7.3 we present the SCIFF language as a powerful declarative language that can be used for specification, verification, monitoring, discovery, etc. Two ProM plug-ins use SCIFF language: SCIFF Checker and DecMiner. First, the SCIFF Checker for verification of event logs against properties specified in the SCIFF language is presented in Section 7.3.1. Second, Section 7.3.2 presents the DecMiner, which is able to discover constraint-based SCIFF models. In Section 7.4 we present how process mining techniques can be used for generating recommendations that provide support for users during execution of process instances. We also describe how DECLARE uses the Log-based Recommendations plug-in in ProM to provide run-time recommendations for users. Finally, Section 7.5 summarizes this chapter.

7.1 Process Mining with the ProM Framework

The ProM framework [91] is an open-source infrastructure for process mining techniques. ProM is available as open source software (under the Common Public License, CPL [13]) and can be downloaded from [8]. It has been applied to various real-life processes, ranging from administrative processes and health-care processes to the logs of complex machines and service processes. ProM is plug-able, i.e., people can plug-in new pieces of functionality. Some of the plug-ins are related to model transformations and various forms of model analysis (e.g., verification of soundness, analysis of deadlocks, invariants, reductions, etc.). Most of the plug-ins, however, focus on a particular process mining technique. Currently, there are more than 200 plug-ins of which about half are mining and analysis plug-ins.

Event logs in MXML format are a starting point for ProM [90]. The MXML format is system-independent and by using ProMimport it is possible to extract logs from a wide variety of systems, i.e., systems based on products such as Staffware [238], FLOWer [180], YAWL [11, 23, 32, 210, 212], etc. DECLARE creates event logs in MXML format, i.e., the execution of instances is recorded in the standard format for ProM. As shown in Figure 7.2, the *Framework* tool logs all events that users trigger while executing instances via their *Worklists*.

Table 7.1 shows a part of the MXML log file created by DECLARE. This MXML file contains information about executed instances of the Fractures Treatment model (cf. Figure 6.5 on page 167). Each event that occurred in an instance is recorded as an *AuditTrailEntry*, together with the time stamp, possible data elements and the user who triggered the event. For example, the entry in line 16

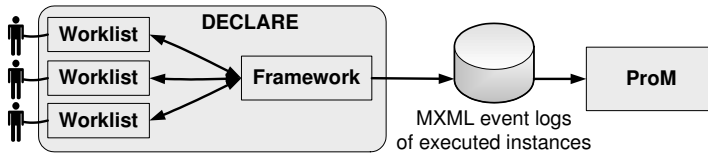


Figure 7.2: Event logs, DECLARE and ProM

specifies that *administrator* completed activity *examine patient* at time *2007-12-07T11:04:50.745+01:00* (cf. fields *Originator*, *EventType*, *WorkflowModelElement* and *Timestamp*, respectively). Values of data elements referring to patient information (i.e., *age*, *diagnosis* and *name*) are also recorded.

Table 7.1: Part of an MXML log file created by DECLARE

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xsi:noNamespaceSchemaLocation="WorkflowLog.xsd"><Source program="MXMLib"/>
4   <Process id="Fractures Treatment">
5     <ProcessInstance id="1">
6       <AuditTrailEntry>
7         <Data>
8           <Attribute name="workItemId">1</Attribute>
9         </Data>
10        <WorkflowModelElement>examine patient</WorkflowModelElement>
11        <EventType>start</EventType>
12        <Timestamp>2007-12-07T11:04:35.325+01:00</Timestamp>
13        <Originator>administrator</Originator>
14      </AuditTrailEntry>
15      <AuditTrailEntry>
16        <Data>
17          <Attribute name="age">9</Attribute>
18          <Attribute name="diagnosis">broken arm</Attribute>
19          <Attribute name="name">Joe Smith</Attribute>
20          <Attribute name="workItemId">1</Attribute>
21        </Data>
22        <WorkflowModelElement>examine patient</WorkflowModelElement>
23        <EventType>complete</EventType>
24        <Timestamp>2007-12-07T11:04:50.745+01:00</Timestamp>
25        <Originator>administrator</Originator>
26      </AuditTrailEntry>
27      <AuditTrailEntry>
28        <Data>
29          <Attribute name="workItemId">2</Attribute>
30        </Data>
31        <WorkflowModelElement>prescribe sling</WorkflowModelElement>
32        <EventType>start</EventType>
33        <Timestamp>2007-12-07T11:04:55.041+01:00</Timestamp>
34        <Originator>administrator</Originator>
35      </AuditTrailEntry>
36      ...
37    </ProcessInstance>
38  </Process>
39 </WorkflowLog>

```

Because DECLARE logs are stored in the MXML format, these logs can be directly accessed by ProM. Figure 7.3 shows an MXML file (part of this file is shown in Table 7.1) loaded into ProM. This event log contains three instances of the Fractures Treatment model presented in Figure 6.5 on page 167. For each instance, ProM presents the sequence of executed events, as stored in the referring MXML file shown in Table 7.1.

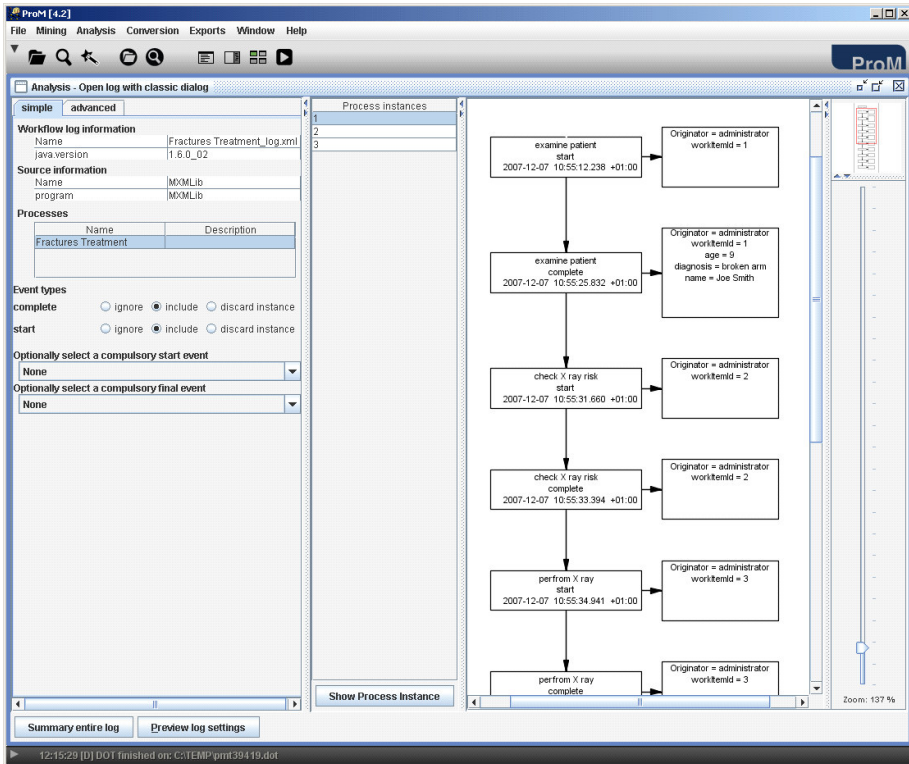


Figure 7.3: DECLARE event log loaded in ProM

One of the basic features in ProM is the process discovery, i.e., deriving a model from some event log. This model is typically a process model. However, ProM offers many more interesting process mining techniques [28, 79, 89, 91, 206, 207, 263]. For example, there are also techniques to discover organization models, social networks, more data-oriented models such as decision trees, etc. Figure 7.4 shows the result of three alternative process discovery algorithms: (1) the α miner shows the result in terms of a Petri net, (2) the multi-phase miner shows the result in terms of an EPC, and (3) the heuristics miner shows the result in terms of a heuristics net.

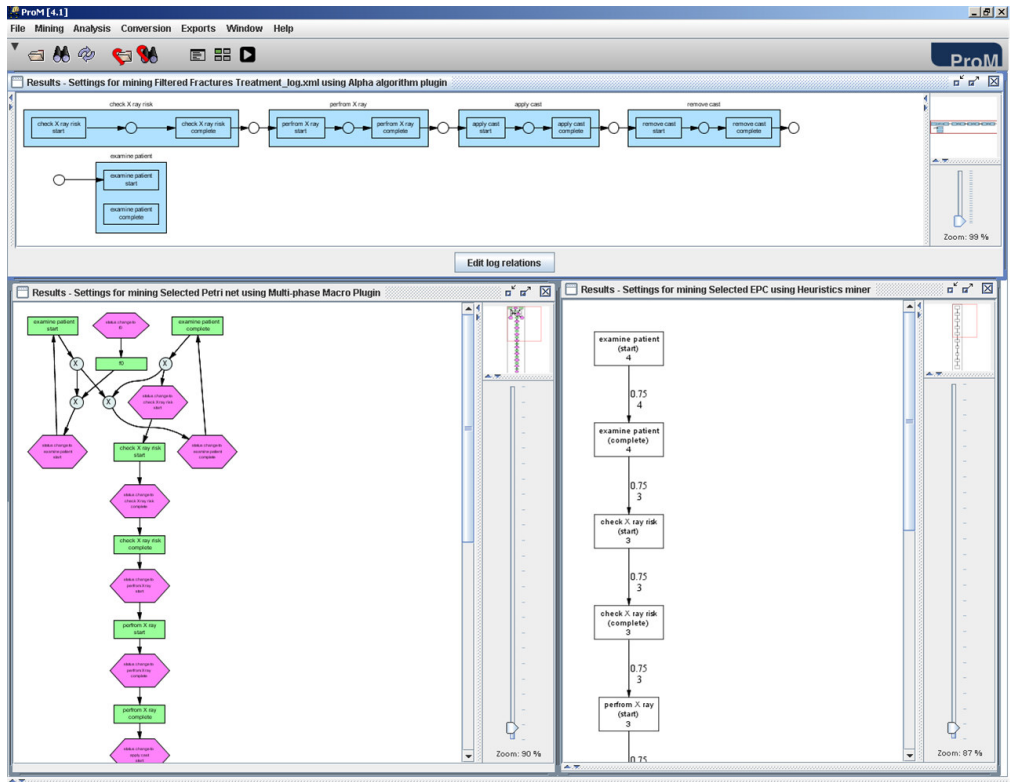


Figure 7.4: The output of three process discovery algorithms supported by ProM when analyzing the event log shown in Figure 7.3

7.2 Verification of Event Logs with LTL Checker

One of the plug-ins offered by ProM is the so-called *LTL Checker* [25]. The LTL Checker offers an environment to check (verify) predefined properties with respect to some event log in MXML format. For each process instance, it is determined whether the given property holds or not, i.e., given a property all process instances are partitioned in two classes: *conforming* and *non-conforming*. If the property holds in an instance, then this instance is classified as a conforming one. If the property does not hold in an instance, then this instance is classified as a non-conforming one.

A-posteriori verification of properties can be particularly useful in the constraint-based approach. Because, generally, this approach allows for more flexibility and users can execute instances in various ways, a-posteriori verification can provide insights into the exact way the processes are being executed in practice. For example, while executing instances of the Fractures Treatment model presented in Section 5.5, users have a lot of freedom to decide which activity to execute and when to execute it. A-posteriori analysis of the Fractures Treat-

ment instances can provide many useful information. Consider, for example, the verification of the Fractures Treatment instances processed in the past against the following two properties: (1) ‘activity *perform surgery* was executed in the instance’, and (2) ‘activity *prescribe rehabilitation* was executed after activity *perform surgery* in the instance’. On the one hand, the verification could show that the *first property* holds in 80% of the instances processed in the previous year, while it holds in only 40% of the instances processed in the year before. This can be an indication that the hospital should hire more surgeons. On the other hand, the verification could show that the *second property* does not hold in 90% of the instances, i.e., the medical staff violated the optional constraint from the Fractures Treatment model in 90% of the instances. This result might indicate that this constraint should either be removed from the model or made mandatory.

Properties are specified as predefined parameterized LTL [74] expressions in the LTL Checker. If the instance trace satisfies the LTL formula, then this instance is classified as conforming. If the instance trace does not satisfy the LTL formula, then this instance is classified as non-conforming. Recall that we defined an instance as a pair of a trace and a model in Definition 4.4.1 on page 98. The LTL Checker considers only the instance trace stored in an MXML log file, while the instance model remains irrelevant (or even unknown). Consider, for example, the two instances ci_1 and ci_2 presented in Table 7.2. The first property (of the two properties) mentioned in the previous paragraph can be specified with the LTL formula $f = \diamond(\text{perform surgery}, t_c)$ specifying that event $(\text{perform surgery}, t_c)$ must occur at least once (cf. Section 5.1). On the one hand, instance ci_1 is conforming to formula f because its trace σ_1 satisfies formula f , i.e., $\sigma_1 \models f$ because $\sigma_1[4] = (\text{perform surgery}, t_c)$. On the other hand, instance ci_2 is non-conforming to formula f because its trace σ_2 does not satisfy formula f , i.e., $\sigma_2 \not\models f$ because $(\text{perform surgery}, t_c) \notin \sigma_2$. Note that instance models cm_1 and cm_2 do not influence the classification and, thus, do not have to be known when determining if traces σ_1 and σ_2 satisfy formula f .

Table 7.2: A conforming and a non-conforming instance for formula $f = \diamond(\text{perform surgery}, t_c)$

$ci_1 = (\sigma_1, cm_1), ci_1 \in \mathcal{U}_{ci}$	$ci_2 = (\sigma_2, cm_2), ci_2 \in \mathcal{U}_{ci}$
$\sigma_1[1] = (\text{examine patient}, t_s)$	$\sigma_2[1] = (\text{examine patient}, t_s)$
$\sigma_1[2] = (\text{examine patient}, t_c)$	$\sigma_2[2] = (\text{examine patient}, t_c)$
$\sigma_1[3] = (\text{perform surgery}, t_s)$	$\sigma_2[3] = (\text{apply cast}, t_s)$
$\sigma_1[4] = (\text{perform surgery}, t_c)$	$\sigma_2[4] = (\text{apply cast}, t_c)$
$\sigma_1[5] = (\text{examine patient}, t_s)$	$\sigma_2[5] = (\text{remove cast}, t_s)$
$\sigma_1[6] = (\text{examine patient}, t_c)$	$\sigma_2[6] = (\text{remove cast}, t_c)$
$\sigma_1[7] = (\text{prescribe rehabilitation}, t_s)$	$\sigma_2[7] = (\text{examine patient}, t_s)$
$\sigma_1[8] = (\text{prescribe rehabilitation}, t_c)$	$\sigma_2[8] = (\text{examine patient}, t_c)$
ci_1 is conforming , i.e., $\sigma_1 \models f$	ci_2 is non-conforming , i.e., $\sigma_2 \not\models f$

recall that $t_s, t_c \in \mathcal{T}$ are event types such that $t_s = \text{started}$ and $t_c = \text{completed}$

The LTL Checker is currently the most important ProM plug-in for the DECLARE prototype (cf. Chapter 6) because both applications use LTL. The LTL Checker can be used for verification of (DECLARE) instances against properties specified in terms of constraint templates (e.g., the ConDec templates presented in Section 5.2) or constraints (e.g., the six constraints from the Fractures Treatment model presented in Section 5.5). The remainder of this section is organized as follows. In Section 7.2.1 we shortly describe the default LTL Checker as a ‘stand-alone’ application and in Section 7.2.2 we present how the LTL Checker and DECLARE can be used together.

7.2.1 The Default LTL Checker

The default version of the LTL Checker contains 60 predefined typical properties one may want to verify using the LTL Checker (e.g., the so-called *4 eyes principle*) [25]. These can be used without any knowledge of the LTL language. In addition the user can define new sets of properties. Each property is specified in terms of an LTL expression. Formulas may be parameterized, are reusable, and carry explanations in HTML format. This way both experts and novices may use the LTL Checker.

LTL used in the LTL Checker has more expressive power than in the ConDec language and the DECLARE prototype (cf. Chapters 5 and 6). While ConDec and DECLARE use LTL to specify relations between activities and event types, the LTL Checker allows referring to *activities*, *event types*, *users*, *time stamps* and *data elements*. In fact, in the LTL Checker, it is possible to refer to any element of audit trail entries appearing in MXML logs [25] (see, e.g., the MXML log presented in Table 7.1). Moreover, it is possible to parameterize properties, which enables reusability.

Properties can be loaded into the LTL Checker via plain text files. Table 7.3 shows a part of such a file. Each property is specified as a formula. For example, formula for the *person_P_executed_activity_A* property in line 11 has two parameters: parameter *P* represents a user and parameter *A* represents an activity. After that, some information about the formula (i.e., a short description and a list of parameters) and the LTL formula are given. This formula can be used to check if a user (*P*) executed an activity (*A*) in an instance.

The LTL Checker presents the description and parameters of loaded formulas, while the LTL expression remains hidden from the users. Figure 7.5 shows how the LTL Checker presents formulas from the file shown in Table 7.3. A value for each parameter of the selected formula must be manually specified. In this manner, formulas can be reused for the verification of event logs using various parameter settings. In the example shown in Figure 7.5, the parameters for the selected *person_P_executed_activity_A* property are set to *John Smith* and *approve transaction*. However, any other person and activity could have been selected (e.g., user *Mary Jones* and activity *perform surgery*).

Table 7.3: A formula involving users in the LTL Checker

```

1 # version      : 0.1
2 # date        : 21-01-2008 10:58:30:112
3 # author      : Manually
4 set ate.EventType;
5 set ate.WorkflowModelElement;
6 set ate.Originator;
7 rename ate.EventType as event;
8 rename ate.WorkflowModelElement as activity;
9 rename ate.Originator as person;
10 #####
11 formula person_P_executed_activity_A( P: person, A: activity ) :=
12 { <h2>Did person <b>P</b> execute activity <b>A</b>?</h2>
13   <p> Arguments:<br>
14     <ul>
15       <li><b>P</b> of type set (<i>ate.Originator</i>)</li>
16       <li><b>A</b> of type set (<i>ate.WorkflowModelElement</i>)</li>
17     </ul>
18   </p> }
19   <>((( person == P /\ activity == A ) /\ event == "complete"));
20 #####
21 formula activity_A_executed( A: activity ) :=
22 { <h2>Was activity <b>A</b> successfully executed?</h2>
23   <p> Arguments:<br>
24     <ul>
25       <li><b>A</b> of type set (<i>ate.WorkflowModelElement</i>)</li>
26     </ul>
27   </p> }
28   <>(( activity == A /\ event == "complete"));
29 ...

```

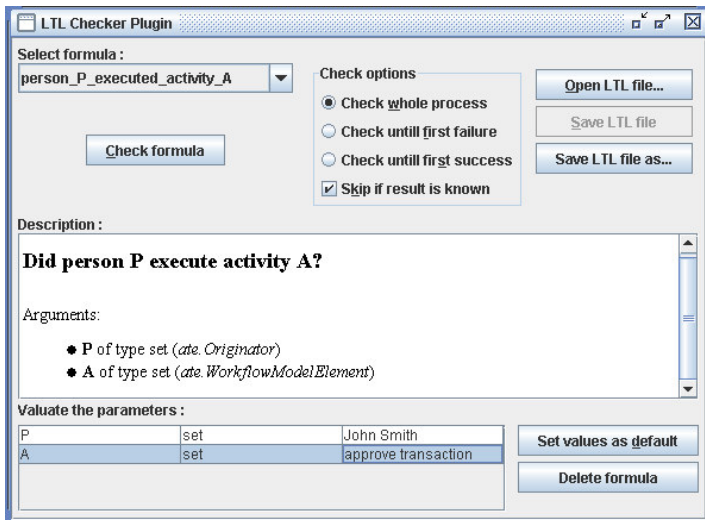


Figure 7.5: LTL Checker: ‘Did John Smith execute activity approve transaction?’

7.2.2 Combining the LTL Checker and declare

DECLARE allows for an automatic export of its LTL formulas to the LTL Checker in two ways, as Figure 7.6 shows. Templates belonging to a language or constraints from a model can be exported to a file that can be loaded into the LTL Checker. In this manner, DECLARE templates and constraints can be reused as verification properties in the LTL Checker. Moreover, the LTL Checker can use DECLARE templates and constraints for the verification of either MXML log files created either by DECLARE (and thus containing information about instances executed in DECLARE) or by any other system.

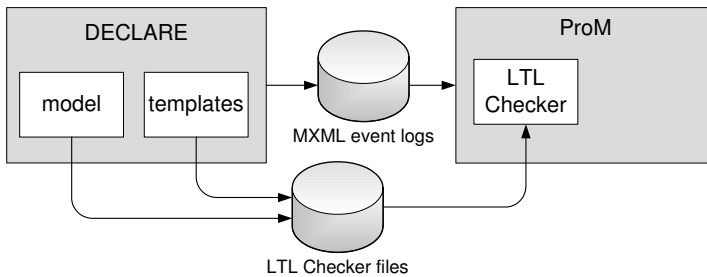


Figure 7.6: Combining DECLARE and the LTL Checker

Table 7.4 shows a part of the LTL Checker file that DECLARE created for all templates in the ConDec language. When exporting templates, DECLARE creates parameterized formulas that can be loaded into the LTL Checker. One formula is created for each template. Each parameter in the created formula refers to one of the template’s parameters (cf. sections 5.2 and 6.2). For example, the *response* formula in line 22 refers to the *response* template presented in Section 5.2.2. Therefore, this formula contains two parameters referring to activities *A* and *B*. After that, some information about the formula (i.e., a short description and a list of parameters) and the template’s LTL formula are given. Note that the LTL Checker and DECLARE use slightly different syntaxes for LTL expressions. During the export, DECLARE automatically converts template’s formulas into the syntax recognized by the LTL Checker.

Figure 7.7 shows how the LTL Checker presents formulas from the file shown in Table 7.4. In the LTL Checker, the parameters for the *response* template can be set to *perform surgery* and *prescribe rehabilitation*, or for any other two activities. For example, it can be also used to verify the *response* formula between activities *curse* and *pray* on event logs containing instances of the model presented in Figure 6.11(a) on page 174.

Besides constraint templates, DECLARE can also export constraints from models to files that can be loaded into the LTL Checker. Table 7.5 shows a part of the LTL Checker file that DECLARE created from constraints in the Fractures Treatment model presented in Figure 6.5 on page 167. While exporting templates cre-

Table 7.4: A part of the file generated by DECLARE while exporting ConDec templates to ProM

```

1 # version      : 0.1
2 # date        : 07-12-2007 11:35:46:414
3 # author      : DECLARE
4 set ate.EventType;
5 set ate.WorkflowModelElement;
6 rename ate.EventType as event;
7 rename ate.WorkflowModelElement as activity;
8 #####
9 formula init ( A: activity ) :=
10 {
11   <h2>init</h2>
12   <p> A has to be the first activity.</p>
13   <p> Arguments:<br>
14     <ul>
15       <li><b>A</b> of type set (<i>ate.WorkflowModelElement</i></li>
16     </ul>
17   </p>
18 }
19 ((( activity==A /\ event=="start") \/ (activity==A /\ event=="ate_abort"))
20 _U (activity==A /\ event=="complete"));
21 #####
22 formula response ( A: activity, B: activity ) :=
23 {
24   <h2>response</h2>
25   <p> Whenever activity B is executed,
26     activity A has to be eventually executed afterwards.</p>
27   <p> Arguments:<br>
28     <ul>
29       <li><b>A</b> of type set (<i>ate.WorkflowModelElement</i></li>
30       <li><b>B</b> of type set (<i>ate.WorkflowModelElement</i></li>
31     </ul>
32   </p>
33 }
34 [(((activity==A /\ event=="complete") -> <>((activity==B /\ event=="complete"))));
35 #####
36 ...

```

ates parameterized formulas, exporting constraints creates formulas without parameters. For example, formula *response_performsurgery_prescriberehabilitation* in line 19 in Table 7.5 does not have any parameters. Instead of the parameters, it uses the actual activities involved in the constraint, i.e., *perform surgery* and *prescribe rehabilitation*.

Figure 7.8 shows how the LTL Checker presents constraints from the Fractures Treatment model presented in Figure 6.5 on page 167 in the file presented in Table 7.5. Formulas in the generated LTL Checker file do not have parameters because constraints involve real activities from the model. Therefore, this type of DECLARE export can not be reused in verification and can only be used to verify event logs against specific constraints.

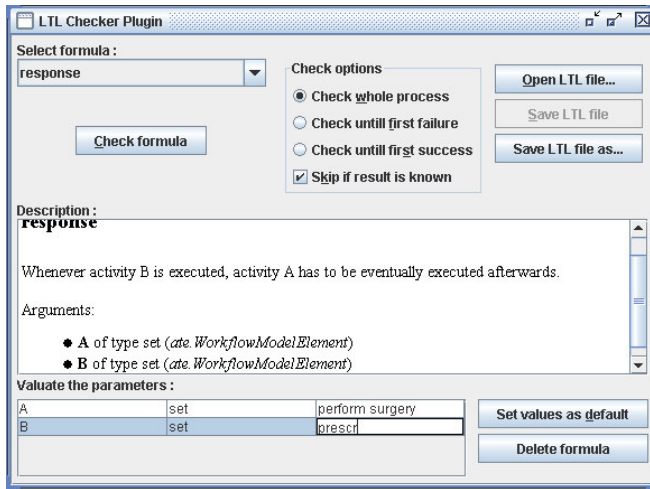


Figure 7.7: ConDec templates in LTL Checker

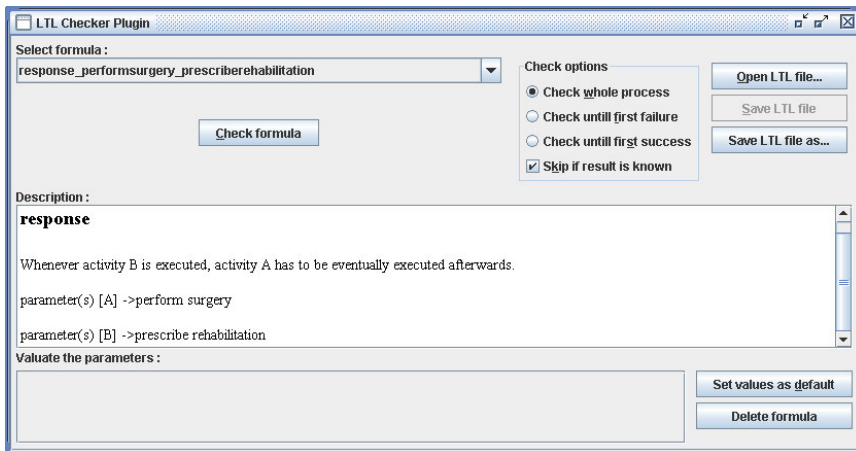


Figure 7.8: Constraints from the Fractures Treatment model in LTL Checker

7.3 The SCIFF Language

The SCIFF framework [48, 49] is based on abductive logic programming [144]. SCIFF is originally thought for the specification and verification of global interaction protocols in open multi-agents systems, which share many aspects with the service-oriented computing setting [59]. SCIFF envisages a powerful logic-based language, with a clear declarative semantics, for specifying social interaction, and is equipped with a proof procedure capable to check at run-time or a-posteriori whether a set of interacting entities is behaving in a conforming manner with respect to a given specification. The SCIFF language is another language that

Table 7.5: A part of the file generated by DECLARE while exporting the Fractures Treatment model to ProM

```

1 # version      : 0.1
2 # date        : 07-12-2007 11:35:46:414
3 # author      : DECLARE
4 set ate.EventType;
5 set ate.WorkflowModelElement;
6 rename ate.EventType as event;
7 rename ate.WorkflowModelElement as activity;
8 #####
9 formula init_examinepatient () :=
10 {
11     <h2>init</h2>
12     <p> A has to be the first activity.</p>
13     <p> parameter(s) [A] ->examine patient</p>
14     <p> type: mandatory </p>
15 }
16 (((activity=="examine patient" /\ event=="start") /\ (activity=="examine patient" /\
17 event=="ate_abort")) _U (activity=="examine patient" /\ event=="complete"));
18 #####
19 formula response_performsurgery_prescriberehabilitation () :=
20 {
21     <h2>response</h2>
22     <p> Whenever activity B is executed,
23         activity A has to be eventually executed afterwards.</p>
24     <p> parameter(s) [A] ->perform surgery</p>
25     <p> parameter(s) [B] ->prescribe rehabilitation</p>
26     <p> type: optional </p>
27 }
28 [](((activity=="perform surgery" /\ event=="complete") ->
29 <>((activity=="prescribe rehabilitation" /\ event=="complete"))));
30 #####
31 ...

```

can be used for process mining in the context of the constraint-based approach. SCIFF is used in ProM to verify and discover constraints. It provides an alternative for LTL with different capabilities.

SCIFF abstracts from the notion of event. Instead of using variables directly referring to events, it uses the more generic notion of a predicate. For example, in the context of workflow management systems, we can define events as a fact that user *Originator* performed event of the type *Type* on activity *Activity*, denoted by *perform(Activity, Type, Originator)*. Using this notation, the fact that *John Smith* started activity *examine patient* is denoted by *perform(examine patient, started, John Smith)*. Note that it is also possible to leave some parameters unspecified, i.e., *perform(examine patient, started, Originator)* denotes that some user started activity *examine patient*.

There are three basic operators that can be used in SCIFF. First, operator $\mathbf{H}(Event, Time)$ denotes that *Event* happened at time *Time*. Second, expectation that *Event* should happen at time *Time* is denoted by $\mathbf{E}(Event, Time)$. Finally, negative expectation $\mathbf{EN}(Event, Time)$ denotes that *Event* is ex-

pected not to happen at time $Time$. Note that $Event$ and $Time$ can be variables, or they could be grounded to a specific value. For example, expression $\mathbf{H}(\text{perform}(\text{examine patient, completed, Originator}), T_e) \wedge T_e > 10$ matches with any completion of activity *examine patient* at a time greater than 10 units, performed by a whatsoever *Originator*. On the other hand, expression $\mathbf{E}(\text{perform}(\text{decide, completed, Originator}), T_e) \wedge T_e < 100$ represents that activity *decide* must be completed within 100 time units.

The three basic operators can be used to specify SCIFF *integrity constraints*, i.e., rules that relate events that *already happened* and events that are *expected to happen in the future*. These rules are represented as forward rules of the form $Body \Rightarrow Head$ [49]. Consider, for example, the response constraint between activities *perform surgery* and *prescribe rehabilitation* from the Fractures Treatment model shown in Figure 5.17 on page 145 and Table 5.8 on page 146. This constraint can be denoted as SCIFF formula $\mathbf{H}(\text{perform}(\text{perform surgery, completed, Originator}), T_S) \Rightarrow \mathbf{E}(\text{perform}(\text{prescribe rehabilitation, completed, Originator}), T_R) \wedge T_R > T_S$. In fact, all ConDec templates (cf. Section 5.2) can be specified in the SCIFF language [70]. Note that [70] presents the mapping between SCIFF and the DecSerFlow language [37, 38]. DecSerFlow is a constraint-based language based on LTL and is very similar to ConDec and also supported by DECLARE¹.

Although both SCIFF and LTL-based ConDec are both suitable for declarative specifications, there are some differences between these languages. First, while automata generated from LTL [74, 111, 112, 158] provide a deadlock-free execution mechanism for ConDec models (cf. Chapter 5), there exists no method for a deadlock-free execution of SCIFF models. Second, the SCIFF language is more expressively powerful than ConDec with respect to the data and time related aspects. Consider, for example, the aspect of time. While LTL implicitly models the concept of time via its temporal operators, SCIFF specifies time by explicitly constraining time variables. This allows for the specification of complex time constraints. Consider, for example, the ConDec *response* template (cf. Section 5.2.2), which specifies that activity B has to be completed after activity A without considering time interval between executions of activities A and B . The SCIFF formulas in Table 7.6 show several examples of how the *response* template can be extended with different deadlines and Figure 7.9 shows illustrations of these deadlines. The first formula represents the classical *response* template without a deadline, i.e., it is important only that B is completed at any moment after A . The following three formulas extend the ‘plain’ *response* template with three different deadlines. The second formula requires that B is completed not earlier than N time units after A is completed. The third formula specifies that B has to be completed not later than N time units after A is completed. Finally, the last formula requires B to be completed not earlier than N and not later than

¹DecSerFlow is tailored towards the specification of web-services and their choreographies.

M time units after A is completed.

Table 7.6: Deadlines in the *response* template in SCIFF

1. no deadline: $\mathbf{H}(\text{perform}(A, t_c, \text{Originator}), T_A) \Rightarrow \mathbf{E}(\text{perform}(B, t_c, \text{Originator}), T_A) \wedge T_B > T_A$
2. after N time units: $\mathbf{H}(\text{perform}(A, t_c, \text{Originator}), T_A) \Rightarrow \mathbf{E}(\text{perform}(B, t_c, \text{Originator}), T_A) \wedge T_B > T_A + N$
3. within N time units: $\mathbf{H}(\text{perform}(A, t_c, \text{Originator}), T_A) \Rightarrow \mathbf{E}(\text{perform}(B, t_c, \text{Originator}), T_A) \wedge T_B > T_A \wedge T_B < T_A + N$
4. between N and M time units: $\mathbf{H}(\text{perform}(A, t_c, \text{Originator}), T_A) \Rightarrow \mathbf{E}(\text{perform}(B, t_c, \text{Originator}), T_A) \wedge T_B > T_A + N \wedge T_B < T_A + M$

Recall that $t_c \in \mathcal{T}$ is an event type such that $t_c = \text{completed}$.

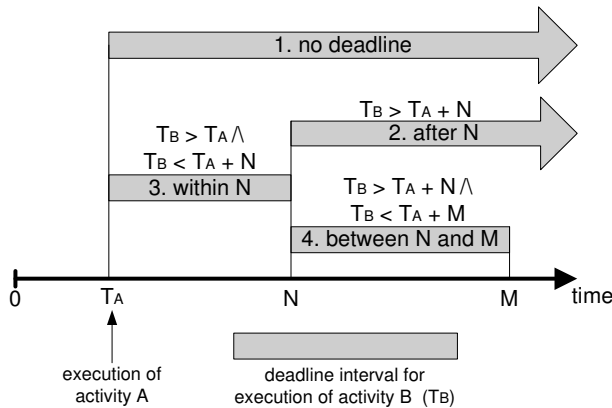


Figure 7.9: Deadlines in SCIFF formulas from Table 7.6

Similarly like LTL, the SCIFF language can be used for process mining in the constraint-based approach. There are two plug-ins in the ProM framework that use SCIFF. First, the SCIFF Checker presented in Section 7.3.1 can be used to verify event logs against SCIFF formulas. Second, DecMiner presented in Section 7.3.2 can be used to learn SCIFF formulas from event logs.

7.3.1 Verification of Event Logs with SCIFF Checker

The *SCIFF Checker* plug-in in ProM is similar to the LTL Checker plug-in both in design and functionality. The SCIFF Checker can verify event logs against properties specified as SCIFF formulas. After an MXML file storing logs of events is loaded into ProM, a SCIFF property that should be verified is selected

in the SCIFF Checker. For example, formula *existence of activity A* can be selected, as shown in the middle screen in Figure 7.10. The verification procedure classifies instances from the referred event log into conforming (i.e., *correct*) and non-conforming (i.e., *wrong*), as shown in the bottom screen in Figure 7.10. In addition, instances for which exceptions occurred during the verification are classified as *exceptional*.

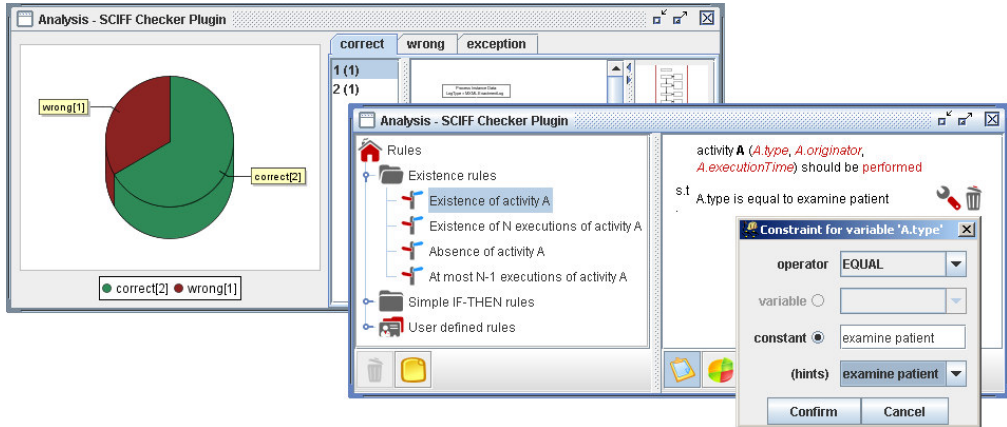


Figure 7.10: The SCIFF Checker plug-in in ProM

Manipulation of parameters in SCIFF Checker is more sophisticated than in the LTL Checker, where for each parameter in the LTL Checker a specific value must be explicitly given (cf. Figure 7.5). In the SCIFF Checker it is, e.g., possible to set the value for the activity name to be equal to or different from a given value, as shown in the top screen in Figure 7.10. The available options depend on the parameter type. For example, the set-up for time parameters is even more elaborate, i.e., it is possible to specify that a time variable is equal, greater or less than a given (absolute or relative) time stamp value.

7.3.2 Discovering Constraints with DecMiner

DecMiner is a ProM plug-in that is able to learn SCIFF formulas from event logs. This plug-in uses a modified algorithm from the field of inductive logic programming for learning models from examples and background knowledge [154–156]. In this approach SCIFF formulas are learned from event logs, which are previously labeled as conforming and non conforming. For example, a hospital can label executed instances of patient’s treatments as normal or too long and then learn a model that discriminates these two classes. The learned model must consist of formulas that all hold for conforming instances. Formulas that do not hold in at least one of the conforming instances are discarded from the learned model [154–156].

After an MXML event log file is loaded in ProM, DecMiner learns a model from this file via three steps, as Figure 7.11 shows. First, all instances from the MXML file must be *classified* or as conforming or non conforming. This classification can be done manually or by previously executing analysis with the SCIFF Checker or the LTL Checker. Second, relevant *activities* are selected. Finally, *templates* (i.e., formulas) that should be considered are selected from a predefined collection. Note that the mapping between the SCIFF language and ConDec and DecSerFlow is already integrated in DecMiner. An existing manual mapping between ConDec templates and SCIFF integrity constraints allows automatic generation of learned models in DECLARE format. In Figure 7.11 DecMiner presents the learned model in the *Designer* component of the DECLARE prototype (cf. Chapter 6). This shows the true integration of the various approaches.

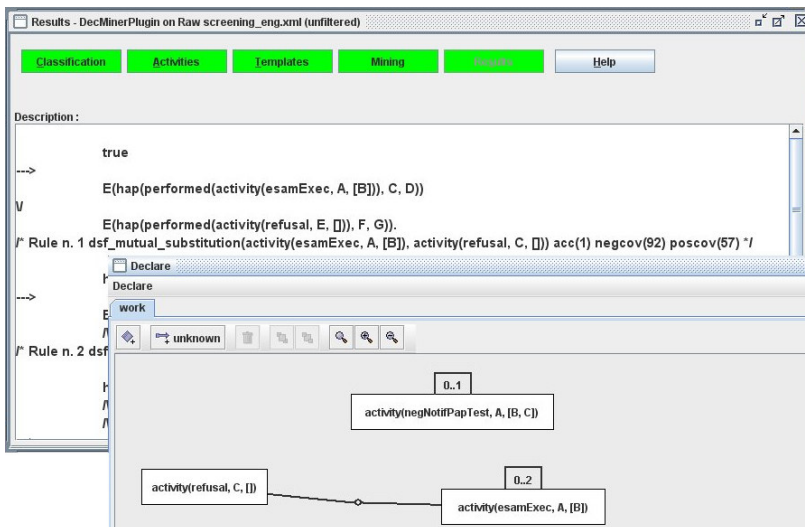


Figure 7.11: The DecMiner plug-in in ProM

7.4 Recommendations Based on Past Executions

While traditional workflow management systems tend to enforce a particular way of working to users, flexible approaches to workflow management systems (e.g., adaptive systems (cf. Section 2.1.5), case-handling systems [195], or our constraint-based approach) aim at shifting the decision making from the system to users. As discussed in Chapter 1, a flexible style of work assumes that end users are both allowed and capable to make good decisions about how to work. However, flexibility usually comes at a cost, i.e. the more flexible a workflow management system is, the less support it provides to its users and hence the more knowledge these users need to have about the process they are a part of.

Also, full support in a workflow management system usually comes at a cost of loosing flexibility, as shown in Figure 7.12.

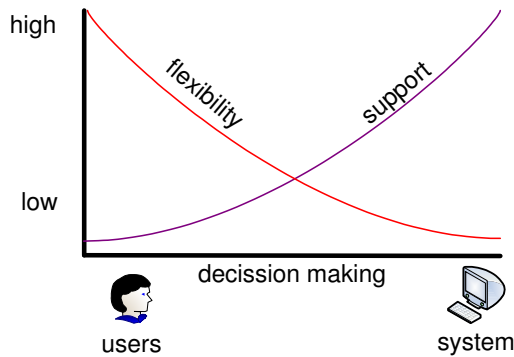


Figure 7.12: Tradeoff: flexibility vs. support [90,258]

Although users of flexible systems have the option to make their own decisions while working, a certain level of support is still necessary. Reasons for this can be various: inexperienced users, exceptional situations, personal preferences, etc. Traditionally, this problem is solved by educating workers (e.g. by making them more process aware), or by having a workflow management system restricting the user and thus sacrificing flexibility. Obviously, both options are costly. Moreover, they both require a process specialist to gain insights in the process supported by the system, either to educate the workers about these processes or to change them into more restrictive ones.

Process mining is by origin a stand-alone application performed *after* the execution, i.e. an event log is taken and used to produce a model (e.g., a process model or social network), to check whether reality fits with the model (cf. the LTL Checker, the Conformance Checker in ProM [25,206]), or to extend an existing model (e.g., building a process model into a simulation model). However, process mining techniques can also be applied to provide *recommendations* to users, while they are executing process instances. A recommendation service more or less applies process mining on-the-fly, i.e. by looking at an event log (set of completed executions) and a current *partial execution*, predictions can be made about the future of the current (partial) instance [258].

Recommendations can be considered as predictions about a current instance, conditioned on the next step that has not been performed yet [258]. For example, given the partial execution of the current Fractures Treatment instance (cf. Figure 6.5 on page 167) and the completed executions of similar instances of the Fractures Treatment model in an event log, it is predicted that this instance will take 90 days if the next step is activity *prescribe sling*. However, if the next step is activity *apply cast*, it will last only 60 days.

Figure 7.13 shows an overview of a recommendation service as it is imple-

mented in the ProM framework and the DECLARE prototype (cf. Chapter 6). However, the same architecture can be achieved by any other process mining tool and any other workflow management system providing some degree of flexibility. The workflow engine creates event logs for executed instances. The recommendation service bases its recommendations on the information in this log. At the moment when the recommendation for an instance is needed, the workflow engine sends the *partial log* of the instance, i.e., a record of all events performed in the instance up to this moment. The recommendation service then answers by sending a recommendation to assist users in choosing the next step(s). Such a recommendation consists of a list of advised next steps (e.g. ‘*examine patient*’) combined with a number of quality attributes (e.g. following this recommendation will lead to a quicker recovery).

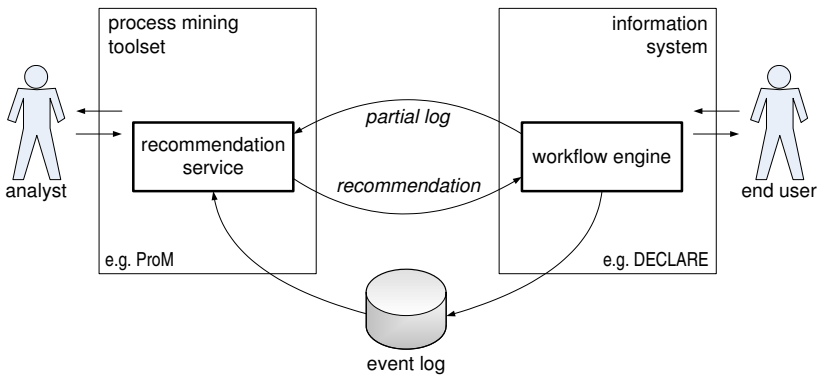


Figure 7.13: An overview of the recommendation service [258]

To decide what the next step in an instance should be, the recommendation service compares the *partial log* of the instance to completed instances in the event log and searches for similar instances. As one can imagine, there can be many criteria for similarity of instances. Figure 7.14 shows several types of simple instance abstractions that can be used as criteria for comparison of instances [258]. If *prefix abstraction* is used, then two instances are similar if their activities are executed in the same order. The *multi-set abstraction* considers instances where the same activities are executed the same number of times in any order as similar instances. Similar instances in the *set abstraction* are instances where the same activities are executed, regardless of how many times and in which order.

Another important concept in the recommendation service is the *goal* of the recommendations. For example, the recommendation can be generated to steer the execution towards the shortest throughput time, towards avoiding executions of a critical activity, minimizing costs, etc.

The Log-Based Recommendations plug-in in ProM is a recommendation service (cf. Figure 7.13). This plug-in allows for selection of a preferred scale and

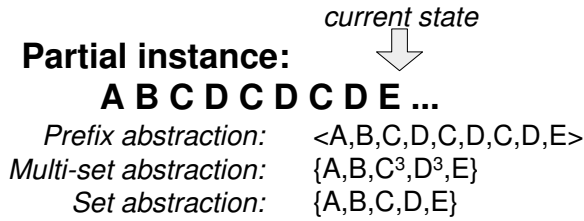


Figure 7.14: Possible abstractions of instances [258]

contributor, as Figure 7.15 shows. The scale refers to the recommendation goal, i.e., the recommendation service in Figure 7.15 generates recommendations that will lead to short execution times of instances because *Duration scale* is selected. The contributor refers to the criteria of instances similarity, i.e., in this case *prefix abstraction* is selected (cf. Figure 7.14). Further on, some additional settings are available for managing and monitoring the performance of the service.

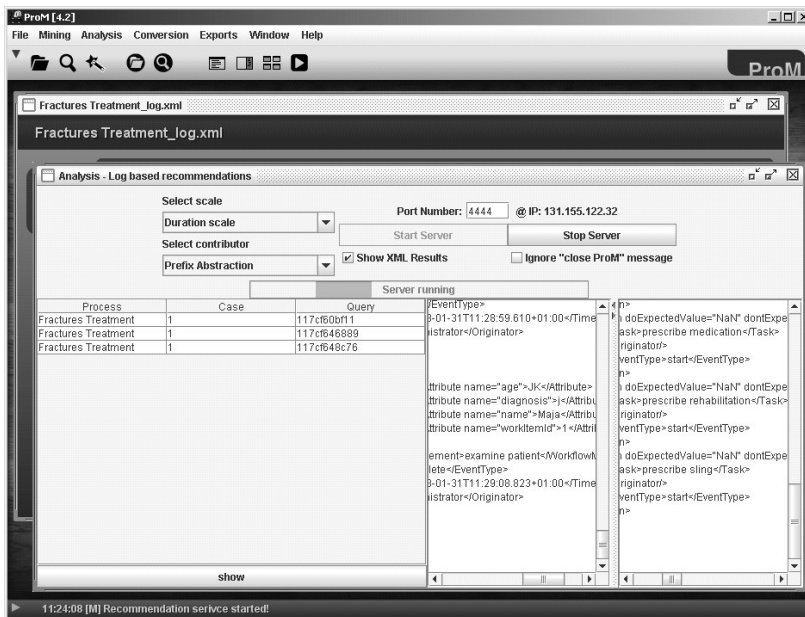


Figure 7.15: The Log-Based Recommendations plug-in in ProM

The DECLARE prototype uses the Log-Based Recommendations plug-in in ProM for providing recommendations to users. In fact, recommendations are optional in DECLARE, i.e., this option can be either turned-on or switched-off in the prototype. The *Framework* component of DECLARE communicates with the Log-Based Recommendations plug-in as its recommendation service. Each time when a user triggers a new event in an instance (by *starting*, *completing*,

or *cancelling* an activity), a new recommendation for that instance is requested from ProM and presented to all users working on that instance. The *Worklist* component presents recommendations in a special panel next to the instance, as Figure 7.16 shows. The recommendation shown in Figure 7.16 is generated for an instance of the Fractures Treatment model (cf. Figure 6.5 on page 167) using the Log-Based Recommendations plug-in and settings shown in Figure 7.15, i.e., the recommendation service scale is *duration* (i.e., the goal is to minimize flow time) and the contributor is *prefix abstraction* (i.e., the recommendation is based on instances with a similar prefix). The current recommendation suggests that the next step in this instance should be *starting* activity *check X ray risk*. Moreover, the recommendation specifies that if activity *check X ray risk* is *started* next, the expected average execution duration for this instance is 34.06 time units. Also, *not starting* activities *examine patient*, *prescribe medication*, *prescribe rehabilitation* and *prescribe sling* will cause the instance to be completed within 34.06 time units. In other words, instances with a similar prefix where activity *check X ray risk* was *started* at this point had a short execution time, and their average execution time was 34.06 time units.

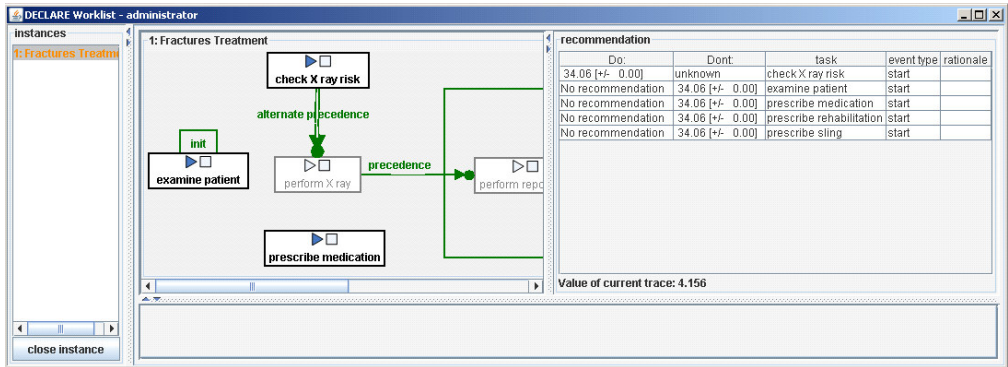


Figure 7.16: DECLARE: presenting recommendations to users

Note that DECLARE presents recommendations purely as additional information in a *Worklist*, i.e., users are not forced to follow recommendations. Instead, they can freely decide what to do, even if this means acting against what is recommended. In this manner, DECLARE offers a significant level of support to its users, without sacrificing the flexibility. It also nicely illustrates that process mining and flexibility fit well together. Allowing for a lot of freedom, but at the same time monitoring and supporting, seems to combine the best of two worlds.

7.5 Summary

Process mining [28] can be applied to the constraint-based approach. The DECLARE prototype, which can be used for execution and verification of constraint

models, stores information about executed instances in ProM readable MXML files. This is the first necessary step towards the integration of process mining and the constrain-based approach. Moreover, several plug-ins that use techniques tailored towards the constraint-based approach are already available in the ProM framework [91]. While the DECLARE prototype can execute and verify constraint models, these plug-ins offer other useful capabilities, as Table 7.7 shows.

Table 7.7: Capabilities of DECLARE and the four ProM plug-ins

	DECLARE	ProM plug-ins			
		LTL Checker	SCIFF Checker	DecMiner	Log-Based Recommendations
enactment	✓				
verification	✓				
conformance		✓	✓		
discovery				✓	
support					✓

The LTL Checker plug-in offers verification of event logs against properties specified in LTL [25]. Moreover, this plug-in can be easily used to verify past executions recorded in event logs against ConDec templates and constraints because the DECLARE prototype offers automatic export of templates and constraints to LTL Checker files.

The SCIFF Checker plug-in uses the powerful SCIFF language [48, 49] for advanced verification of event logs. This plug-in allows for the use of time and data variables in a more sophisticated way than the LTL Checker. Moreover, the DecMiner plug-in is able to learn SCIFF formulas from event logs and to automatically generate a model containing these formulas as constraints [154–156]. Thanks to the mapping between ConDec and SCIFF [70], DecMiner automatically generates a ConDec model from the learned formulas.

Process mining can also be useful during the execution of instances. Analysis of past executions can serve as basis for generating recommendations for users that are currently executing process instances [258]. By using the Recommendations plug-in in ProM, DECLARE is able to overcome the flexibility vs. support tradeoff, i.e., DECLARE users can get support from the system without sacrificing the flexibility.

Process mining techniques provide a powerful complement for the constraint-based approach. Moreover, the flexible style of work supported by DECLARE can benefit from the true integration of the two approaches, as Figure 7.17 shows.

Accountability, which is addressed by monitoring of executed instances with the LTL Checker, SCIFF Checker and DecMiner, plays an important role in flexible processes. On the one hand, flexibility allows people to work in various ways, i.e., people working with systems like DECLARE are more likely to be able to work in their own preferable way. The traditional procedural approach, on

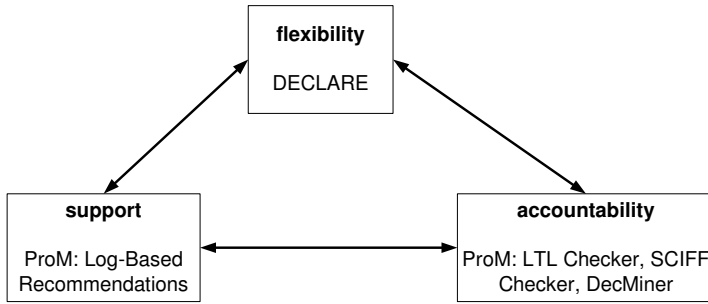


Figure 7.17: Integration of various approaches

the other hand, tends to force people to work in a pre-defined way and people have much less choice. Thus, monitoring the actual execution of processes can be considered as more important in the constraint-based than in the procedural approach. Consider, for example, the subway systems in Paris and Amsterdam. The Paris subway relates to the procedural approach, i.e., one must have a ticket in order to enter the subway. Therefore, frequent ticket controls are no longer necessary once the passenger is in the subway. Amsterdam's subway is more flexible, i.e., anyone can enter it. Due to this fact, random ticket controls are often conducted within the subway system in Amsterdam.

Flexibility can sometimes come at a cost. When multiple options are available, making the right decision might be difficult (e.g., inexperienced users, unusual situations, etc.). Therefore, providing an adequate support for users is crucial in flexible processes. The Log-Based Recommendations plug-in in ProM can offer support to users working with flexible systems like DECLARE. Moreover, the provided support does not sacrifice the intended flexibility at all. Moreover, the support provided by this plug-in is customizable and adjustable, e.g., based on the information retrieved during analysis of past executions (e.g., the LTL Checker, SCIFF Checker and DecMiner). Moreover, these four plug-ins enable a-posteriori analysis of the effects of the provided support. For example, if the effects are not satisfiable, the recommendation service (i.e., Log-Based Recommendations plug-in) can be adjusted.

Chapter 8

Conclusions

In this chapter we summarize our findings: in Section 8.1 we describe how we addressed our research goal, in Section 8.2 we summarize the contributions, in Section 8.3 we describe the limitations of our work, and in Section 8.4 we propose directions for future work.

8.1 Evaluation of the Research Goal

The goal of the research presented in this thesis is *to enable companies that use BPM systems to achieve an optimal balance between local and centralized decision making*. In order to achieve this (i.e., the goal in the research), we (1) proposed a comprehensive constraint-based approach towards process support and (2) developed the DECLARE prototype of a workflow management system that can offer an optimal ratio between flexibility and support (cf. Section 1.5). DECLARE can be downloaded from <http://declare.sf.net>. The problem with current systems for the automation of business processes is that they either focus on providing flexibility or support. The drawback of such systems is that users who work with flexible systems (e.g., groupware systems) do not get sufficient support from the systems, and users who work with systems that do provide support (e.g., workflow management systems) do not have enough flexibility in their work (cf. Chapter 1). In this thesis, we presented a constraint-based approach to workflow management systems that is able to combine flexibility and support. Moreover, besides the theoretical definition of the language in chapters 4 and 5, in Chapter 6 we also present the ‘proof of concept’ prototype DECLARE [2, 183]. We hope that enriching workflow management systems with flexibility will encourage organizations to combine workflow technology on the one hand and democratic work regimes with a high degree of localized decision making on the other. This way, organizations can benefit from the automated support that these systems offer.

8.2 Contributions

We believe that our approach is comprehensive in more aspects than just combining flexibility and support. First, in Section 6.11, we have showed that it is possible to combine multiple approaches into a federated workflow management system. This is a particularly interesting finding from a practical point of view because contemporary organizations implement multiple business processes, which can have different characteristics. Moreover, it is often the case that some parts of a business process require a high degree of support, while other parts of the same process must be flexible. Second, in Chapter 7, we showed that existing process mining techniques can support our constraint-based approach in the diagnosis phase of the BPM cycle (cf. Figure 1.1 on page 2), which proves that our approach can be applied to all phases of the cycle.

This section summarizes our contributions. In Section 8.2.1 we describe how our approach allows for different kinds of flexibility of workflow management systems. In Section 8.2.2 we summarize the different types of support that our approach provides. Section 8.2.3 discusses how our approach can help apply workflow technology in organizations that use democratic regimes of work with a high degree of localized decision making. In Section 8.2.4 we briefly describe the possibility to combine various approaches to business processes. The applicability of our approach to the full BPM life cycle is described in Section 8.2.5.

8.2.1 Flexibility of the Constraint-Based Approach

As pointed out in [226–228], there are several types of flexibility when it comes to workflow management systems: (1) flexibility by design, (2) flexibility by underspecification, (3) flexibility by change, and (4) flexibility by deviation. In Chapter 2 we presented many approaches and systems that aim at improving the flexibility of workflow technology, but none of these approaches covers all types of flexibility. In the context of flexibility, the most important contribution of our approach is the fact that it is able to support *all types of flexibility*. Actually, despite the fact that the primary motivation of our research is enabling a high degree of flexibility by design, it is remarkably easy to also provide for all other types of flexibility using our constraint-based approach (cf. sections 3.2.5 and 3.3).

Flexibility by Design

Flexibility by design is achieved in a workflow management system when its process models cover many execution alternatives [125, 226–228]. In general, constraint-based approaches are obvious candidates for offering a high degree of flexibility by design because execution alternatives are *implicitly* specified in constraint models, i.e., all execution alternatives that do not violate constraints are possible [226–228]. When it comes to our constraint-based approach, we allow for all execution alternatives that do not violate any mandatory constraint.

Moreover, with our approach it is possible to easily develop process models that offer a wide range of execution alternatives. For example, both models presented in figures 5.17 on page 145 and 5.18 on page 148 allow for infinite number of execution alternatives.

Flexibility by Underspecification

Flexibility by underspecification is achieved in a workflow management system if it is possible to design process models that have unspecified parts, which will be determined at run-time [125,226–228]. This type of flexibility is offered with dynamic decomposition of constraint-based models and YAWL models, as described in Section 6.11. Flexibility by underspecification in our approach can be achieved in a dynamic way, where the user at the latest moment during the execution of instances decides (1) whether to invoke a subprocess, (2) which subprocess to invoke, and (3) with which parameters.

Flexibility by Change

Flexibility by change is achieved in a workflow management system when it is possible to change models of running instances during their execution [125,226–228]. Our constraint-based approach and the DECLARE prototype offer flexibility by change by allowing for a comprehensive change of running instances: activities and constraints in instances can be easily added, removed, and changed. Considering the problems that traditional approaches face when it comes to run-time change (cf. Section 2.1.5), our constraint-based approach uses a remarkably simple method to handle this kind of change [184]. On the one hand, it is straightforward to check if a specific ad-hoc change is applicable to the state of the current instance by simply checking if the referring change permanently violates a mandatory constraint. On the other hand, in case of an invalid change, it is possible to produce a detailed diagnostics pinpointing the reason for the change failure (cf. sections 4.5, 5.7, and 6.5).

Flexibility by Deviation

Flexibility by deviation is achieved in a workflow management system if it is possible to deviate from a process model during execution, without having to change the model [226–228]. Our constraint-based approach allows for optional constraints in process models (besides mandatory constraints). While mandatory constraints must be fulfilled during execution, optional constraints can be violated (cf. sections 4.2 and 5.4). Moreover, the DECLARE prototype will provide an informative warning to users each time when they are about to violate an optional constraint, as shown in Figure 6.8 on page 171. In this manner, users can make decisions whether or not to violate the constraint and to deviate from the constraint-based model.

8.2.2 Support of the Constraint-Based Approach

Besides allowing for a high degree of flexibility, our constraint-based approach and the DECLARE prototype support various types of user assistance. In this section we briefly summarize the various types of support that can be provided by our approach and the DECLARE prototype: verification of models to detect errors, monitoring states of constraints and instances, enforcing the correct execution of instances, run-time recommendations based on past executions, and analysis of instances executed in the past.

Verification of Models

Constraint-based models can contain an arbitrary number of constraints that interfere in subtle ways. This can cause errors in models. Verification of constraint-based models provides an automated mechanism for detecting two types of errors, as described in sections 4.6 and 5.8. First, it is possible to automatically detect if a constraint-based model contains an event/activity that can never be executed, i.e., the so-called dead event/activity. Second, it is possible to detect that instances of a constraint-based model can never be executed correctly because it is not possible to satisfy all mandatory constraints from the model, e.g., there is a conflict in the model. In addition to automated verification of constraint-based models, it is also possible to detect the exact combination of constraints that causes (each of) the error(s). Reporting both the error and its direct cause (e.g., Figure 6.11 on page 174 and Figure 6.12 on page 175) helps developers of constraint-based models to understand the problem and eliminate errors.

Monitoring States of Instances and Constraints

Execution of instances of constraint-based models is driven by constraints. In order to execute an instance in a correct way, it is necessary that, at the end of the execution, all mandatory constraints are *satisfied*, i.e., that the instance is *satisfied*. Executing activities in an instance may change the state of one or more constraints, and the instance itself (cf. sections 4.4 and 5.6). Therefore, it is important that the instance state and states of all its constraints are presented to users throughout the execution of the instance. The insight into the state of the instance and its constraints helps users of the DECLARE prototype to understand what is going on and which actions are necessary in order to execute the instance in a correct way (cf. Figure 6.7 on page 169).

Enforcing Correct Execution of Instances

Some actions of users might cause an instance (and its constraints) leave the *satisfied* state. In some of such cases it is possible to take some actions that will eventually lead to a correct, i.e., *satisfied*, instance. We refer to this type

of violations as to *temporary violations*. In other cases, the instance becomes *permanently violated*, i.e., it becomes impossible to *satisfy* the instance in the future. Especially in instances with multiple constraints, it is very difficult for users to be aware of actions that will permanently violate the instance. Therefore, as described in sections 4.4, 5.6, and 6.4, the DECLARE prototype *prevents* users from taking actions that lead to permanent violation of instances.

Recommending Effective Executions

While executing constraint-based models, users typically have many alternatives available, i.e., the constraint-based approach offers a high degree of flexibility. In some situations, it might be difficult for users to decide themselves which alternative is the most appropriate one. Simultaneous run-time recommendations about which action leads to which result can help users in these situations. As described in Section 7.4, the DECLARE prototype provides run-time recommendations generated by the ProM tool [8, 91, 258] to its users. The architecture of the recommendation service in ProM allows for generation of various kinds of recommendations. In general, recommendations are generated based on past executions and are tailored towards a specific goal. For example, one recommendation strategy could be recommending actions that, in the past, led to quick instance completion. By presenting recommendations as additional information on the screen (cf. Figure 7.16 on page 216), DECLARE allows its users to choose whether to follow the recommendations or not.

Analysis of Past Executions

Workflow management systems support the execution of vast numbers of instances. Often, it is hard to keep track of all instances that were executed in the past. However, the information about past executions can be very useful in practice. Process mining techniques focus on various types of analysis of past executions, which can help to improve business processes (e.g., by redesigning process models) [28]. Although initially motivated by traditional approaches, existing process mining techniques can also be applied in the context of our constraint-based approach, as described in Chapter 7. Moreover, several mining techniques tailored towards the constraint-based paradigm are already available (e.g., the *LTL Checker*, *SCIFF Checker* and *DecMiner* presented in sections 7.2, 7.3.1 and 7.3.2, respectively). This is promising as processes that require a lot of flexibility typically benefit most from the results of process mining.

8.2.3 The Constraint-Based Approach and Organization of Human Work

Due to the lack of flexibility, procedural workflow management systems are unable to support Democratic Work Regimes (DWRs), as shown by the evalua-

tion with respect to structural requirements of De Sitter [231] in Table 2.3 on page 43. Enhancing flexibility of workflow technology allows users to make decisions about how to work, i.e., flexibility enables local decision making in business processes (cf. Chapter 1). Due to a higher degree of flexibility, our constraint-based approach and the DECLARE prototype enable the implementation of organizational styles that advocate local decision making in business processes, e.g., Socio-Technical Systems (STS) (cf. Section 2.3). STS advocate organization of work into Self-Managed Work Teams (SMWTs), where a meaningful piece of a business process (i.e., a subprocess instead of a single activity) is allocated to a SMWT as their assignment. Within one assignment, decisions are made locally by the SMWT, i.e., the SMWT controls the execution of the assignment.

The DECLARE prototype supports the style of work advocated by STS in two ways. First, work can be structured into meaningful pieces via the possibility to create arbitrary decompositions of DECLARE constraint-based models, as described in Section 6.11. Second, a high degree of flexibility allows DECLARE users to choose between multiple execution alternatives within instances they are working on. These two factors enable our approach to fulfil the requirements for a socio-technical style of work specified by De Sitter [231], as shown in Table 8.1.

Table 8.1: Evaluation of DECLARE with respect to the STS structural requirements of De Sitter [231]

	Socio-Technical requirements	DECLARE
1	<i>functional deconcentration</i> (multiple parallel processes)	YES: multiple execution alternatives allow users to choose the most appropriate alternative for each work order.
2	<i>integration of performance and control</i>	YES: flexibility allows people who execute an instance to control how the instance is executed.
3	<i>performance integration A</i> (whole tasks)	YES: model decomposition allows structuring processes into meaningful pieces of work.
4	<i>performance integration B</i> (prepare + produce + support)	NOT APPLICABLE
5	<i>control integration A</i> (sensing + judging + selecting + acting)	YES: flexibility allows people to select the appropriate corrective alternative and execute it.
6	<i>control integration B</i> (quality + maintenance + logistics + personnel, etc.)	NOT APPLICABLE
7	<i>control integration C</i> (operational + tactical + strategic)	YES: because people control the operational aspect of their work, operational, tactical and strategic control can be integrated at the workplace level.

Functional deconcentration. In a functional deconcentration different groups of work orders have different executions [231]. Constraint-based models allow for many execution alternatives as long as the main rules (i.e., constraints) are followed. This allows DECLARE users to choose the most appropriate execution alternative for each (group of) work order(s), i.e., functional deconcentration can easily be achieved with DECLARE.

Integration of performance and control. The same people who perform the work should also be authorized and responsible for control of work [231]. Flexibility and decomposition of business processes in DECLARE allows for the integration of performance and control, i.e., users can control the piece of a business process that they are executing.

Integration into whole tasks. Instead of specialized, short-cycled tasks, tasks should form a meaningful unit of work allocated to a group of people for execution [231]. Decomposition of DECLARE models allows for structuring a large business process into subprocesses, which are allocated to a group for execution. Therefore, integration into whole tasks is possible in DECLARE.

Integration of preparation, production and support. Preparation, production and support functions must be integrated at the workplace level [231]. As discussed in Section 2.3, workflow management systems support the production function only, and therefore this parameter is not applicable. This also applies for DECLARE.

Integration of control functions: sensing, judging, selecting, and acting. The functions of a control cycle are: (1) sensing the process states, (2) judging about the need for a corrective action, (3) selecting the appropriate correction action, and (4) acting with the selected control action [231]. All control functions should be integrated at the workplace level [231]. Because DECLARE allows people to control their work, they can sense, judge, select, and finally act based on the selected control action. Hence, this requirement of De Sitter [231] is also supported.

Integration of the control of quality, maintenance, logistics, personnel, etc. Control of quality, maintenance, logistics, personnel, etc. should be conducted at the workplace level by people who are performing the work [231]. As discussed in Section 2.3, workflow management systems support the production function only, and therefore this parameter is not applicable. This also applies for DECLARE.

Integration of operational, tactical and strategic controls. Operational, tactical and strategic controls should be integrated at the workplace level [231]. Thanks to the flexibility provided by DECLARE, people executing the work control the operational aspect of their work. This makes it possible to integrate operational, tactical and strategic control at the workplace level.

If we compare Table 8.1 to the earlier analysis of contemporary workflow management systems (cf. Table 2.3 on page 43), then it becomes apparent that DECLARE provides much more support for the style of human work advocated by STS.

8.2.4 Combining the Constraint-Based Approach with Other Approaches

Different types of business processes must often be combined in practice. On the one hand, a company can run various types of business processes. On the other hand, a business process itself can consist of subprocesses with different characteristics. Therefore, workflow technology should be able to support various types of processes. The DECLARE prototype can support mixtures of various processes via the decomposition of constraint-based DECLARE and procedural YAWL processes (cf. Section 6.11). Moreover, the same principle can be applied to other systems in order to enable combining even more approaches (e.g. worklets [41, 44, 45]).

8.2.5 Business Process Management with the Constraint-Based Approach

The Business Process Management (BPM) life cycle consists of several phases: design, implementation, enactment and diagnosis of business processes. Workflow management systems and process mining tools are two types of software products that can, combined, support the whole BPM cycle. As a workflow management system, the DECLARE prototype supports the first three phases: design, implementation and enactment phases. Process mining tools, e.g., the ProM tool, aim at supporting the diagnosis phase by allowing for automated analysis of executed business processes. Although not tailored for our constraint-based approach, existing process mining techniques can be used to diagnose executions of constraint-based processes. Moreover, some of the existing techniques (e.g., the LTL Checker, the SCIFF Checker, and the DecMiner) are fully applicable to the constraint-based approach. This shows that the constraint-based approach can be applied to the whole BPM cycle, as shown in Figure 8.1.

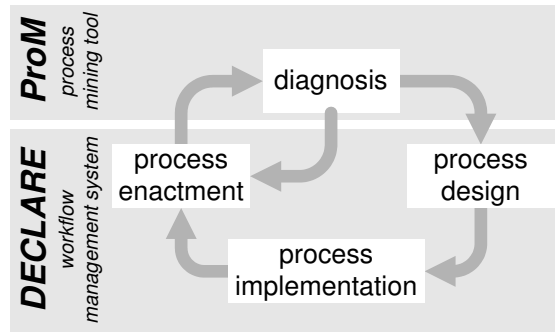


Figure 8.1: Constraint-based approach in the BPM life cycle

8.3 Limitations

Besides the problem of a missing activity life-cycle in the proposed ConDec language, which is described in detail in Section 5.9, the research presented in this thesis has two further limitations. First, in Section 8.3.1, we describe problems related to the complexity of models with many constraints. Second, in Section 8.3.2 we describe problems related to the absence of a proper empirical evaluation of our approach.

8.3.1 Complexity of Constraint-Based Models

There are two problems related to the complexity of models with a large number of constraints. The first problem is of the technical nature: due to the use of LTL for constraint specification, efficiency dramatically decreases when the number and complexity of mandatory constraints rises. The second problem is related to the human capacity to deal with information.

Efficiency-Related Problems

As described in Section 5.4, an automaton is generated for a conjunction of formulas of all mandatory constraints in an LTL-based constraint model, i.e., for the so-called mandatory formula (cf. Definition 5.4.1 on page 142). The mandatory formula is used for (1) execution, (2) ad-hoc change and (3) verification of constraint models based on LTL. Since the automata generated for an LTL formula is exponential in the size of the formula [74, 84, 85, 108, 111, 112, 158], the time needed for generating these automata becomes very long for big mandatory formulas. This can cause problems. For example, generating the automaton for an instance with a big mandatory formula may be extremely slow.

There are two possible causes of this problem. First, the more mandatory constraints there are in a model, the larger the mandatory formula for the model

will be. Second, as shown in Section 5.2, various constraint templates have different LTL formulas. For example, the LTL formula for the *alternate precedence* template presented in Table 5.2 on page 129 is significantly more demanding from a computational point of view than the formula for the *existence* template presented in Table 5.1 on page 127.

Consider the *Fractures Treatment* model presented in Figure 6.5 on page 167 as an illustrative example. Loading a new instance of this model in DECLARE takes approximately *12 seconds* on a computer with a Pentium 4 processor [4] of 2.80 GHz and 0.99GB of RAM using Microsoft Windows XP Professional version 2002 [5]. If the *alternate precedence* constraint between activities *check X ray risk* and *perform X ray* is removed from the original Fractures Treatment model, then starting an instance in DECLARE on the same computer takes approximately *1 second*. Obviously, the size of the LTL formula for the *alternate precedence* constraint dramatically increases the time to construct the automaton for the mandatory formula of the instance, i.e., it may take too long to create and start a new instance of this model in DECLARE.

The efficiency problem described above can occur at several points in DECLARE. First, when creating a new instance, an automaton is generated for the mandatory formula, which can cause the instance creation to take a long time. Second, when applying an ad-hoc change an automaton is generated for the mandatory formula of the changed instance, which can cause the application of the ad-hoc change to last too long. Third, whenever a condition on a mandatory constraint changes (i.e., after completion of an activity in the instance), a new automaton is generated for the mandatory formula, which can cause the processing of a completed activity to take too long (cf. Section 6.9). Fourth, during verification, i.e., analyzing dead activities, conflicts and history violations during ad-hoc change (cf. sections 5.8 and 5.7), an automaton is generated for combinations of mandatory constraints in order to identify the cause of error, which can cause the verification to be too time-consuming.

Note that, in case of models that contain dead activities and conflicts, the ‘plain’ detection of errors might take less time than finding the set of constraints that causes the error. This is because the original model has a smaller set of satisfying traces than its ‘submodels’, i.e., models where some of the mandatory constraints are removed (cf. Property 4.2.5). Therefore, the automaton generated for the whole model contains less traces than the automata created for the subsets of mandatory constraints. As a consequence, the generation of the automaton for the whole model might be considerably faster than the generation of the automata for subsets of mandatory constraints.

Capacity of Humans to Deal With Information

Constraints represent rules that should be followed while executing instances of constraint-based models. Therefore, it is important that people who are exe-

cuting an instance are aware of states of constraints in the instance throughout the whole execution. In this manner, people can understand what can or cannot be done in an instance and why, and what should be done in order to satisfy constraints and execute the instance correctly. Because of this, the DECLARE prototype presents whole instances and states of all constraints (by means of different colors) to its users (cf. Section 6.4).

Because people must keep track of states of all constraints while executing instances, instances with many constraints can easily become too complex for people to cope with. The fact that the different types of constraints have different semantics and the fact that constraints can interact in various ways makes handling instances with many constraints even more complex. On the one hand, an instance with many constraints contains a high degree of variance and the amount of information that people must handle. On the other hand, there are severe limitations on the amount of information people are able to receive, process and remember [172]. For example, research shows that it is difficult for people to deal with (approximately) more than seven chunks of varying information at a time [122,172]. Therefore, models with many constraints can easily become too complex for humans to cope with.

Process Decomposition as a Solution

At the moment, the only possible solution that can solve the two problems described above seems to be using the constraint-based approach to model business processes with a moderate number of constraints. To achieve this, we recommend decomposing large processes into small units of work, preferably modeled using our constraint-based approach. This will increase the efficiency on the one hand, and make it easier for people to execute their work on the other hand.

8.3.2 Evaluation of the Approach

Another limitation of the work presented in this thesis is the lack of a proper empirical evaluation of the proposed constraint-based approach and the DECLARE prototype. An empirical test is missing due to the time limitations. However, we did consider the available options for an evaluation. We see three possibilities for an evaluation: (1) evaluation by experts, (2) conducting laboratory experiments, and (3) testing in practice.

Evaluation by Experts

Evaluation by experts is not an uncommon practice in the area of workflow technology [247,248]. This type of tests can be performed by, e.g., means of conducting a survey or conducting interviews with experts in the field. These experts can come from academia and industry and the evaluation is based on their knowledge and professional opinion about the evaluated approach. The

advantage of this kind of evaluation is that experts already have the knowledge about the state-of-the-art in the field, and can easily judge the advantages and disadvantages of our approach. A drawback of this type of evaluation is that the evaluation is conducted by experts, who may have different opinions than the end-users.

Laboratory Experiments

Our approach can also be evaluated with experiments conducted in laboratory conditions. For example, a group of participants (e.g., students) can act as end-users and work with the DECLARE prototype on some imaginary scenarios. The advantage of experiments is that the approach is evaluated by people who acted as end-users and used the DECLARE tool for a while. The main drawback of experiments as an evaluation method is related to the fact that flexibility ‘comes into play’ when unexpected (exceptional) scenarios occur. On the one hand, waiting for ‘spontaneous’ exceptions usually takes a lot of time, and can make experiments too costly. On the other hand, prescribing exceptional situations and causing them on purpose during experiments is not a good solution, because prescribed situations cannot be considered unpredicted and exceptional.

Testing in Practice

The most desirable manner of evaluation seems to be testing the approach in practice. This would solve the problems imposed with the previous two methods: (1) end-users are the ones who evaluate the approach, and (2) practice can offer realistic unpredictable situations where flexibility and support are both needed. However, practice tests are also the most difficult type of evaluation to conduct. There are several reasons for this. First, DECLARE is only a prototype of a workflow management system and would need a lot of further product development in order to be suitable for use in practice. Second, it is not likely that an organization would be willing to abandon using a commercial workflow system and commit execution of its business processes to the testing of a prototype. Third, in order to evaluate our approach, it is necessary that end-users also have knowledge about (many) other approaches, which is not likely in practice.

8.4 Directions for Future Work

The work presented in this thesis represents an initiative to increase flexibility of workflow management systems, without sacrificing the support. We hope to have shown that a constraint-based approach is applicable to workflow management systems, and that workflow technology can benefit from the proposed approach. However, the current approach and the DECLARE tool can be improved in various ways, as indicated below.

In Chapter 5 we propose ConDec as a constraint-based language. ConDec focuses only on relations between activities in constraint models (cf. Section 5.2). However, business processes can often also depend on rules that include other perspectives of processes, like, e.g., data, resource and time perspective. In the future, it would be interesting to extend the ConDec language with other perspectives. For example, the time perspective would enable the use of deadlines (e.g., the *response* template can specify that ‘activity A must be followed by activity B within 5 days’). This can be achieved by, e.g., using the so-called Extended Timed Temporal Logic and timed automata [63] or LogLogics [123].

In Chapter 5 we propose LTL for specifying constraint templates and constraints. As described in Section 5.9, using LTL for this purpose introduces the problem of a rather artificial activity life cycle. Moreover, as discussed in Section 8.3, specifying constraints with LTL seriously decreases efficiency of the approach. Therefore, it would be interesting to also use *other languages* than LTL as a basis in order to identify the most suitable language. Moreover, using other languages for the constraint specification may reveal other possibilities with respect to possible constraint templates (i.e., maybe different templates can be specified using some other language).

The proposed approach should also be *evaluated* in a proper way. An appropriate evaluation can reveal both strong and weak points of the approach and indicate directions for further improvement.

As described in Chapter 1, workflow management systems are not the only type of systems that are used for business process support in practice. Groupware systems also can perform this function. However, these two types of systems focus on different aspects of business processes: while workflow technology aims at automating the operational aspect of processes, groupware is mostly used to support human collaboration. As a workflow management system, the DECLARE prototype focuses on introducing flexibility with respect to the operational aspect of work. However, flexibility implies a deeper involvement of people in work and, therefore, a more intensive collaboration. Therefore, we also propose extending our approach with ‘groupware-like’ functionality in order to be able to support *both operational and collaborative aspects* of the flexible work style.

8.5 Summary

In this thesis we presented a new approach to workflow management systems that is able to achieve a better balance between flexibility and support. We hope that, by optimizing flexibility and support, workflow technology will provide sufficient support for handling complex situations and, at the same time, enable people to control their work. As a result, people who work with workflow management systems will be more satisfied and achieve better results while performing their daily work.

Appendix A

Work Distribution in Staffware, FileNet and FLOWer

In order to gain insight into how workflow management systems distribute work to people, we have modeled the work distribution mechanisms of three commercial workflow management systems: Staffware [238], FileNet [107] and FLOWer [180]. Staffware and FileNet are examples of two widely used traditional workflow management systems. FLOWer is based on the case-handling paradigm, which can be characterized as a more flexible approach [26,39]. Each of the models is built as a CPN model [138,139,152] and is an extension of the *basic model* presented in Section 3.1.2. In the remainder of this chapter we present each of the developed CPN models: Staffware in Section A.1, FileNet in Section A.2, and FLOWer in Section A.3. Finally, Section A.4 concludes the chapter.

A.1 Staffware

We extended the *basic model* to represent the work distribution of Staffware. The way of modeling the *organizational structure* and *resource allocation algorithm* are changed, while the concept of *work queues* and the possibility of the user to *forward and suspend* a work item are added to the model. In this section we first describe the organizational structure of Staffware. Second, we describe the work queues and the two level distribution that accompanies them. Third, we explain the resource allocation of Staffware and its allocation function. Finally, we show which features have to be added to the *basic model* to implement the suspension and forwarding of work.

Simple organizational structure can be created in Staffware using the notions of *groups* and *roles*. The notion of group is defined as in the *basic model*, i.e., one group can contain several users, and one user can be a member of several

groups. However, specific in Staffware is that a role can be defined for only one user. This feature does not require any changes in the model structure or color sets. However, it changes the way the initial value for the *user maps* should be defined – one role should be assigned to only one user.

A.1.1 Work Queues

Groups are used in Staffware to model a set of users that share common rights. If a whole group is authorized to execute a work item, then each member of the group is authorized to execute the work item. Staffware introduces a *work queue* for every group. The work queue contains work items that group members can execute and it is accessible to all members of the group. Single users can be considered to be groups that contain only one member. Thus, one work queue is also created for every user and this personal queue is only accessible by a single user. Each user has access to the personal work queue and to work queues of all the groups the user is a member of.

While the *basic model* offers a work item directly to *users*, Staffware offers items in two levels. First, a work item is offered to *work queues* (*colset* $WQ = \text{string}$) in the *work distribution* module (cf. Figure A.1). We refer to this kind of work items as to *queue work items* (*colset* $QWI = \text{product } WI * WQ$). Second, every queue work item is offered to all members of a group (i.e., work queue) in the *offering* sub-module (cf. Figure A.1). Only one member will execute the queue work item once. We refer to a queue work item that is offered to a member (of a work queue) as to *user work item* (*colset* $UWI = \text{product } User * QWI$).

Figure A.1 shows the first level of distribution in the *work distribution* module of Staffware. The transition *offers to work queues* removes a work item token from the place *new work items* and offers it to work queues by producing queue work item tokens in place *to offer to work queues*. To do this, it retrieves *activity maps*, *user maps* and *field maps* as input elements. It also produces a work item token in the place *offered work items*. The queue work item tokens in the place *offer to work queues* are produced by the allocation function *offer_qwi* in the arc inscription between the transition *offers to work queues* and the place *to offer to work queues*. This function takes a work item, *activity maps*, *user maps* and *field maps*¹ as parameters. The effects of this function are explained in Section A.1.2. The transition *offers to work queues* produces a queue work item token in the place *offered work items* to store the information about which work items are expected to be completed by work queues. A token in the place *to offer to work queues* sends a message to the *offering* sub-module that the queue work item should be further distributed to the work queue members. After the completion of a queue work item, the *offering* sub-module creates a queue work item in the

¹The fifth parameter is an empty list and is used as aid to perform calculations in the function. This parameter should always be left empty and does not influence the function results.

place *completed queue work items*. The transition *completes work item* considers a work item to be completed when all queue work items that originate from that work item are completed. This transition retrieves a work item from the place *offered work items* and waits until all queue work items that originate from (that were offered to work queues based on) the referring work item. For this reason, the allocation function *offer_qwi* is called on the arc inscription between the place *completes work item* and the transition *completes work item* with the same parameters like in the arc inscription between the transition *offers to work queues* and the place *to offer to work queues*.

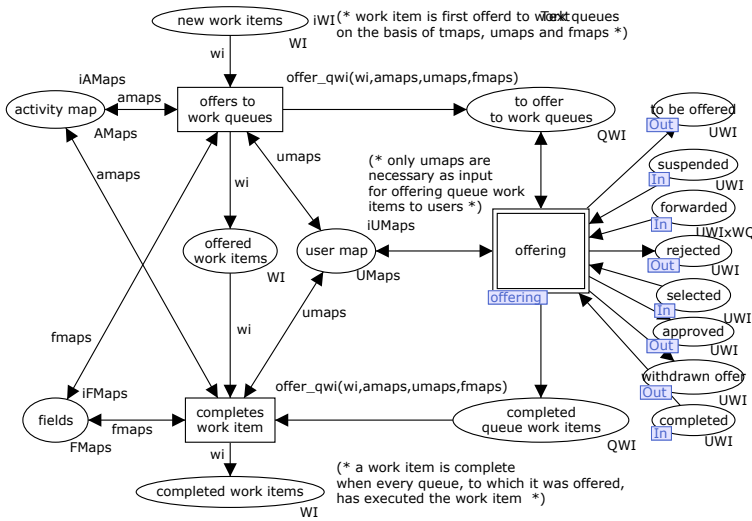


Figure A.1: Staffware - work distribution

Distribution to work queues in Staffware follows a similar logic like the distribution in the *basic model*, but also introduces some changes. A difference between these two distribution models is that, instead of distributing work directly to the *work lists* module (users) like in the *basic model*, the Staffware *work distribution* module hands-off the distribution to users to its sub-module *offering*. While a work item is the object of distribution in the *basic model*, the Staffware *work distribution* module distributes queue work items.

Figure A.2 shows the second level of distribution in Staffware within the *offering* sub-module. The first transition to fire here is the transition *offers to work queues*, when the message about the new queue work item is received from the *work distribution* module as a new queue work item token at the place *to offer to work queues*. This transition (1) removes the queue work item from the place *to offer to work queues* and produces it in the place *offered work queues*, and (2) retrieves *user maps* and creates new user work items in the place *to be offered*. The offers for users are created by the allocation function *offer_uwi*, which takes

assign a list of users, roles, groups and fields to each activity ($TMap = product\ Task * Users * Roles * Groups * Fields$). Figure A.3 shows how an *activity map* is specified in Staffware. Based on *activity maps*, function *offer_qwi* (cf. Figure A.1) allocates work queues that are authorized to execute the work item: (1) when a user name is provided in a *activity map*, the work item is offered to personal work queue of the referring user; (2) for every role in the *activity map*, this function offers the work item to the personal work queue of the user with that role (note that one role can be assigned to only one user); (3) a work item is offered to the work queue of every group that is stated in the *activity maps*; and (4) for authorizations via fields, allocation is executed at the run-time following the three above described allocation strategies. The allocation at run-time is referred to as a dynamic work allocation. Every field has a unique name (*colset Field = string*), e.g., *'next user'*. During the execution of the process, every field is assigned a value, and this value changes (e.g., users can assign values to fields). Staffware assumes that the value of the assigned data field can be a group name, a role name or a user name. If the field *'next user'* (which for example has the value of *Joe Smith* assigned) is specified in the *activity map* of an activity, then the actual value of the field is assigned to the *activity map* entry at the moment when the activity becomes enabled. Thus, *Joe Smith* will be used in the allocation.

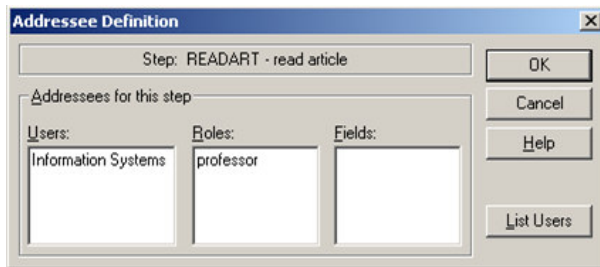


Figure A.3: Staffware - an activity map

Figure A.4 shows Staffware Process Client tool, where users can access their work queues and process the work items. In this case, there are two work queues: (1) the work queue for the group "Information Systems", and (2) the personal work queue of the user *Joe*.

When all the properties of the Staffware work distribution are merged together, unexpected scenarios might happen. In the example shown in Table 3.3 on page 59 activity *read article* should be allocated to users which are from the group *Information Systems* and have the role *professor*. The *basic model* allocates this activity to users that are from the group *Information Systems* and have the role *professor*, i.e., to the user *Joe*. Unlike the *basic model*, Staffware allocates this activity to: (1) the work queue of the group *Information Systems* (which members are *Mary* and *Joe*), and (2) the personal queue of the user who

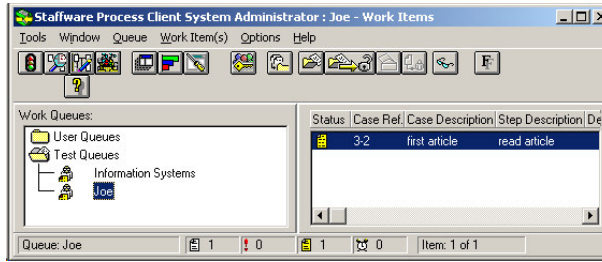


Figure A.4: Staffware - a work queue with a work item

has the role *professor* (with one member *Joe*). A work item is completed in Staffware when all its queue work items are completed (cf. Figure A.1). Thus, the activity *read article* will be executed two times: (1) once by a member of the group *Information Systems* – *Mary* or *Joe*, and (2) once by the user who has the role *professor* – *Joe*. As the result of Staffware work distribution, the work item *read article* has two possible scenarios of the execution. This activity will be executed either once by *Mary* and once by *Joe*, or two times by *Joe*. Which one of these two scenarios will take place, depends only on which user is faster, i.e., on which users select the activity before the others do.

A.1.3 Forward and Suspend

When the user selects a work item in the *basic model*, the work item is *assigned* to him/her, and (s)he can *start* the work item and *execute* it. Figure A.5 shows that Staffware offers a more realistic and somewhat more complex model of the life cycle of a work item than the *basic model* (cf. Figure 3.9 on page 57). After a user *selects* the work item, it is *assigned* to him/her, and then the user can either *start* the work item or *forward* it to another user. Forwarding transfers the work item to the state *offered*, because it is automatically *offered* to the new user. If the user chooses to *start* the work item, (s)he can *execute* it or *suspend* it. When a work item is suspended, it is transferred back to the state *initiated*. After this, the system *offers* the work item again to all authorized users.

Forwarding and suspending of work items adds two messages that are exchanged between *work distribution* and *work lists* modules in Staffware model. Figures A.1 and A.2 show two new places – *forward* and *suspend*. Users trigger these two new actions in the *start work* and *stop work* sub-modules of the *work list* module (cf. Figure 3.12(b) on page 61).

Figure A.6(a) shows that in the Staffware sub-module *start work* the user can choose to *select* or *forward* (to another work queue) the work item. To enable forwarding, we add the transition *forward* to the *start work* sub-module in Staffware model. The request to select a work item is represented with a user work item in the place *requested*. After this request, the *start work* sub-module

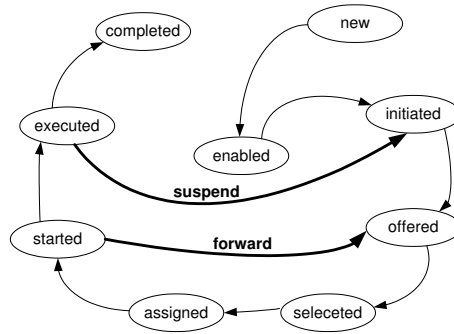


Figure A.5: Staffware - work item life cycle

waits until the *work distribution* module approves the request, by creating a user work item token in the place *approved*. When the request is approved the transitions *start work* and *forward* can fire depending on the user decision. Both transitions consume the two matching user work item tokens from the places *requested* and *approved*. The transition *start work* has the same effect as the *basic model* (cf. Figure 3.12(e) on page 61). The transition *forward* matches the user token in the place *logged on* with the referring user work item, retrieves a work queue token from the place *work queues* and produces a token in the place *forwarded*. The initial marking for the place *work queues* consists of all group names and all user names registered in the system. This is straightforward because Staffware creates group work queues for all groups and personal work queues for all users. The place *forwarded* is of the color set type that combines a user work item and a work queue to which the work item should be forwarded ($colset\ UWIxWQ = product\ UWI*WQ$). The transition *forward* produces a token in the place *forwarded* with the arc inscription $((u, qwi), wq)$. This token sends the message to the *work distribution* module that the referring user (u) wants to forward the referring queue work item (qwi) to the referring work queue (wq).

Figure A.6(b) shows that in the sub-module *stop work* the user can choose to *complete* or *suspend* the work item. The transition *suspend* is added to the sub-module. While a user is executing a queue work item, a referring user work item token is in the place *in progress*. At any time during the execution of a work item, one of the transitions *complete* and *suspend* can fire. While transition *complete* has the same effects as in the *basic model* (cf. Figure 3.12(f) on page 61), transition *suspend* is new in Staffware model. This transition matches the user token in the place *logged on* with the user work item in the place *in progress*. It consumes the user work item from the place *in progress* and produces the referring user work item token in the place *suspended*. A user work item token in the place *suspended* sends the message to the *work distribution* module that the referring user wishes to suspend the referring queue work item.

The *work distribution* module (cf. Figure A.1) handles forwarding and sus-

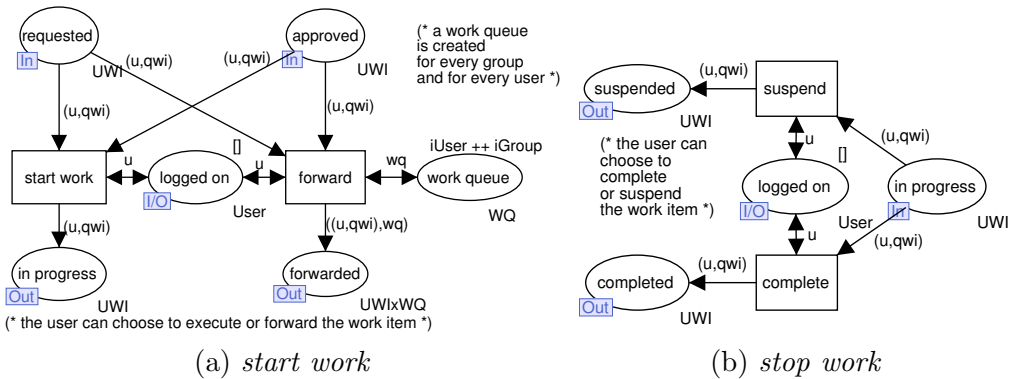


Figure A.6: Staffware - sub-modules *start work* and *stop work*

pending in a new sub-module: the *suspend* and *forward* sub-module shown in Figure A.7. On the one hand, in case of *forwarding* the work item is automatically *cancelled* for the current work queue and *offered* to the new work queue. On the other hand, in case of *suspending* the work item is *cancelled* for the current work queue and *re-offered* as a new work item. When a message that a user wishes to forward a queue work item to a work queue from the *work lists* module arrives, a token is produced in the place *forwarded*. The *forward* and *suspend* sub-module then automatically fires the transition *forward*. This transition consumes the token from the place *forwarded* and produces two different tokens in places *to re-offer* and *to cancel*. The queue work item token that is forwarded is produced in the place *to cancel*. A new queue work item, which consists of a referring work item and a new work queue, is produced in the place *to re-offer*. When the message that a user wishes to suspend a user work item a token is produced in the place *suspended*. The transition *suspend* fires automatically when the message arrives, consumes the user work item token from the place *suspended* and produces two identical referring queue work item tokens in the places *to cancel* and *to re-offer*. The transitions *re-offer* and *cancel* fire automatically when tokens are produced in places *to re-offer* and *to cancel*, respectively. Transition *cancel* consumes matching queue work item tokens from the places *to cancel* and *selected work items*. In this way the queue work item is removed from the model. The transition *re-offer* consumes a queue work item token from the place *to re-offer* and produces one in the place *to offer to work queues*. In this way, the *offering* sub-module can offer the queue work item to the members of the work queue again.

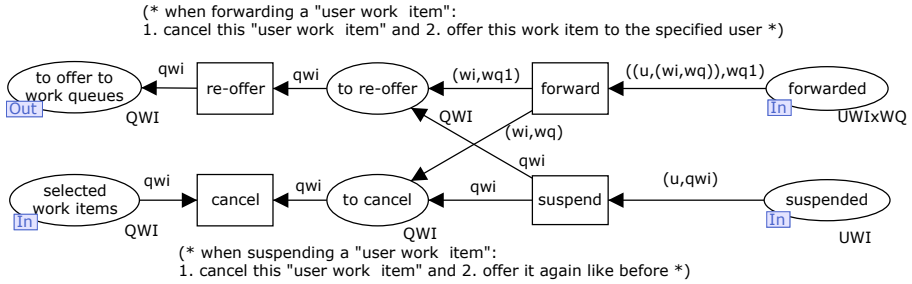


Figure A.7: Staffware - forward and suspend

A.2 FileNet

Like Staffware, FileNet is a widely used traditional process-oriented workflow management system. In this section we will describe the FileNet CPN model that we develop by extending the *basic model*.

Unlike the *basic model* and Staffware, FileNet does not allow for modeling *roles* of users. The organizational structure in FileNet can be modeled via two types of groups. First, administrators of the FileNet system can define *work queues* (*colset* $WQ = string$) and assign their members by selecting users of the FileNet system. Work queues are defined on the global level of the FileNet system – they are valid for every process (workflow) definition. Second, process modelers can define *workflow groups* (*colset* $WG = string$) in every process model. Thus, workflow groups belong to and are valid only in the process (workflow) model in which they are defined. Workflow groups represent teams in FileNet. While executing an activity of a process definition, users have the possibility to change the structure of workflow groups of the referring process.

A.2.1 Queues

Work queues and *personal queues* are two types of queues (*colset* $Q = string$) in FileNet. Queues are pools from which users can select and execute work items. A work queue can have a number of members while a personal queue has only one member. When a work item is offered to a queue, one of the queue members can select and execute the work item. FileNet distributes work in two levels using queues. First, the work item is offered to queues as a *queue work item* (*colset* $QWI = product WI * Q$). Second, the queue work item is offered to the members of the queue as a *user work item* (*colset* $UWI = product User * QWI$).

Figures A.8 and A.9 show that the model of the two-level work distribution in FileNet is similar to the Staffware model. For more detailed description of this kind of distribution we refer the reader to the Staffware description in Section A.1.

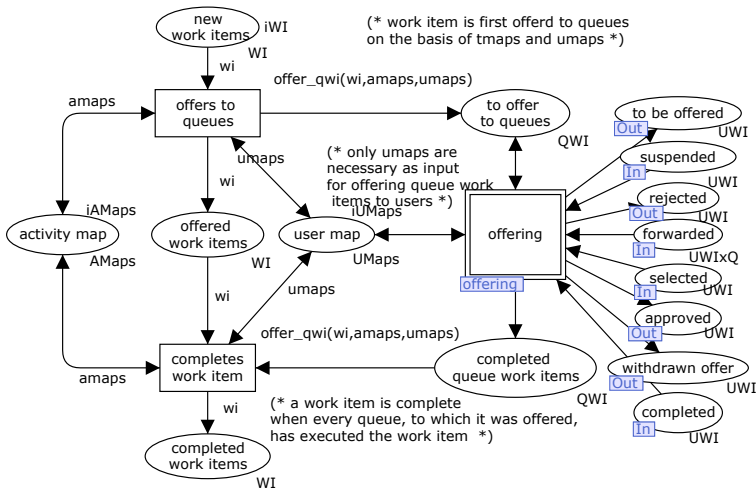


Figure A.8: FileNet - work distribution

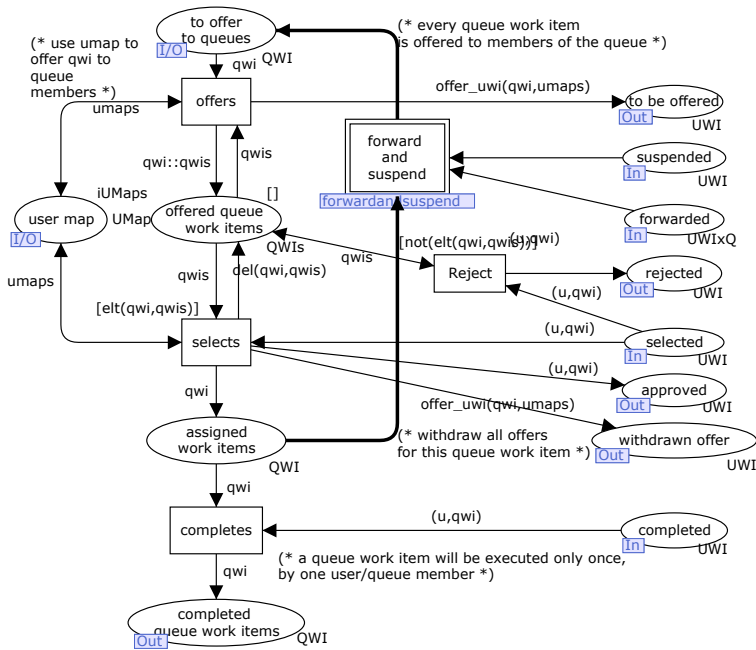


Figure A.9: FileNet - offering

A.2.2 Resource Allocation

FileNet allocates work using work queues and lists of participants. Figure A.10 shows that an activity in FileNet can be allocated to either a work queue *or* to a list of participants. In this figure we can see that the activity *read article* has

been allocated to the participants that belong to the workflow group *Information Systems*. Users and workflow groups can be entries of a list of participants. In the FileNet model, *activity maps* are defined as a combination of an activity, a list of work groups, and a work queue ($colset AMap = product Activity * WGs * WQ$, cf. Section 3.1.2). When defining the input value for a *activity map*, either a work queue or a list of workflow groups should be initiated, but not both.

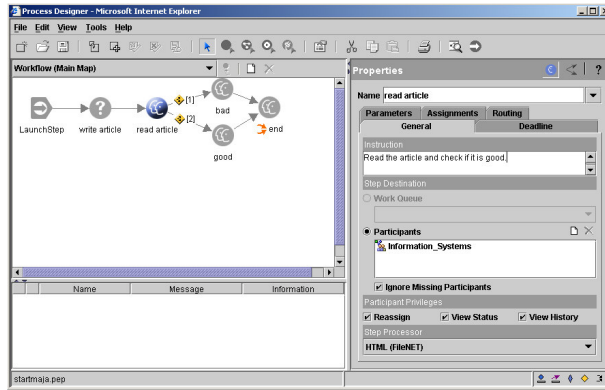


Figure A.10: FileNet - allocation for work queues or participants

If the activity is allocated to a work queue, FileNet offers the referring work item to the work queue. If the activity is allocated to a list of participants, then it is offered to personal queues of all users that are given as individual participants or are members in participating workflow groups. Allocation via participants is introduced to support team work in FileNet, via the so-called ‘process voting’. During the execution of an activity, all participants vote for the specified decision. The work distribution mechanism uses their decisions to determine which work items will be executed next. Since our models abstract from the process perspective, we did not model process voting in the FileNet model.

The allocation function *offer_qwi* allocates queues that are authorized to execute the referring activity. Figure A.8 shows this function in the inscription on the arc between the transition *offers to queues* and the place *to offer to queues*. This function takes four parameters: (1) the referring work item, (2) *activity maps*, (3) *user maps*, and (4) an empty list – used as an utility for calculations. This function first searches the *activity maps* for the map of the activity that is specified in the work item. The referring *activity map* will point to either a work queue or to a list of participants. In case of a work queue the function produces a queue work item token for the referring work queue. The situation is slightly more complex in the case of a list of participants, because this list may contain users and workflow groups as elements. For each user in the list of participants a queue work item token is produced for the personal queue of the user. For

each workflow group in the list of participants queue work items are produced for personal queues of all group members.

A.2.3 Forward and Suspend

Users can forward and suspend work items in FileNet. When the user selects a work item (s)he can start working on it or forward it to another user. In this case FileNet automatically offers the work item to the new user. When the user is executing a work item s(he) can complete or suspend the work item. In this case FileNet needs to apply the distribution mechanism again, and offer the work item to all allocated users. Figure A.11 shows the life cycle of a work item in FileNet. When the life cycle models of FileNet and Staffware (cf. Figure A.5) are compared, it can be seen that they are identical. Therefore, we use the same adjustments in FileNet like in Staffware models to implement forwarding and suspension: modules start work and *stop work* are changed and sub-module *suspend and forward* is added in the *work distribution* module. For detailed description of these sub-modules we refer the reader to Staffware description in Section A.1.3.

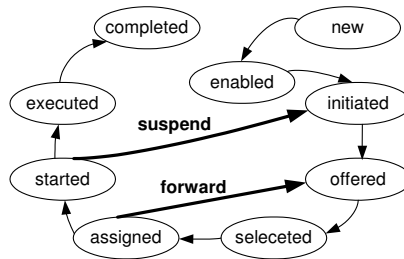


Figure A.11: FileNet - work item life cycle

A.3 FLOWer

FLOWer is a case-handling system. Case-handling systems differ in their perspective from traditional process-oriented workflow management systems because they focus on the instance, instead of the process [26,39]. FLOWer offers a whole instance to a user by offering all available work items from the instance. When working with FLOWer, the user does not have to follow the predefined order of activities in the process definition, i.e., this system offers *flexibility by deviation* [226–228].

To model FLOWer, we extend the *basic model* in such a way that (1) it handles *case-handling distribution* instead of the process-oriented one, (2) it enables the complex *authorization* and *distribution* specifications that FLOWer has, and (3) it enables users to *execute*, *open*, *skip* and *redo* work items.

A.3.1 Case Handling

To model a case-handling system like FLOWer, a number of color sets are introduced. Every process definition in FLOWer is referred to as an instance type (*colset InstanceType = string*). Thus, every instance type refers to a list of activities (*colset Activities = list Activity*), which form the process definition (*colset Process = product InstanceType * Activities*) for that instance type. One instance (*colset Instance = product InstanceID * InstanceType*) represents an instance of an instance type and is identified by an instance identification (*colset InstanceID = INT*).

FLOWer distributes work in two levels. First, an instance is distributed to users (*colset UInstance = product User * Instance*). Only one user can select and open the instance at one moment. Unlike the distribution in the *basic model*, where distributed work items refer to single activities, FLOWer distributes whole instances on its first level of distribution. Second, the selected instance is opened for the user. Work items (*colset WI = product Instance * Activity*) from the instance are offered to the user, based on the authorization and distribution rules. In the second level of FLOWer distribution users can execute, open, skip and redo work items from the selected instance, instead of only executing work items from multiple instances like in the *basic model*, Staffware and FileNet.

A.3.2 Authorization Rights

Authorization rights are defined for every instance type. First, process-specific roles are defined within an instance type (*colset PRole = product Role * InstanceType*). Second, to make authorization rights, roles are assigned to activities within the instance type. These authorization rights are stored in *activity maps* (*colset AMap = product Activity * Role * InstanceType*). The authorization rights determine what users *can do* and are applied by the distribution mechanism when opening the instance for the user. The user is allowed to work only on activities for which (s)he has the authorized roles. Although authorization exists in the *basic model*, Staffware and FileNet, in these models it is defined on the global (system) level, instead of embedding roles in process models. Rather, roles are defined in the global organizational model.

A.3.3 Distribution Rights

Distribution rights define what users *should do*. Unlike authorization rights, distribution rights are defined on the global level of the FLOWer system, and are valid for all instance types. These rights can be used to model the organizational structure and to assign authorization rights from the process definitions (instance types) to users. *Function profiles* and *work profiles* define distribution rights. Function profile has a unique function name (*colset FN = string*) and a list of

instance type authorization roles ($colset FP = product FN * PRoles$). If, for example, there are two instance types (two processes) – one with “secretary1” and the other with “secretary2” as an authorization role, the function profile “secretary” could include both authorization roles. When we would assign the function profile “secretary” to a user, we would indirectly assign both authorization roles from two processes. Work profiles assign function profile(s) to users and they can be used to structure organization into groups, departments or units. One work profile consists of a unique name ($colset WN = string$), a list of users and a list of function profiles ($colset WP = product WN * Users * FNs$). Distribution rights are used to define the organizational model in FLOWer. While this model is independent of the authorizations in the *basic model*, Staffware and FileNet, in FLOWer it has to be related to special authorization roles from instance types. In this way, FLOWer creates two-layered organization specification: one part of it is in the distribution rights and the other in the authorization rights.

A.3.4 Distribution of Instances

Figure A.12 shows the *work distribution* module of FLOWer. In this module, FLOWer model distributes new instances to users, instead of distribution work items like the *basic model*. When a new instance token is available in the place *new instances*, the transition *offers instance* fires. This transition consumes the instance token from the place *new instance* and retrieves *activity maps*, work profiles and function profiles from places *activity map*, *work profile* and *function profile*, respectively. It adds a token to the list of instances in the place *offered instances*. This place stores a list of instances that were offered to users but not yet selected by any user. The most important effect of this transition is that it produces a user instance token in the place *offer instance*, via the instance allocation function *offerinstance* in the arc inscription. A user instance token in the place *offer instances* sends a message to the *work lists* module to offer the referring instance to the referring user. The allocation function *offerinstance* takes four parameters: (1) instance that will be allocated, (2) *activity maps* to find the mapping of the referring instance type activity to the instance type role, (3) a list of function profiles to find the ones that contain the instance-type-specific role from the *activity maps*, and (4) a list of work profiles to find the users that are assigned to the appropriate function profiles.

Next, the *work distribution* module waits for the message from the *work lists* module that a user wants to select an instance. This message arrives with a user instance token in the place *selected instance*. Only one user can select an instance at the same time in FLOWer. The *work distribution* module responds to this message accordingly to this rule by checking if the instance has already been selected, i.e., if the referring instance is contained in the list of offered instances in the place *offered instances*. Transitions *selects instance* and *reject instance* are the two alternative transitions that can respond to the new user instance

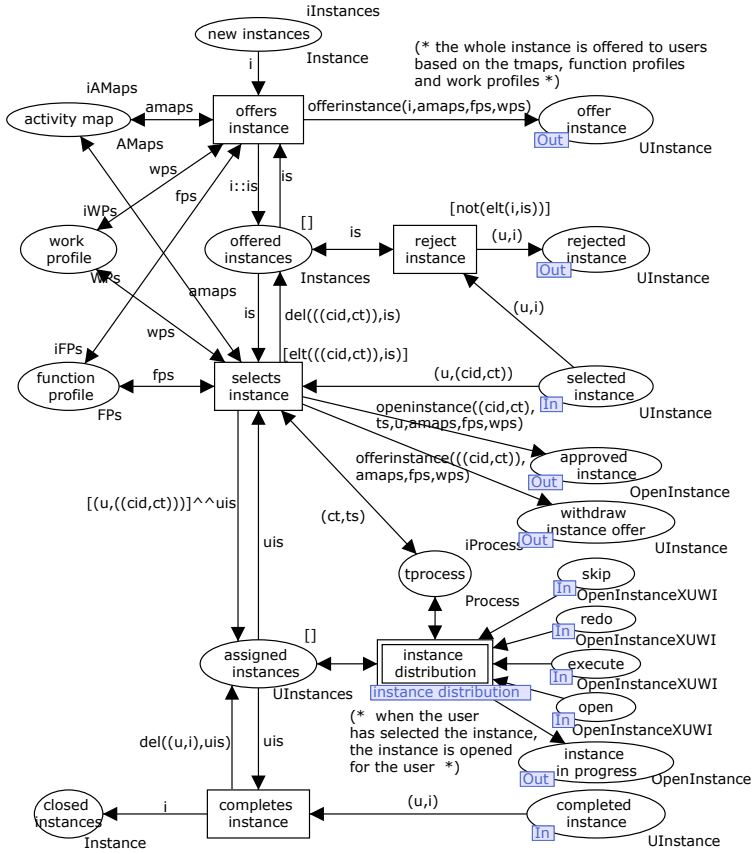


Figure A.12: FLOWer - work distribution

token in the place *selected instance*. The transition *selects instance* will fire if the referring instance is contained in the list of instances in the place *offered instances*, which can be seen in the guard of the transition. This transition will consume the user instance token from the place *selected instance*, remove the referring instance from the list of instances in the place *offered instances* and produce the user instance token in the place *assigned instances*. By removing the token from the place *offered instance*, we assure that the instance cannot be selected again. The transition *selects instance* also sends two messages to the *work lists* module. First, since the instance has just been selected, a message is sent to the *work lists* module to withdraw all offers of the referring instance. The transition *selects instance* sends this message by producing all previous offers of the referring instance in the place *withdraw instance offer*. Second, the approval message for the selection of the instance (for the user) is sent by producing the referring open instance token in the place *approved instance*.

The function *openinstance* in the arc inscription between the transition *se-*

lects *instance* and the place *approved instance* produces an open instance token. This function takes six parameters: (1) the identification and the type of the instance to be open, (2) the activities that are contained in the process definition of the instance type, (3) the user for whom the instance is open, (4) a list of *activity maps* to find instance type authorized roles for every activity, (5) a list of function profiles to search for the ones that contain the authorization roles for the activities, and (6) a list of work profiles to determine which of the selected function profiles are assigned to the user. The open instance token (*colset OpenInstance = product UInstance*InstanceState*) that is produced stores the information about the user, instance and the state of the instance (*colset InstanceState = product WIs*WIs*WIs*WIs*). The instance state consists of four lists of work items that are: (1) *waiting* to be enabled, (2) *active* (i.e. they are enabled and can be executed), (3) *finished* (executed), and (4) *skipped*. When the instance is opened for the first time, the list of active items contains the first work item in the instance, the list of waiting items all the other authorized work items, and the lists of executed and skipped items are empty.

After the *work distribution* module opens the instance for the user, the *instance distribution* sub-module handles the distribution within the instance. This sub-module manages events when users work on activities within the instance. We refer to this part of the FLOWer work distribution as to the distribution within the instance and describe it Section A.3.5.

The last message that arrives from the *work lists* module is that the user has finished working with the instance. This message arrives with a new user instance token in the place *completed instance*. The transition *completes instance* consumes this token, removes the referring user instance token from the list of assigned instances in the place *assigned instances* and produces the referring instance token in the place *closed*. Although, in FLOWer system, after the instance has been closed it is possible to be offered again, we do not model this in the FLOWer CPN model due to the complexity and size of the model. However, it is possible to include this behavior in the model by: (1) returning the closed instance token to the place new instance, and (2) storing permanently the state of every instance, similarly like *activity maps*, function profiles, work profiles and process definitions.

Figure A.13 shows the *work lists* module of the FLOWer model. Generally, the functionality of the part of this module that deals with the distribution of instances is the same as the *work lists* module of the *basic model* shown in Figure 3.12(b) on page 61. However, there are some differences between these two modules. First, the places are named differently to match the context. There are two kinds of places in the FLOWer model: (1) names of the places and transitions that deal with the instance distribution contain word ‘instance’ (e.g., the place *offer instance*), and (2) names of the places that deal with the distribution within the instance do not contain the word “instance” (e.g., place *execute*). Second, the places that deal with the distribution are of the user instance type, instead

of the user work item type. Finally, the sub-module *action* deals with actions of users in the context of the distribution within an instance. The *action* sub-module is described in Section A.3.5. Because the distribution of the instances in the FLOWer *work lists* module is similar to the distribution of work items in the *work lists* module of the *basic model*, for a detailed description we refer the reader to Section 3.1.2.

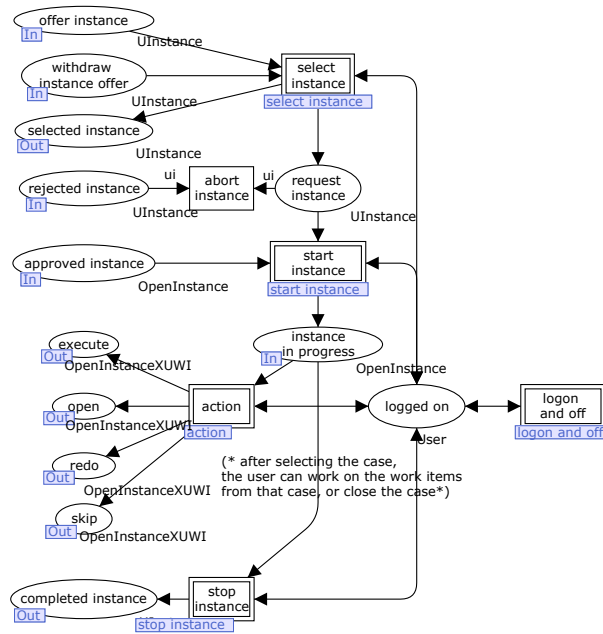


Figure A.13: FLOWer - work lists

A.3.5 Distribution within an Instance

When working with traditional, process-oriented, systems users can mostly execute or cancel work items. This property of such systems can be found in the *basic model*, Staffware model and FileNet model. Unlike these process-oriented systems (models), a case-handling system FLOWer allows users to perform *four actions* on work items: open, execute, skip and redo. Figure A.14 shows that the life cycle of a work item in FLOWer is somewhat more complex than the life cycles of the other models. Because a user selects a whole instance, work items are *assigned* to the user before they are *enabled*. Following the process definition of an instance type (because of the complexity we assume this to be a sequence of activities) the FLOWer systems *enables* the next work item in the sequence. After the user *selects* an enabled work item, (s)he starts with its execution and the work item is transferred to the state *execute*. Once the execution stops, the work item becomes *completed*. It is possible to *skip* an *enabled* work item and

Table A.1: Four actions in FLOWer

preconditions	action	postcondition	
work item was		work item becomes	side effects
<i>waiting</i>	open	<i>active</i>	<i>Waiting</i> and <i>active</i> items that succeed become <i>skipped</i> .
<i>active</i>	execute	<i>finished</i>	The direct successor becomes <i>active</i> .
<i>active</i> or <i>waiting</i>	skip	<i>skipped</i>	Succeeding <i>waiting</i> and <i>active</i> items become <i>skipped</i> . The direct predecessor becomes <i>active</i> .
<i>finished</i> or <i>skipped</i>	redo	<i>active</i>	Preceding <i>finished</i> and <i>skipped</i> items become <i>waiting</i> .

stance token is produced in place *instance in progress*, the distribution within the instance starts and the user can work on the work items in that instance.

Figure A.15 shows the *action* sub-module, which is a new sub-module in the FLOWer *work lists* module (cf. Figure A.13). This sub-module handles the actions of a user when s(he) works within an instance and makes sure that the *preconditions* (cf. Table A.1) are met before each of the four actions can take place. The *action* sub-module can be seen as an extension of the *start work* sub-module of the *basic model*, which is shown in Figure 3.12(e) on page 61. In the *start work* sub-module the user can only start the work item in progress. However, when an open instance is in progress in the *action* sub-module the user can: (1) *execute* the work item which is next in the process definition of the instance type – an item contained in the list of active items in the instance state; (2) *open* for executing a work item that is still not ready for execution according to the process definition of the instance type – an item contained in the list of waiting items in the instance state; (3) *skip* a work item that is currently enabled or waiting to be enabled – an item contained in the lists of active or waiting items in the instance state, or (4) *redo* a work item and execute again a work item which has already been executed – an item contained in the lists of finished or skipped items in the instance state.

Four transitions in the *action* sub-module refer to the four actions of users – open, execute, skip, and redo. All transitions retrieve an user token from the place *logged on*, to make sure that only the users who are currently logged on can perform these actions. Also, all transitions consume the open instance token from the place *instance in progress*. The open instance token stores the information about the user, instance, and instance state (i.e., lists of waiting, active, finished and skipped work items for that instance). It is necessary to consume(remove) the open instance token from the place *instance in progress* because after every action, the *work distribution* module (more specifically – its *instance distribution*

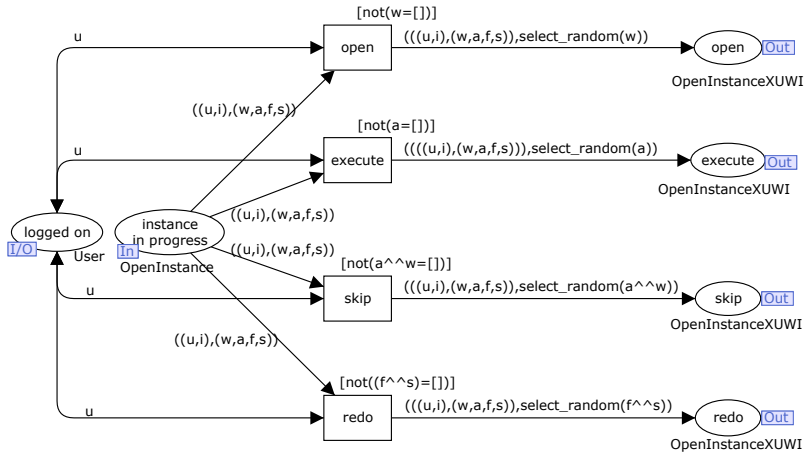


Figure A.15: FLOWer - action

sub-module) changes the state of the instance, which is stored in the open instance token in the place *instance in progress*. After performing one action, the user cannot perform the next action before the *instance distribution* sub-module updates the instance state and produces the referring open instance token in the place *instance in progress*. The transition *open* can fire only if the list of waiting items in the instance state is not empty, as can be seen in the guard of this transition. The transition *open* produces a token in the place *open*. This place is of a complex type, which consists of an open instance and a user work item. When a token is produced in this place, the message is sent to the *instance distribution* sub-module that the referring user work item should be open in the referring open instance. Although a user who works with FLOWer can freely choose which item should be open, for the simplicity we use an random function to select an item from the list of waiting items. The inscription on the arc from the transition *open* produces a token from the current open instance and the (randomly) selected waiting item in the place *open*. Similarly, according to the preconditions (cf. Table A.1), guards on transitions *execute*, *skip*, and *redo* ensure that they fire only when lists of *active*, *active and waiting*, and *finished and skipped* items are not empty, respectively. Places *execute*, *skip* and *redo* are of the same type as the place *open*, and a token in each of those places sends a message to the *instance distribution* module that the referring user work item should be executed, skipped or redone, respectively. Following the preconditions (cf. Table A.1), the inscriptions on the arcs between the (1) transition and place *execute*, (2) transition and place *skip* and (3) transition and place *redo* each create a token containing the open instance and the randomly selected (1) active, (2) active or waiting, and (3) finished or skipped work item in the places (1) *execute*, (2) *skip* and (3) *redo*, respectively.

When working on an instance in the FLOWer system, users work with the interface tool Wave Front [180] where they can see the state of the open instance. Users can see which work items are waiting, active, finished and skipped. Figure A.16 shows one example of an open instance in the Wave Front. The first two activities (*Claim Start* and *Register Claim*) are *finished* work items and they are marked with a ‘check’ symbol. The third work item (*Get Medical Report*) was *skipped*, as can be seen from the ‘arrow’ symbol. Thus, finished and skipped work items are presented after the Wave Front line. The three *active* work items on the Wave Front line are *Get Police Report*, *Assign Loss Adjuster* and *Witness Statements*. Finally, the two last work items (*Policy Holder Liable* and *Close Instance*) are *waiting* before the Wave Front line to become active.

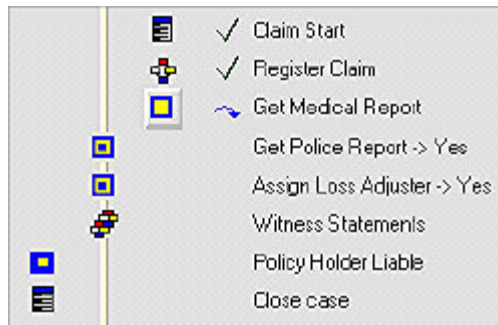


Figure A.16: FLOWer wave front

Instance distribution is a sub-module of the FLOWer *work distribution* module (cf. Figure A.12). This sub-module responds to user’s requests to open, execute, skip or redo work items in the distribution within the instance. The task of the *instance distribution* sub-module is to respond to the actions of users by changing the state of the instance accordingly to the postcondition of every action (cf. Table A.1). Figure A.17 shows the *instance distribution* sub-module of the FLOWer model. The requests (messages) for actions are received via tokens in places *open*, *execute*, *skip* and *redo*. These places are of the type which stores the information about the open instance (the user instance and the instance state) and the user work item to which the action (open, execute, skip or redo) should be applied. Due to delays, it is possible that a message to execute a work item from the instance arrives after the instance had been closed, the transition *ignore* behaves as a ‘garbage collector’ of such requests. This transition retrieves user instance token from the place *assigned instances* and consumes a token from the places *open*, *execute*, *skip* and *redo*. Thus, when the transition *ignore* fires, the message to perform an action is ignored and removed from the model. The guard on this transition makes sure that the transition will fire only if the instance is not closed, i.e., the appropriate user instance token is not found in the list in place *assigned instances*. Transitions *open*, *execute*, *skip* and *redo*

fire when tokens arrive to places *open*, *execute*, *skip* and *redo*, respectively. Each of these transitions consumes the arrived token from the appropriate place (e.g., transition *open* consumes the arrived token from the place *open*) and retrieves the list of user instances from the place *assigned instances*. Guards on transitions *open*, *execute*, *skip* and *redo* show that they will fire if the request is valid, i.e., if the appropriate user instance token is found in the list in the place *assigned instances*. The result of each of those four transitions is a produced open instance token in the place *instance in progress*. The inscriptions on the arcs between these transitions and the place *instance in progress* change state of the instance, accordingly to the postcondition of each action. More specifically, the new instance state is created by four functions in the inscriptions on the arcs between transitions *open*, *execute*, *skip* and *redo*, and the place *instance in progress*. These functions take three parameters: (1) user work item to which the action should be applied, (2) the old instance state that should be changed, and (3) the instance process definition – the activities of the instance type. The third parameter is retrieved from the place *tprocess*, which stores process definitions for all instance types. Functions *open_item*, *execute_item*, *skip_item* and *redo_item* create the new instance state accordingly to the postcondition of the referring action, as shown in Table A.1. When a token is produced in the place *instance in progress*, a message is sent to the *action* sub-module that the user can select the next action for the referring open instance.

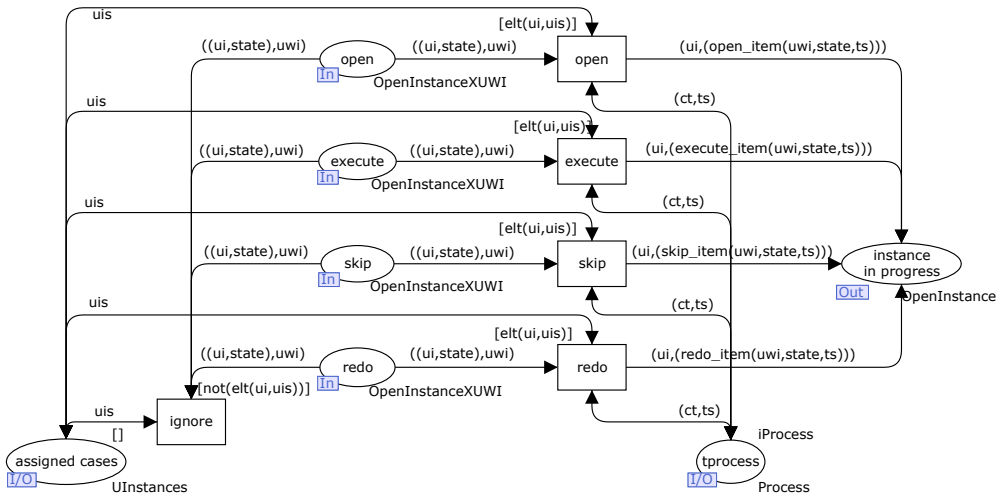


Figure A.17: FLOWer - instance distribution

The FLOWer CPN model implemented significant changes to the *basic model*. Because of its case-handling nature, the FLOWer model differs the most from the other work distribution CPN models presented in this thesis. The greatest differences are caused by the fact that the system distributes the instances and

the work items within the instances, instead of only work items. The *start work* module of the *basic model* was significantly extended because users can open, execute, skip and redo work items in FLOWer. Regardless the differences between FLOWer and process-oriented systems (modeled by the *basic model*, Staffware model and FileNet model), it was possible to extend the *basic model* to the FLOWer work distribution model.

A.4 Summary

Workflow management systems should provide flexible work distribution mechanisms for users. This will increase the work satisfaction of users and improve their ability to deal with unpredictable situations at work. Therefore, work distribution is investigated as the functionality provided for the user – workflow management systems are tested in laboratories [211, 216] or observed (in empirical research) in practice [64]. This kind of research observes systems *externally* and provides insights into *what* systems do. Analysis of the systems from an *internal* perspective can explain *how* systems provide for different work distribution mechanisms. Due to the complexity of workflow management systems as software products, internal analysis starts with developing a model of the system. Unlike the mostly used static models (e.g., UML class diagrams, entity-relationship diagrams), dynamic models (e.g., CPN models) provide for interactive investigation of work distribution as a dynamic feature. CPN models can be used for the investigation of both *what* systems do and *how* they do it.

Workflow management systems often provide for different features or use different naming for the same features. Investigation of work distribution requires analysis, evaluation and comparison of models of several systems. In order for models of different systems to be comparable, it is necessary to start with developing a common framework – a *reference model*. We use the *basic model* presented in Section 3.1.2 as a reference model for work distribution mechanisms in workflow management systems. The models of Staffware, FileNet, FLOWer are comparable because all models are developed as extensions of the same model – the *basic model*.

The model of a workflow system is structured into two modules. The *work distribution* module represents the *workflow engine* (cf. Figure 3.1 on page 48). The *work lists* module represents the *workflow client application*, which serves as an interface between the workflow engine and users (cf. Figure 3.1 on page 48). The interface between the two modules (i.e., the messages that are exchanged between them) should contain as little information as possible about the way work items are managed in modules. The *work lists* module should abstract from the way the work items are created, allocated and offered in the *work distribution* module. The reverse also holds: how work items are actually processed by users is implemented in the *work lists* module. Once a proper interface is defined, it is

easy to implement various ways of work distribution by adding/removing simple features in either one of the modules.

Work distribution mechanism determines what users can do with work items. Users of Staffware and FileNet models have the freedom to forward and suspend work. In FLOWer, as the most flexible system, users have four possibilities: execute, open, skip and redo work. Our models show that a more complex model work distribution adds messages between the *work distribution* and *work lists* modules. These new messages correspond to new actions (operations) that users can do.

Both the system-based and the patterns-based CPN models showed that one of the core elements of work distribution is the ‘allocation algorithm’. This algorithm includes rules of a specific work distribution. It is implemented in the *work distribution* module as the function *offer*, which allocates work based on (1) new work items, (2) process definition, and the (3) organizational model. This function should be analyzed further in order to discover an advanced allocation algorithm, which should be more configurable and less system-dependent.

Every system has its own method of modeling organizational structure. Staffware models groups and roles. In FileNet the organizational model includes groups of users and teams, but does not model roles. FLOWer groups users based on a hierarchy of roles, function profiles and work profiles. Thus, each of the system offers a unique predefined type of the organizational structure. Since every allocation mechanism uses elements of the organizational model, limitations of the organizational model can have a negative impact on the work distribution in the system. For example, because in Staffware one role can be assigned to only one user, it is not possible to offer a work item to a set of *call center operator-s*.

Each of the three models of workflow management systems distributes work using two hierarchy levels. Staffware and FileNet use two levels of work distribution: queue work items are first distributed to work queues, and then work items are distributed within each of the work queues. The FLOWer model starts with the instance distribution and then distributes work items of the whole instance. Although all three systems distribute work at two levels, they have unique *distribution algorithms* (the set of allocation rules implemented in the function *offer*) and *objects of distribution* (work items, queue work items, instances).

Appendix B

Evaluation of Workflow Patterns Support

B.1 Control-Flow Patterns

Table B.1: Support for control flow patterns in Staffware (SW), FileNet (FN) and FLOWer (FW) [211,213]

(+ = direct support, - = no direct support, +/- = partial support)

Nr	Pattern	SW	FN	FW
1	sequence	+	+	+
2	parallel split	+	+	+
3	synchronization	+	+	+
4	exclusive choice	+	+	+
5	simple merge	+	+	+
6	multi-choice	-	+	+
7	structured synchronizing merge	-	+	+
8	multi-merge	-	+	+/-
9	structured discriminator	-	-	-
10	arbitrary cycles	+	+	-
11	implicit termination	+	+	+
12	multiple instances without synchronization	+	+	+
13	multiple instances with a <i>priori</i> design-time knowledge	+	-	+
14	multiple instances with a <i>priori</i> run-time knowledge	+	-	+
15	multiple instances without a-priori run-time knowledge	-	-	+
16	deferred choice	-	+/-	+
17	interleaved parallel routing	-	-	+/-
18	milestone	-	-	+/-
19	cancel task	+	+	+/-
20	cancel case	-	+	+/-
21	structured loop	-	+	+
22	recursion	+	-	-
23	transient trigger	+	-	-
24	persistent trigger	-	+	+
25	cancel region	-	-	-
26	cancel multiple instance task	+	-	-
27	complete multiple instance task	-	-	+/-
28	blocking discriminator	-	-	-
29	cancelling discriminator	-	-	-
30	structured partial join	-	-	-
31	blocking partial join	-	-	-
32	cancelling partial join	-	-	-
33	generalized and-join	-	+	-
34	static partial join for multiple instances	-	-	-
35	cancelling partial join for multiple instances	-	-	-
36	dynamic partial join for multiple instances	-	-	-
37	acyclic synchronizing merge	-	-	+
38	general synchronizing merge	-	+	-
39	critical section	-	-	+/-
40	interleaved routing	-	-	+/-
41	thread merge	-	-	-
42	thread split	-	-	-
43	explicit termination	-	-	-

B.2 Resource Patterns

Table B.2: Support for resource patterns in Staffware (SW), FileNet (FN), FLOWer (FW) and *basic model* (BM) [182,211,216]

(+ = direct support, - = no direct support, +/- = partial support, o = out-of-scope)

Nr	Pattern	SW	FN	FW	BM
1	direct allocation	+	+	+	+/-
2	role-based allocation	+	+/-	+	+
3	deferred allocation	+	+	-	-
4	authorization	-	-	+	-
5	separation of duties	-	-	+	-
6	case handling	-	-	+	-
7	retain familiar	-	-	+	-
8	capability-based allocation	-	-	+	-
9	history-based allocation	-	-	-	-
10	organizational allocation	+/-	+/-	+/-	+/-
11	automatic execution	+	+	+	o
12	distribution by offer - single resource	-	-	-	-
13	distribution by offer - multiple resources	+	+	+	+
14	distribution by allocation - single resource	+	+	+	-
15	random allocation	-	-	-	+
16	round robin allocation	-	-	-	-
17	shortest queue	-	-	-	-
18	early distribution	-	-	+	-
19	distribution on enablement	+	+	+	+
20	late distribution	-	-	-	-
21	resource-initiated allocation	-	-	+	+
22	resource-initiated execution - allocated Work Item	+	+	+	+
23	resource-initiated execution - offered Work Item	+	+	-	-
24	system-determined work list management	+	+	+	o
25	resource-determined work list management	+	+	+	o
26	selection autonomy	+	+	+	+
27	delegation	+	+	-	-
28	escalation	+	+	-	-
29	deallocation	-	-	-	-
30	stateful reallocation	+/-	+	-	-
31	stateless reallocation	-	-	-	-
32	suspension/resumption	+/-	+/-	-	-
33	skip	-	-	+	o
34	redo	-	-	+	o
35	pre-do	-	-	+	o
36	commencement on creation	-	-	-	-
37	commencement on allocation	-	-	-	-
38	piled execution	-	-	-	-
39	chained execution	-	-	+	-
40	configurable unallocated work item visibility	-	-	-	o
41	configurable allocated work item visibility	-	-	+	o
42	simultaneous execution	+	+	+/-	+
43	additional resources	-	-	-	-

B.3 Data Patterns

Table B.3: Support for data patterns in Staffware (SW) and FLOWer (FW) [214,215]

(+ = direct support, - = no direct support, +/- = partial support)

Nr	Pattern	SW	FW
1	task data	-	+/-
2	block data	+	+
3	scope data	-	+/-
4	folder data	-	-
5	multiple instance data	+/-	+
6	case data	+/-	+
7	workflow data	+	-
8	environment data	+	+
9	data interaction between tasks	+	+
10	data interaction - block task to Sub-workflow	+	+/-
11	data interaction - Sub-workflow to block task	+	+/-
12	data interaction - to multiple instance task	-	+
13	data interaction - from multiple instance task	-	+
14	data interaction - case to case	+/-	+/-
15	data interaction - task to environment - push	+	+
16	data interaction - environment to task - pull	+	+
17	data interaction - environment to task - push	+/-	+/-
18	data interaction - task to environment - pull	+/-	+/-
19	data interaction - case to environment - push	-	+
20	data interaction - environment to case - pull	-	+
21	data interaction - environment to case - push	+/-	+
22	data interaction - case to environment - pull	-	+
23	data interaction - workflow to environment - push	-	-
24	data interaction - environment to workflow - pull	+/-	-
25	data interaction - environment to workflow - push	-	-
26	data interaction - workflow to environment - pull	+	-
27	data transfer by value - incoming	-	-
28	data transfer by value - outgoing	-	-
29	data transfer - copy in / copy out	-	+/-
30	data transfer by reference - unlocked	+	+
31	data transfer by reference - locked	-	+/-
32	data transformation - input	+/-	+/-
33	data transformation - output	+/-	+/-
34	data precondition - data existence	+	+
35	data precondition - data value	+	+
36	data postcondition - data existence	+/-	+
37	data postcondition - data value	+/-	+
38	event-based task trigger	+	+
39	data-based task trigger	-	+
40	data-based routing	+/-	+/-

Bibliography

- [1] CPN Tools. <http://wiki.daimi.au.dk/cpntools/cpntools.wiki>.
- [2] DECLARE. <http://declare.sf.net>.
- [3] ExSpect. <http://www.exspect.com>.
- [4] Intel. <http://www.intel.com>.
- [5] Microsoft Corporation. <http://www.microsoft.com>.
- [6] *newYAWL*. <http://www.yawl-system.com/newYAWL>.
- [7] Organization for the Advancement of Structured Information Standards (OASIS). <http://www.oasis-open.org>.
- [8] Process Mining. <http://www.processmining.org>.
- [9] Workflow Management Coalition. <http://www.wfmc.org>.
- [10] Workflow Patterns. <http://www.workflowpatterns.com>.
- [11] YAWL. <http://www.yawl-system.com/>.
- [12] TIBCO Staffware Process Monitor (SPM). <http://www.tibco.com>, 2005.
- [13] Common Public License, Version 1.0. <http://www.opensource.org/licenses/cpl1.0.php>, 31 October 2006.
- [14] GNU General Public License, Version 3. <http://www.gnu.org/copyleft/gpl.html>, 29 June 2007.
- [15] W.M.P. van der Aalst. Designing Workflows Based on Product Structures. In K. Li, S. Olariu, Y. Pan, and I. Stojmenovic, editors, *Proceedings of the ninth IASTED International Conference on Parallel and Distributed Computing Systems*, pages 337–342. IASTED/Acta Press, Anaheim, 1997.
- [16] W.M.P. van der Aalst. Flexible Workflow Management Systems: An Approach Based on Generic Process Models. In T. Bench-Capon, G. Soda, and A. Min-Tjoa, editors, *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA'99)*, volume 1677 of *Lecture Notes in Computer Science*, pages 186–195. Springer-Verlag, Berlin, 1999.
- [17] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
- [18] W.M.P. van der Aalst. On the Automatic Generation of Workflow Processes Based on Product Structures. *Computers in Industry*, 39:97–111, 1999.
- [19] W.M.P. van der Aalst. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin, 2000.

- [20] W.M.P. van der Aalst. Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. *Information Systems Frontiers*, 3(3):297–317, 2001.
- [21] W.M.P. van der Aalst. How to Handle Dynamic Change and Capture Management Information: An Approach Based on Generic Workflow Models. *International Journal of Computer Systems, Science, and Engineering*, 16(5):295–318, 2001.
- [22] W.M.P. van der Aalst. Reengineering Knock-out Processes. *Decision Support Systems*, 30(4):451–468, 2001.
- [23] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004.
- [24] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
- [25] W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, Berlin, 2005.
- [26] W.M.P. van der Aalst and P.J.S. Berens. Beyond Workflow Management: Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, New York, 2001.
- [27] W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer Verlag, Berlin, 2007.
- [28] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
- [29] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2004.
- [30] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek. An Alternative Way to Analyze Workflow Graphs. In A. Banks-Pidduck, J. Mylopoulos, C.C. Woo, and M.T. Ozsu, editors, *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, volume 2348 of *Lecture Notes in Computer Science*, pages 535–552. Springer-Verlag, Berlin, 2002.
- [31] W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. Forschungsbericht Nr. 380, Universität Karlsruhe, Institut AIFB, Karlsruhe, 1998.
- [32] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
- [33] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

-
- [34] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.
- [35] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
- [36] W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.
- [37] W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2006.
- [38] W.M.P. van der Aalst and M. Pesic. Specifying and Monitoring Service Flows: Making Web Services Process-Aware. In L. Baresi and E. Di Nitto, editors, *Test and Analysis of Web Services*, pages 11–56. Springer-Verlag, 2007.
- [39] W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
- [40] B. Abrahamsson. *Bureaucracy or Participation: The Logic of Organization*. Sage, Beverly Hills, CA, USA, 1977.
- [41] M. Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007.
- [42] M. Adams, A.H.M. ter Hofstede, W.M.P. van der Aalst, and D. Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In F. Curbera, F. Leymann, and M. Weske, editors, *Proceedings of the OTM Conference on Cooperative Information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 95–112. Springer-Verlag, 2007.
- [43] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Facilitating Flexibility and Dynamic Exception Handling in Workflows through Worklets. In O. Belo, J. Eder, O. Pastor, and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering Forum (CAiSE05 Forum)*, pages 45–50. Springer-Verlag, 2005.
- [44] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In R. Meersman and Z. Tari, editors, *Proceedings of 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308. Springer-Verlag, 2006.
- [45] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation. Technical Report BPM Center Report BPM-07-03, BPMcenter.org, 2007.
- [46] A. Agostini and G. De Michelis. Improving Flexibility of Workflow Management Systems. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000.
- [47] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. Knig, F. Leymann, R. Mller, G. Pfau, K. Plsler, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. Web Services Human Task (WSHuman-Task), version 1.0, 2007.

- [48] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Web Service Contracting: Specification and Reasoning with SCIFF. In E. Franconi, M. Kifer, and W. May, editors, *Proceedings of the 4th European Semantic Web Conference (ESWC'06), Innsbruck, Austria*, Lecture Notes in Computer Science, pages 68–83. Springer-Verlag, 2007.
- [49] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable Agent Interaction in Abductive Logic Programming: the SCIFF proof-procedure. Technical Report DEIS-LIA-06-001, DEIS, Bologna, Italy, 2006.
- [50] J.F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [51] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, G. Gunthor, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In S.Y.W. Su, editor, *Proceedings of the 12th International Conference on Data Engineering*, pages 574–581, New Orleans, USA, 1996.
- [52] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, Architectures, and Applications*. Springer-Verlag, 2003.
- [53] T. Andrews, F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
- [54] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Golland, N. Kartha, C.K. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. WS-BPEL TC OASIS, 2005.
- [55] P.C. Attie, M.P. Singh, E.A. Emerson, A. Sheth, and M. Rusinkiewicz. Scheduling Workflows by Enforcing Intertask Dependencies. *Distributed Systems Engineering Journal*, 3(4):222–238, December 1996.
- [56] P.C. Attie, M.P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *19th International Conference on Very Large Data Bases (VLDB)*, pages 134–145, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
- [57] J.C.M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.
- [58] L. Bainbridge. Ironies of Automation. In J. Rasmussen, K. Duncan, and J. Leplat, editors, *New Technology and Human Error*, New York, NY, USA, 1987. John Wiley and Son.
- [59] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In M. Bravetti and L. Kloul and G. Zavattaro, editor, *International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, volume 3670 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2005.
- [60] P. van Beek. *Exact and Approximate Reasoning about Qualitative Temporal Relations*. PhD thesis, University of Waterloo, Canada, 1990.
- [61] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60("1/3"):109–137, 1984.
- [62] A.J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, 1996. MIT Press.
- [63] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model-checking for extended timed temporal logics. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326, London, UK, 1996. Springer-Verlag.

-
- [64] J. Bowers, G. Button, and W. Sharrock. Workflow From Within and Without: Technology and Cooperative Work on the Print Industry Shopfloor. In *The Fourth European Conference on Computer-Supported Cooperative Work (ECSCW 95)*, pages 51–66, Stockholm, September 1995. Kluwer Academic Publishers, Dordrecht, The Netherlands.
- [65] T. Burns and G.M. Stalker. *The Management of Innovation*. Tavistock Publications, London, UK, 1961.
- [66] C. Bussler, S. Jablonski, and H. Schuster. A New Generation of Workflow Management Systems: Beyond Taylorism with MOBILE. *ACM SIGOIS Bulletin archive*, 17(1):17–20, 1996.
- [67] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999.
- [68] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998.
- [69] A.B. Cherns. The Principles of Socio-Technical Design. *Human Relations*, 8(29):783–792, 1976.
- [70] F. Chesani, P. Mello, M. Montali, and S. Storari. Towards a DecSerFlow Declarative Semantics Based on Computational Logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy, 2007.
- [71] V. Christophides, R. Hull, A. Kumar, and J. Simeon. Workflow Mediation Using Vortexml. *IEEE Data Engineering Bulletin*, 24(1):40–45, 2001.
- [72] P. Chrzastowski-Wachtel. A Top-down Petri Net Based Approach for Dynamic Workflow Modeling. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 336–353. Springer-Verlag, Berlin, 2003.
- [73] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.
- [74] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.
- [75] Workflow Management Coalition. Terminology and Glossary. WFMC-TC-1011 Issue 3.0, Workflow Management Coalition, February 1999.
- [76] G. Cugola. Tolerating Deviations in Process Support Systems via Vlexible Enactment of Process Models. *IEEE Transactions on Software Engineering*, 24(11):982–1001, 1989.
- [77] F. Daoudi and S. Nurcan. A Benchmarking Framework for Methods to Design Flexible Business Processes. In *Software Process Improvement and Practice*, volume 12, pages 51–63. Wiley & Sons, 2007.
- [78] H. Davulcu, M. Kifer, and I.V. Ramakrishnan. CTR-S: A Logic For Specifying Contracts in Semantic Web Services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW Alt. '04)*, pages 144–153. ACM Press, New York, NY, USA, 2004.
- [79] A.K. Alves de Medeiros. *Genetic Process Mining*. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, November 2006.
- [80] G. Decker, A. Grosskopf, and A. Barros. A Graphical Notation for Modeling Complex Events in Business Processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 27–36. IEEE Computer Society, 2007.

- [81] G. Decker, J.M. Zaha, and M. Dumas. Execution Semantics for Service Choreographies. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proceedings of the 3rd Workshop on Web Services and Formal Method (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 163–177. Springer-Verlag, 2006.
- [82] J. Dehnert and W.M.P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.
- [83] J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Berlin, 2001.
- [84] S. Demri, F. Laroussinie, and Ph. Schnoebelen. A Parametric Analysis of the State-Explosion Problem in Model Checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006.
- [85] S. Demri and Ph. Schnoebelen. The Complexity of Propositional Linear Temporal Logics in Simple Cases. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings of 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 98)*, volume 1373/1998 of *Lecture Notes in Computer Science*, pages 61–72, Paris, France, 1998. Springer-Verlag.
- [86] J. Desel. Reduction and Design of Well-behaved Concurrent Systems. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR 1990*, volume 458 of *Lecture Notes in Computer Science*, pages 166–181. Springer-Verlag, Berlin, 1990.
- [87] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
- [88] R.M. Dijkman, M. Dumas, and C. Ouyang. Formal Semantics and Analysis of BPMN Process Models. Technical Report QUT Preprint 7115, Queensland University of Technology, 2007.
- [89] B.F. van Dongen. *Process Mining and Verification*. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, July 2007.
- [90] B.F. van Dongen and W.M.P. van der Aalst. A Meta Model for Process Mining Data. In J. Casto and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*, volume 2, pages 309–320. FEUP, Porto, Portugal, 2005.
- [91] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
- [92] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw. Freeflow: Mediating Between Representation and Action in Workflow Systems. In *Proceedings of the CM Conference on Computer Supported Cooperative Work (CSCW '96)*, pages 190–198. ACM Press, New York, NY, USA, 1996.
- [93] M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
- [94] S. Dustdar. Caramba - A Process-Aware Collaboration System Supporting Ad Hoc and Collaborative Processes in Virtual Teams. *Distributed and Parallel Databases*, 15(1):45–66, 2004.
- [95] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software*

-
- Engineering (ICSE '99)*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [96] P.M. Fitts (Ed.). *Human Engineering for an Effective Air-navigation and Traffic-control System*. National Research Council Committee on Aviation Psychology, Washington, DC, USA, 1951.
- [97] J. Eder and W. Liebhart. The Workflow Activity Model (WAMO). In S. Spaccapietra and T. Yokoi, editors, *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS-95)*, pages 87–98, Vienna, Austria, 1995. University of Toronto Press.
- [98] J. Eder and W. Liebhart. Workflow Recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS96)*, pages 124–134, Brussels, Belgium, 1996. IEEE Computer Society.
- [99] F.M. van Eijnatten and A.H. van der Zwaan. The Dutch IOR Approach to Organization Design. An Alternative to Business Process Re-Engineering? *Human Relations*, 3(51):289–318, 1998.
- [100] C.A. Ellis and K. Keddera. A Workflow Change Is a Workflow. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin, 2000.
- [101] C.A. Ellis, K. Keddera, and G. Rozenberg. Dynamic Change within Workflow Systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10 – 21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York, NY, USA.
- [102] F.E. Emery. *Toward Real Democracy*. Ontario Quality of Working Life Centre/ Ministry of Labor, Toronto, Ontario, Canada, 1989.
- [103] F.E. Emery and M. Emery. Participative Design: Work and Community Life. pages 94–113, 1989.
- [104] F.E. Emery and E.C. Trist. The Causal Texture of Organisational Environments. *Human Relations*, 18(1):21–32, 1965.
- [105] H. Fayol. *General and Industrial Management*. Pitman, London, UK, 1949. Translated from the French edition (Dunod, 1925) by Constance Storrs.
- [106] J. Ferber and O. Gutknecht. Aalaadin: A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. In *Proceedings of the International Conference on Multi Agent Systems (RR-LIRMM 97189)*, pages 128–135, Montpellier, France, 1998.
- [107] FileNET. *FileNet Business Process Manager 3.0*. FileNET Corporation, Costa Mesa, CA, USA, June 2004.
- [108] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006.
- [109] D. Georgakopoulos. Teamware: An Evaluation of Key Technologies and Open Problems. *Distributed and Parallel Databases*, 15(1):9–44, 2004.
- [110] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
- [111] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.

- [112] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.
- [113] D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems (FORTE '02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, London, UK, 2002. Springer-Verlag.
- [114] F. Gilbreth. *Primer of Scientific Management*. D. Van Nostrand Co., New York, NY, USA, 1914.
- [115] N. Glance, D. Pagani, and R. Pareschi. Generalised Process Structure Grammars (GPSG) for Flexible Representations of Work. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW96)*, pages 190–198. ACM Press, New York, NY, USA, 1996.
- [116] M. Goergeff and J. Pyke. *Staffware White Paper: Dynamic Process Orchestration*. Precedence research, Staffware Plc, March 2003. Version 1.
- [117] D. Grigori, F. Charoy, and C. Godart. Anticipation to Enhance Flexibility of Workflow Execution. In H.C. Mayr, J. Lazanský, G. Quirchmayr, and P. Vogel, editors, *Proceeding of the 12th International Conference on Database and Expert Systems Applications (DEXA 2001)*, volume 2113 of *Lecture Notes in Computer Science*, pages 264–273, Munich, Germany, 2001. Springer-Verlag.
- [118] C.W. Günther, S. Rinderle, M. Reichert, and W.M.P. van der Aalst. Change Mining in Adaptive Process Management Systems. In R. Meersman and Z. Tari, editors, *Proceedings of 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 2006.
- [119] C. Hagen and G. Alonso. Flexible Exception Handling in the OPERA Process Support System. In *International Conference on Distributed Computing Systems*, pages 526–533, 1998.
- [120] C. Hagen and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.
- [121] J.J. Halliday, S.K. Shrivastava, and S.M. Wheeler. Flexible Workflow Management in the OPENflow System. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 01)*, pages 82–98, Washington, DC, USA, 2001. IEEE Computer Society.
- [122] G. Hart. The Mythical, Magical Number 7. *Intercom*, April:38–39, 2006.
- [123] K. van Hee, O. Oanea, A. Serebrenik, N. Sidorova, and M. Voorhoeve. LogLogics: A Logic for History-Dependent Business Processes. *Science of Computer Programming*, 65(1):30–40, 2007.
- [124] K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335–354. Springer-Verlag, Berlin, 2003.
- [125] P. Heintz, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A Comprehensive Approach to Flexibility in Workflow Management Systems. In *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC '99)*, pages 79–88. ACM Press, New York, NY, USA, 1999.

-
- [126] C. Helein. Workflow and Process Synchronization with Interaction Expressions and Graphs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 243–252, Heidelberg, Germany, 2001. IEEE Computer Society.
- [127] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, MA, USA, 1988.
- [128] P.G. Herbst. *Socio-Technical Design: Strategies in Multi-Disciplinary Research*. London Tavistock Publications, London, UK, 1974.
- [129] T. Herrmann, M. Hoffmann, G. Kunau, and K.-U. Loser. A Modeling Method for the Development of Groupware Applications as Socio-Technical Systems. *Behaviour and Information Technology*, 18(8):313–323, 1999.
- [130] T. Herrmann and K.-U. Loser. Vagueness in Models of Socio-Technical Systems. *Behaviour and Information Technology*, 23(2):119–135, 2004.
- [131] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.
- [132] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.
- [133] InConcert. *InConcert Process Designer's Guide*. InConcert Inc., Cambridge, Massachusetts, 1997.
- [134] Institute of Transportation Engineers. Freeway Management and Operations Handbook. ITE Journal, Washington, DC, USA, 2004.
- [135] S. Jablonski. MOBILE: A Modular Workflow Model and Architecture. In *Proceedings of 4th International Working Conference on Dynamic Modelling and Information Systems*, Nordwijkerhout, The Netherlands, 1994.
- [136] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
- [137] M.H. Jansen-Vullers, P.A.M. Kleingeld, M.W.N.C. Looschilder, and H.A. Reijers. Performance Measures to Evaluate the Impact of Best Practices. In B. Pernici and J.A. Gulla, editors, *Proceedings of Workshops and Doctoral Consortium of the 19th International Conference on Advanced Information Systems Engineering (BPMDS workshop)*, volume 1, pages 359–368, Trondheim, 2007. Tapir Academic Press.
- [138] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
- [139] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3–4):209–414, 2007.
- [140] G. Joeris. Defining Flexible Workflow Execution Behaviors. In P. Dadam and M. Reichert, editors, *Workshop Informatik 99: Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, volume 24 of *CEUR Workshop Proceedings*, pages 49–55, Paderborn, Germany, 1999.
- [141] G. Joeris. Decentralized and Flexible Workflow Enactment Based on Task Coordination Agents. In B. Wangler and L. Bergman, editors, *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *Lecture Notes in Computer Science*, pages 41–62, Stockholm, Sweden, September 2000. Springer-Verlag.
- [142] N. Jordan. Allocation of Functions Between Man and Machines in Automated Systems. *Journal of Applied Psychology*, 47(3):161–165, June 1963.

- [143] J.J. Kaasbøll and O. Smørðal. Human Work as Context for Development of Object Oriented Modelling Techniques. In *Proceedings of the Working Conference on Method Construction and Tool Support (IFIP TC8, WG8.1/8.2)*, pages 111–125, London, UK, 1996. Chapman & Hall, Ltd.
- [144] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
- [145] P.J. Kammer, G.A. Bolcer, R.N. Taylor, A.S. Hitomi, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work*, 9(3-4):269–292, 2000.
- [146] G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
- [147] G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley, Reading MA, 1998.
- [148] M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, volume 9 of *Special issue of the journal of Computer Supported Cooperative Work*, 2000.
- [149] J. Klingemann. Controlled Flexibility in Workflow Management. In B. Wangler and L. Bergman, editors, *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *Lecture Notes in Computer Science*, pages 126–141, Stockholm, Sweden, September 2000. Springer-Verlag.
- [150] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, and I. Trickovic. WS-BPEL Extension for People BPEL4People. IBM Corporation, <http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/>, 2005.
- [151] M. Kradolfer and A. Geppert. Dynamic Workflow Schema Evolution Based on Workflow Type Versioning and Workflow Migration. In *Proceedings of 4th International Conference on Cooperative Information Systems (CoopIS 1999)*, pages 104–114, Edinburgh, Scotland, 1999. IEEE Computer Society.
- [152] L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner’s Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
- [153] A. Kumar and J.L. Zhao. A Framework for Dynamic Routing and Operational Integrity Controls in a Workflow Management System. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS 1996)*, volume 3, pages 492–501. IEEE Computer Society, 1996.
- [154] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Inducing Declarative Logic-Based Models from Labeled Traces. In *Proceedings of the 5th International Conference on Business Process Management*, number 4714 in *Lecture Notes in Computer Science*, pages 344–359. Springer, 2007.
- [155] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. A Methodology for Learning Social Integrity Constraints from Labeled Service Interaction Logs. Technical Report DEIS-LIA-07-001, DEIS, Bologna, Italy, 2007.
- [156] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. Applying Inductive Logic Programming to Process Mining. In *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP 2007)*, volume 4894 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, 2008.
- [157] P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event Driven Process Chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets*

-
- 1998, volume 1420 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, Berlin, 1998.
- [158] T. Latvala. Efficient Model Checking of Safety Properties. In *Proceedings of the 10th SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer Verlag, Berlin, 2003.
- [159] F. Leymann and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1):102–123, 1997.
- [160] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
- [161] H. Lin, Z. Zhao, H. Li, and Z. Chen. A Novel Graph Reduction Algorithm to Identify Structural Conflicts. In *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-35)*. IEEE Computer Society Press, 2002.
- [162] R. Lu. *Constraint-based Flexible Business Process Management*. PhD thesis, University of Queensland, Brisbane, Australia, May 2008.
- [163] R. Lu, S. Sadiq, V. Padmanabhan, and G. Governatori. Using a temporal constraint network for business process execution. In *In proceedings of the 17th Australasian Database Conference (ADC '06)*, pages 157–166, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.
- [164] L. Thao Ly, S. Rinderle, and P. Dadam. Semantic Correctness in Adaptive Process Management Systems. In S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors, *Proceedings of the 4th International Conference (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 193–208. Springer Verlag, Berlin, 2006.
- [165] L. Thao Ly, S. Rinderle, and P. Dadam. Integration and Verification of Semantic Constraints in Adaptive Process Management Systems. *Data & Knowledge Engineering*, 64(1):3–23, 2008.
- [166] T.W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C.S. Osborn, A. Bernstein, G. Herman, M. Klein, and E. O'Donnell. Tools for Inventing Organizations: Toward a Handbook of Organizational Processes. *Management Science*, 45(3):425–443, 1999.
- [167] P. Mangan and S. Sadiq. On Building Workflow Models for Flexible Processes. In *Proceedings of the 13th Australasian Database Conference (ADC 02)*, pages 103–109, Darlinghurst, NSW, Australia, 2002. Australian Computer Society, Inc.
- [168] E. Mayo. *The Human Problems of an Industrialised Civilization*. Macmillan, New York, NY, USA, 1933.
- [169] D. McGregor. *The Human Side of Enterprise*. McGraw-Hill, New York, NY, USA, 1960.
- [170] J. Mendling. *Detection and Prediction of Errors in EPC Business Process Models*. PhD thesis, Vienna University of Economics and Business Administration, Austria, 2007.
- [171] Sun Microsystems. Java Programming Language. <http://java.sun.com>.
- [172] G.A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956.
- [173] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
- [174] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–40, 1992.
- [175] M. zur Muehlen. *Workflow-based Process Controlling: Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin, 2004.

- [176] N. Mulyar, M. Pesic, W.M.P. van der Aalst, and M. Peleg. Declarative and Procedural Approaches for Modelling Clinical Guidelines: Addressing Flexibility Issues. In A. ter Hofstede, B. Benatallah, and H.Y. Paik, editors, *Business Process Management Workshops, BPM 2007 International ProHealth Workshop*, volume 4928 of *Lecture Notes in Computer Science*, pages 335–346, Brisbane, Australia, 2008. Springer-Verlag.
- [177] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [178] OASIS. Web Services Business Process Execution Language for Web Services, version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>, 2007.
- [179] Object Management Group (OMG). Business Process Modeling Notation (BPML), Version 1.1, 2008. OMG Available Specification.
- [180] Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
- [181] M. Pesic and W.M.P. van der Aalst. A Declarative Approach for Flexible Business Processes. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, Workshop on Dynamic Process Management (DPM 2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2006.
- [182] M. Pesic and W.M.P. van der Aalst. Modelling Work Distribution Mechanisms Using Colored Petri Nets. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3–4):327–352, 2007.
- [183] M. Pesic, M.H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 287–298, Washington, DC, USA, 2007. IEEE Computer Society.
- [184] M. Pesic, M.H. Schonenberg, N. Sidorova, and W.M.P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In R. Meersman and Z. Tari, editors, *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 77–94. Springer-Verlag, 2007.
- [185] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. *ACM SIGPLAN Notices*, 42(9):141–152, 2007.
- [186] A.B. Raposo and H. Fuks. Defining Task Interdependencies and Coordination Mechanisms for Collaborative Systems. In M. Blay-Fornarino, A.M. Pinna-Dery, K. Schmidt, and P. Zaratè, editors, *Cooperative Systems Design*, volume 74 of *Frontiers in Artificial Intelligence and Applications*, pages 88–103, Amsterdam, The Netherlands, 2002. IOS Press.
- [187] A.B. Raposo, L.P. Magalhaes, I.L.M. Ricarte, and H. Fuks. Coordination of Collaborative Activities: A Framework for the Definition of Tasks Interdependencies. In *Proceedings of the 7th International Workshop on Groupware (CRIWG)*, pages 170–179. IEEE Computer Society, 2001.
- [188] G. Regev and A. Wegmann. A Regulation-Based View on Business Process and Supporting System Flexibility. In *CAiSE05 Workshop on Business Process Modeling, Design and Support (BPMD05)*, pages 35–42, Porto, Portugal.
- [189] M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
- [190] M. Reichert, P. Dadam, and T. Bauer. Dealing with forward and backward jumps in workflow management systems. *Software and Systems Modeling*, 2(1):37–58, March 2003.

-
- [191] M. Reichert, S. Rinderle, and P. Dadam. ADEPT Workflow Management System: Flexible Support for Enterprise-Wide Business Processes. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *Business Process Management 2003*, volume 2678 of *Lecture Notes in Computer Science*, pages 370–379. Springer-Verlag, 2003. Tool presentation.
- [192] M. Reichert, S. Rinderle, U. Kreher, H. Acker, M. Lauer, and P. Dadam. ADEPT - Next Generation Process Management Technology. In *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE Forum '06)*, volume 231 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006. Tool Demonstration.
- [193] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive Process Management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 1113–1114, Tokyo, Japan, 2005. IEEE Computer Society. Tool Demonstration.
- [194] H. Reijers. *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002.
- [195] H. Reijers, J. Rigter, and W.M.P. van der Aalst. The Case Handling Case. *International Journal of Cooperative Information Systems*, 12(3):365–391, 2003.
- [196] H.A. Reijers. Workflow Flexibility: The Forlorn Promise. In *15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006)*, pages 271–272, Manchester, United Kingdom. IEEE Computer Society.
- [197] H.A. Reijers, S. Limam, and W.M.P. van der Aalst. Product-based Workflow Design. *Journal of Management Information Systems*, 20(1):229–262, 2003.
- [198] H.A. Reijers and S. Limam Mansar. Best Practices in Business Process Redesign: An Overview and Qualitative Evaluation of Successful Redesign Heuristics. *Omega: The International Journal of Management Science*, 33(4):283–306, 2005.
- [199] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
- [200] A. Reuter and F. Schwenkreis. ConTracts: A Low-Level Mechanism for Building General-Purpose Workflow Management Systems. *Data Engineering Bulletin*, 18(1):4–10, 1995.
- [201] S. Rinderle, M. Reichert, and P. Dadam. Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
- [202] S. Rinderle, M. Reichert, and P. Dadam. Flexible Support of Team Processes by Adaptive Workflow Systems. *Distributed and Parallel Databases*, 16(1):91–116, 2004.
- [203] S. Rinderle, M. Reichert, and P. Dadam. On Dealing with Structural Conflicts between Process Type and Instance Changes. In *Proceedings of the 2nd International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 274–289. Springer-Verlag, 2004.
- [204] F. Roethlisberger and W. Dickson. *Management and the Worker: An Account of a Research Program Conducted by the Western Electric Company, Chicago*. Harvard University Press, Cambridge, 1939.
- [205] W.P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, New York, NY, USA, 2001.
- [206] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.

- [207] A. Rozinat and W.M.P. van der Aalst. Decision Mining in ProM. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 420–425. Springer-Verlag, Berlin, 2006.
- [208] N. Russell. *Foundations of Process-Aware Information Systems*. Phd thesis, Queensland University of Technology, June 2007.
- [209] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Workflow Exception Patterns. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering Forum (CAiSE05 Forum)*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302. Springer-Verlag, 2006.
- [210] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *newYAWL: Specifying a Workflow Reference Language using Coloured Petri Nets*. In K. Jensen, editor, *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007)*, volume 584 of *DAIMI*, pages 107–127, Aarhus, Denmark, October 2007. University of Aarhus.
- [211] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, Berlin, 2005.
- [212] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and D. Edmond. *newYAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-Flow, Data and Resource Perspectives*. Technical Report BPM Center Report BPM-07-05, BPMcenter.org, 2007.
- [213] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM Center Report BPM-06-22, BPMcenter.org, 2006.
- [214] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.
- [215] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, Berlin, 2005.
- [216] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Resource Patterns. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2005.
- [217] N.C. Russell and W.M.P. van der Aalst. Evaluation of the BPEL4People and WS-HumanTask Extensions to WS-BPEL 2.0 using the Workflow Resource Patterns. Technical Report BPM Center Report BPM-07-11, BPMcenter.org, 2007.
- [218] H. Saastamoinen and G.M. White. On Handling Exceptions. In N. Comstock and C. Ellis, editors, *Proceedings of the ACM Conference on Organizational Computing Systems (COCS95)*, pages 302–310, Milpitas, CA, USA, 1995. ACM Press, New York, NY, USA.
- [219] S. Sadiq, O. Marjanovic, and M.E. Orłowska. Managing Change and Time in Dynamic Workflow Processes. *International Journal of Cooperative Information Systems*, 9(1-2):93–116, 2000.
- [220] S. Sadiq, W. Sadiq, and M.E. Orłowska. Pockets of Flexibility in Workflow Specification. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER 01)*, pages 513–526, London, UK, 2001. Springer-Verlag.

-
- [221] W. Sadiq and M.E. Orlowska. Modeling and Verification of Workflow Graphs”, address =.
- [222] W. Sadiq and M.E. Orlowska. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. In M. Jarke and A. Oberweis, editors, *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE '99)*, volume 1626 of *Lecture Notes in Computer Science*, pages 195–209. Springer-Verlag, Berlin, 1999.
- [223] W. Sadiq and M.E. Orlowska. Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, 25(2):117–134, 2000.
- [224] M.W. Scerbo. Adaptive Automation. In W. Karwowski, editor, *International Encyclopedia of Ergonomics and Human Factors*, pages 1077–1079, London, UK, 2001. Taylor and Francis, Inc.
- [225] A.W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994.
- [226] M.H. Schonenberg, R.S. Mans, N.C. Russell, N. Mulyar, and W.M.P. van der Aalst. Taxonomy of process flexibility. Technical Report BPM Center Report BPM-06-22, BPM-center.org, 2007.
- [227] M.H. Schonenberg, R.S. Mans, N.C. Russell, N. Mulyar, and W.M.P. van der Aalst. Process Flexibility: a Survey of Contemporary Approaches. In *To appear in CIAO! workshop proceedings*, Lecture Notes in Business Information Processing. Springer-Verlag, 2008.
- [228] M.H. Schonenberg, R.S. Mans, N.C. Russell, N. Mulyar, and W.M.P. van der Aalst. Towards a Taxonomy of Process Flexibility. In *To appear in Proceedings of CAiSE08 Forum*, 2008.
- [229] P. Schutte. Complementation: An Alternative to Automation. *Journal of Information Technology Impact*, 1(3):113–118, 1999.
- [230] A. Sheth. From Contemporary Workflow Process Automation to Adaptive and Dynamic Work Activity Coordination and Collaboration. In R. Wagner, editor, *Database and Expert Systems Applications, 8th. International Workshop, DEXA '97, Proceedings*, pages 24–27, Toulouse, France, September 1997. IEEE Computer Society Press, Los Alamitos, California, 1997.
- [231] L.U. de Sitter, J.F. den Hertog, and B. Dankbaar. From Complex Organisations with Simple Jobs to Simple Organizations with Complex Jobs. *Human Relations*, 510(5):497–534, 1997.
- [232] S. Carlsen J. Krogstie A. Slyberg and O.I. Lindland. Evaluating Flexible Workflow Systems. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences (HICSS-30)*, volume 2, pages 230–239, Wailea, HI, USA, 1997. ieeecs.
- [233] R.A. Snowdon, B.C. Warboys, R.M. Greenwood, C.P. Holland, P.J. Kawalek, and D.R. Shaw. On the Architecture and Form of Flexible Process Support. In *Software Process Improvement and Practice*, volume 12, pages 21–34. Wiley & Sons, 2007.
- [234] P. Soffer. On the Notion of Flexibility in Business Processes. In *Workshop on Business Process Modeling, Design and Support (BPMDS05), Proceedings of CAiSE05 Workshops*, pages 35–42, 2005.
- [235] Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
- [236] Software-Ley. *COSA Activity Manager*. Software-Ley GmbH, Pullheim, Germany, 2002.
- [237] Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.

- [238] Staffware. *Using the Staffware Process Client*. Staffware, plc, Berkshire, United Kingdom, May 2002.
- [239] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, second edition, 1994.
- [240] R. van Stiphout, T.D. Meijler, A. Aerts, D. Hammer, and R. Le Comte. TREX: Workflow Transaction by Means of Exceptions. In H.J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Proceedings of the Sixth International Conference on Extending Database Technology (EDBT98)*, pages 21–26, Valencia, Spain, 1998.
- [241] D.M. Strong and S.M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems (TOIS)*, 13(2):206–233, 1995.
- [242] F.W. Taylor. *The Principles of Scientific Management*. Harper Bros, New York, NY, USA, 1911.
- [243] G. Trajcevski, C. Baral, and J. Lobo. Formalizing (and Reasoning about) the Specifications of Workflows. In O. Etzion and P. Scheuermann, editors, *Proceedings of the OTM Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.
- [244] E.L. Trist and K.W. Bamforth. Some Social and Psychological Consequences of the Longwall Method of Coal-Getting. *Human Relations*, 1(4):3 – 38, 1951.
- [245] E.L. Trist, G.W. Higgin, H. Murray, and A.B. Pollock. *Organizational Choice: Capabilities of Groups at the Coal Face Under Changing Technologies; The Loss, Re-Discovery and Transformation of a Work Tradition*. Tavistock Publications, London, UK, 1963. reissued Garland New York, NY, USA, 1987.
- [246] F. M. van Eijnatten. *The Paradigm that Changed the Work Place*. Van Gorcum, Assen, The Netherlands, 1993.
- [247] I. Vanderfeesten. Designing Workflow Systems. An Algorithmic Approach to Process Design and a Human Oriented Approach to Process Automation. Master’s thesis, Eindhoven University of Technology, August 2004.
- [248] I. Vanderfeesten and H.A. Reijers. A Human-Oriented Tuning of Workflow Management Systems. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, September 2005.
- [249] I. Vanderfeesten, H.A. Reijers, and W.M.P. van der Aalst. Product Based Workflow Design With Case Handling Systems. BETA Working Paper Series, WP 189, Eindhoven University of Technology, Eindhoven, 2006.
- [250] I. Vanderfeesten, H.A. Reijers, and W.M.P. van der Aalst. An Evaluation of Case Handling Systems for Product Based Workflow Design. In J. Cardoso, J. Cordeiro, and J. Filipe, editors, *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS 2007)*, volume Information Systems Analysis and Specification of Volume on, pages 39–46, Medeira, Portugal, 2007. Institute for Systems and Technologies of Information, Control and Communication, INSTICC.
- [251] I. Vanderfeesten, H.A. Reijers, and W.M.P. van der Aalst. Product Based Workflow Support: A Recommendation Service for Dynamic Workflow Execution. BPM Center Report BPM-08-03, BPM Center, 2008. <http://www.BPMcenter.org>.
- [252] I. Vanderfeesten, H.A. Reijers, and W.M.P. van der Aalst. Product Based Workflow Support: Dynamic Workflow Execution. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE '08)*, Montpellier, France, 2008.

-
- [253] M.Y. Vardi. Branching vs. Linear Time: Final Showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22, London, UK, 2001.
- [254] E. Verbeek. *Verification of WF-nets*. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, June 2004.
- [255] H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based Workflow Diagnosis Tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, Berlin, 2000.
- [256] J. Wainer and F. de Lima Bezerra. Constraint-based flexible workflows. In *Proceedings of the 9th International Workshop on Groupware: Design, Implementation, and Use (CRIWG 2003)*, volume 2806, pages 151 – 158. Springer-Verlag, 2003.
- [257] J. Wang and A. Kumar. A Framework for Document-Driven Workflow Systems. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 285–301. Springer-Verlag, September 2005.
- [258] B. Weber, B.F. van Dongen, M. Pesic, C.W. Günther, and W.M.P. van der Aalst. Supporting Flexible Processes Through Recommendations Based on History. BETA Working Paper Series, WP 212, Eindhoven University of Technology, Eindhoven, 2007.
- [259] B. Weber, M. Reichert, S. Rinderle, and W. Wild. Towards a Framework for the Agile Mining of Business Processes. In C. Bussler and A. Haller, editors, *In proceedings of the 1st Workshop on Business Process Intelligence (BPI 2005)*, volume 3812 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 2005.
- [260] B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In J. Krogstie, A.L. Opdahl, and G. Sindre, editors, *Proceedings of the 19th International Conference (CAiSE 2007)*, volume 4495 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 2007.
- [261] B. Weber, W. Wild, and R. Breu. CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-based Reasoning. In *Proceedings of European Conference on Casebased Reasoning (ECCBR04)*, volume 3155 of *Lecture Notes in Computer Science*, pages 434–448, Madrid, Spain, 2004. Springer-Verlag.
- [262] M. Weber. *The Theory of Social and Economic Organization*. The Free Press, 1947. Translated from the German edition by A.M.Henderson and Talcott Parsons.
- [263] A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process Mining with the Heuristics Miner-algorithm. BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven, 2006.
- [264] M. Weske. Flexible Modeling and Execution of Workflow Activities. In *Proceedings of the 31st Annual Hawaii International Conference on System Science (HICSS-314)*, volume 7, pages 713–722. IEEE Computer Society, 1998.
- [265] M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In R. Sprague, editor, *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society Press, Los Alamitos, California, 2001.
- [266] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag Berlin Heidelberg, 2007.
- [267] D. Wodtke and G. Weikum. A Formal Foundation for Distributed Workflow Execution Based on State Charts.

-
- [268] M.T. Wynn, W.M.P. van der Aalst, and A.H.M. ter Hofstede D. Edmond. Verifying Workflows with Cancellation Regions and OR-Joins: An Approach Based on Reset Nets and Reachability Analysis. In S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors, *Proceedings of the 4th International Conference (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 389–394. Springer-Verlag, 2006.
- [269] J.M. Zaha, A.P. Barros, M. Dumas, and A.H.M. ter Hofstede. Let’s Dance: A Language for Service Behavior Modeling. In R. Meersman and Z. Tari, editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162. Springer-Verlag, 2006.

Constraint-Based Workflow Management Systems: Shifting Control to Users

Summary

Many organizations use information technology to support various aspects of their business processes: the operational aspect, collaboration between employees, etc. Workflow management systems aim at *supporting* the operational aspect of complex business processes by using process models to automate the ordering of activities (i.e., flow of work). The term ‘support’ here relates to the ability of workflow management systems to control the execution of business processes.

Contemporary workflow management systems lack *flexibility*, i.e., the system controls in detail how employees should execute business processes. While workflow management systems deal well with predictable business processes, they are not able to handle unforeseen situations, which occur often in real-life business processes. Although employees mostly have the knowledge and experience that enables them to deal with exceptional situations, they are not able to apply the right action because the system enforces the standard procedure of work. This often has various undesired consequences: work is done ‘outside’ the system, work cannot be done in the appropriate way, dissatisfaction of employees, resistance towards the system, etc. As a result, workflow management systems cannot be used properly if it is necessary that employees control the execution of business processes.

This thesis proposes a new approach to workflow management systems that can facilitate contemporary business processes in a better way by enabling a better balance between flexibility and support. As opposed to traditional approaches which use *procedural process models* to *explicitly* (i.e., step-by-step) specify the execution procedure, the proposed approach aims at the specification of business processes *using constraints*, i.e., processes are modeled by rules that should be

followed while executing business processes. Constraint-based models *implicitly* specify the execution procedure by means of constraints: any execution that does not violate constraints is possible. In addition to proposing a constraint-based approach, a concrete language for specification of constraints is given and the proof-of-concept prototype DECLARE is described.

On the one hand, constraint-based management systems are flexible, which allows employees to deal with specific (e.g., unpredicted) situations in the most adequate way. On the other hand, constraint-based management systems can support employees when it comes to aspects of business processes that are too complex for humans to handle. There are several ways in which constraint-based management systems can provide both flexibility and support. We classify flexibility into flexibility by *design*, *change*, *underspecification* and *deviation*.

Design Constraint-based workflow management systems allow any execution of business processes that does not violate specified constraints. In this way, business processes do not have to be explicitly specified at the procedural level. Instead, it is enough to predefine a set of general rules (i.e., constraints). The execution procedure is implicitly derived from these constraints: employees can execute business processes in any way within the boundaries set up by constraints.

Change By adding or removing constraints or activities at runtime, the approach proposed in this thesis provides a simple method for changing constraint-based process models during execution and migrating instances from an existing process to a new process.

Underspecification Underspecified process models contain unspecified parts that can be seen as ‘black boxes’ that allow employees to decide during execution what to do. This type of flexibility is often supported by workflow management systems and our DECLARE prototype supports it as well through a connection with the YAWL system.

Deviation The constraint-based process models we propose can contain both mandatory constraints (i.e., constraints that must be fulfilled) and optional constraints (i.e., constraints that can be violated). Optional constraints represent rules that are advised but not enforced during execution of business processes. Therefore, our approach supports flexibility by deviation, as long as no mandatory constraints are violated.

The flexibility provided by our constraint-based approach offers a wide range of possibilities for execution of business processes. Therefore, constraint-based workflow management systems should offer sufficient support when it comes to issues that are too complex for humans to deal with. A workflow management system, based on our constraint-based approach, offers support for the *design*, *execution* and *diagnosis* of business processes.

Design Constraint-based models can contain an arbitrary number of constraints

that interfere in subtle ways, which can cause errors in models. For example, it might be the case that some of the constraints are contradictory to each other. The variety of constraints and their semantics makes it hard for humans to detect possible errors by means of reasoning. Therefore, this thesis presents a method for the automated verification of constraint-based models at *design time*.

Execution During execution, support is provided by *enforcing correct execution, constraint state monitoring and recommendations*.

As stated before, the approach proposed in this thesis enables constraint-based workflow management systems to enforce a *correct execution* of a business process, i.e., an execution that satisfies all constraints from the model.

Furthermore, in order to execute a business process in a correct way, users must keep track of the states of all constraints during the execution. For example, some constraints might already be satisfied, while others still need to be satisfied in the future. The diversity and complexity of constraints makes it difficult for humans to constantly keep track of all constraints in all business processes that are being executed. Therefore, the approach presented in this thesis enables constraint-based workflow management systems to constantly *monitor states* of constraints and present these states to users in an intuitive manner.

Finally, since the flexibility of the constraint-based approach gives users many possibilities when executing business processes, deciding on how exactly to execute a business process can sometimes be too difficult. For example, it might be the case that the user is inexperienced, it is not clear what is the most appropriate execution for given situation, etc. Therefore, users of constraint-based workflow management systems can make use of *run-time recommendations*. These recommendations are generated based on the experiences from past executions of business processes and can be tuned towards achieving a certain goal (e.g., minimize through put times). The DECLARE prototype uses the recommendation service of the ProM tool to provide run-time recommendations for the execution of constraint-based models. These recommendations are intended to help DECLARE users when choosing how to execute processes, but users can as well choose to act against the recommendations, i.e., the support is not enforced.

Diagnosis When it comes to workflow management systems, diagnosis of these systems is performed after the execution of business processes. The inherent flexibility of constraint-based workflow management systems may allow for a wide variety of business process executions. Therefore, a-posteriori analysis of executed business processes is particularly interesting for these systems. The DECLARE prototype creates logs containing information about executed processes. These logs can be used for various types of analysis, all

supported by the process mining tool ProM. For example, analysis results can show how the processes were actually executed and indicate how to improve constraint-based process models.

The constraint-based approach and all principles related to it that are presented in this thesis are implemented in the DECLARE prototype. This prototype shows that the ideas presented can be turned into in a fully functional workflow management system. In order to keep the prototype as flexible as possible, we support several constraint languages, such as ConDec, DecSerFlow and CIGDec.

Constraint-Based Workflow Management Systems: Shifting Control to Users

Samenvatting

Organisaties gebruiken vaak informatietechnologie om verschillende aspecten van hun bedrijfsvoering, zoals de operationele bedrijfsprocessen en de samenwerking tussen medewerkers, te ondersteunen. Workflowmanagementsystemen proberen met name het operationele aspect van de bedrijfsvoering te ondersteunen, door gebruik te maken van procesmodellen. Met ondersteuning wordt hier dus het afdwingen van de volgorde waarin taken uitgevoerd worden, bedoeld.

De workflowmanagementsystemen van tegenwoordig bieden weinig tot geen *flexibiliteit*, want deze systemen bepalen tot in detail hoe medewerkers operationele processen uit moeten voeren. Hoewel dit goed werkt in situaties waar deze processen voorspelbaar verlopen, presteert een dergelijke aanpak erg slecht als onvoorziene situaties steeds meer voor blijken te komen. Dit komt, doordat medewerkers vaak wel de ervaring en kennis hebben om met dergelijke onvoorziene situaties om te gaan, maar het systeem hun daartoe niet in staat stelt. Dit leidt vaak tot vervelende consequenties: werk dat “buiten het systeem om” gedaan wordt, werk dat niet op de juiste manier afgehandeld kan worden, ontevredenheid van medewerkers, verzet tegen het gebruik van computersystemen, enzovoorts. Hieruit blijkt dat, als medewerkers controle moeten houden over de uitvoering van bedrijfsprocessen, workflowmanagementsystemen niet op de juiste manier gebruikt kunnen worden.

In dit proefschrift wordt een nieuwe aanpak voorgesteld om workflowmanagementsystemen te ontwerpen die, door middel van een betere balans tussen flexibiliteit en ondersteuning, medewerkers wel de mogelijkheid biedt om controle te houden over de uitvoering van bedrijfsprocessen. Traditionele aanpakken leggen door middel van *procedurele procesmodellen expliciet* (stap voor stap) vast

hoe de operationele procedures eruitzien. Dit proefschrift stelt echter voor om de operationele processen te specificeren aan de hand van *regels*. Deze regels leggen de grenzen vast waarbinnen bedrijfsprocessen uitgevoerd moeten worden; alles mag, zolang de regels gevolgd worden. In die zin specificeren zij de te volgen procedures *impliciet*. Naast een op regels gebaseerd systeem, presenteert dit proefschrift een concrete taal om die regels in te specificeren en een prototype van een, op die taal gebaseerde, implementatie van zo'n systeem, genaamd DECLARE.

Aan de ene kant zijn op regels gebaseerde workflowmanagementsystemen flexibel, zodat medewerkers zo adequaat mogelijk kunnen reageren op specifieke (onvoorspelbare) situaties. Aan de andere kant moeten dergelijke systemen ondersteuning bieden op die aspecten die te complex zijn om door gebruikers afgehandeld te worden. Die flexibiliteit en ondersteuning worden, door de aanpak van dit proefschrift, op verschillende manieren geboden. Flexibiliteit kan geclassificeerd worden in vier categorieën: *Ontwerp*, *Verandering*, *Onderspecificatie* en *Variatie*.

Ontwerp Op regels gebaseerde workflow management systemen laten toe dat processen uitgevoerd worden, zolang de regels gevolgd worden. Daardoor hebben deze processen geen expliciete specificatie nodig. In plaats daarvan is het genoeg om een verzameling van generieke regels te specificeren. De operationele procedure van een process wordt dan impliciet afgeleid van de regels: medewerkers kunnen hun taken in willekeurige volgorde uitvoeren, binnen de grenzen van de gegeven regels.

Verandering Doordat tijdens de executie van processen regels en activiteiten kunnen worden toegevoegd en verwijderd, biedt deze aanpak een simpele methode om processen aan te passen. Eventuele lopende zaken kunnen worden aangepast aan de nieuwe situatie terwijl er nog aan gewerkt wordt.

Onderspecificatie Ondergespecificeerde modellen bevatten ongespecificeerde delen, die gezien kunnen worden als "black boxes". Zulke "black boxes" laten de medewerker op het laatste moment beslissen hoe een bepaalde activiteit eruitziet. Een dergelijke mate van flexibiliteit wordt vaak ondersteund door workflowmanagementsystemen en het prototype DECLARE ondersteunt dit dan ook, door samen te werken met het YAWL systeem.

Variatie De modellen in de aanpak uit dit proefschrift bieden naast de bindende regels (regels die per sé gevolgd moeten worden) ook de mogelijkheid tot het specificeren van facultatieve regels. Facultatieve regels zijn regels waarvan wel geadviseerd wordt om ze te volgen, maar waarvan het mogelijk is ze tijdens de executie van de bedrijfsprocessen te overtreden. Op die manier wordt variatie in het operationele process toegestaan, zolang bindende regels niet overtreden worden.

De flexibiliteit van de in dit proefschrift voorgestelde aanpak biedt veel mogelijkheden tijdens de executie van bedrijfsprocessen. Daarom is het van belang dat aan medewerkers voldoende ondersteuning geboden wordt op die punten

waar het voor mensen te moeilijk wordt. Een workflowmanagementsysteem dat gebaseerd is op deze aanpak biedt ondersteuning tijdens het *ontwikkelen*, *uitvoeren* en de *diagnose* van bedrijfsprocessen.

Ontwikkelen Modellen die op regels gebaseerd zijn, bevatten mogelijk vele regels die elkaar op subtiële wijze beïnvloeden. Dit kan gemakkelijk tot foute modellen leiden, bijvoorbeeld omdat sommige regels elkaar tegenspreken. De grote variëteit aan regels en hun betekenis zorgen ervoor dat het voor mensen lastig is om dergelijke conflicten te ontdekken. Daarom stelt dit proefschrift een methode voor om modellen automatisch te verifiëren tijdens het *ontwikkelen* ervan.

Uitvoeren Tijdens het uitvoeren van processen wordt ondersteuning aan medewerkers geboden door middel van drie mechanismen: het *opleggen van de regels*, het *inzichtelijk maken van de regels*, en *aanbevelingen*.

De aanpak uit dit proefschrift stelt de op regels gebaseerde systemen in staat om de bindende regels *op te leggen*, zodat alle zaken correct afgehandeld worden.

Daarnaast moet, omdat de gebruiker zelf de volgorde van zijn werk kan bepalen, inzichtelijk gemaakt worden wat de toestand van alle regels is tijdens de executie van een proces. Zo kan aan sommige regels al voldaan zijn, terwijl dat voor andere nog niet het geval is. De variëteit en complexiteit van de regels zorgen ervoor dat het vrijwel onmogelijk is voor mensen om dit handmatig te doen. Daarom kan de toestand van iedere regel continu en intuïtief *inzichtelijk* gemaakt worden.

Een gevolg van de grote flexibiliteit van deze aanpak, is dat medewerkers moeite kunnen hebben met het beslissen welke activiteiten in welke volgorde uitgevoerd moeten worden. Dit kan bijvoorbeeld gebeuren bij medewerkers zonder ervaring, of wanneer niet duidelijk is wat de beste actie is in een bepaalde situatie. Daarom biedt deze aanpak de medewerkers de mogelijkheid tot het vragen van *aanbevelingen* tijdens de behandeling van zaken. Deze aanbevelingen worden gegenereerd op basis van ervaringen uit het verleden en kunnen geoptimaliseerd zijn voor een bepaald doel (bijvoorbeeld het minimaliseren van de doorlooptijd van zaken). Het DECLARE prototype gebruikt de “recommendation service” van het “process mining tool” ProM om deze aanbevelingen te leveren. Dergelijke aanbevelingen zijn bedoeld als hulpmiddel om zaken tot een goed einde te brengen, maar worden nooit opgelegd door DECLARE. De gebruiker heeft dus altijd de mogelijkheid om de aanbevelingen naast zich neer te leggen.

Diagnose Nadat een workflowmanagementsysteem een tijd gebruikt is, ligt het voor de hand om te zoeken naar mogelijke verbeteringen. De inherente flexibiliteit van op regels gebaseerde systemen maakt dat er een grote variatie bestaat in het aantal mogelijkheden om zaken af te handelen. Daarom is het des te meer interessant om dergelijke systemen achteraf te analyseren.

Om dit mogelijk te maken, kan het DECLARE prototype logs produceren. Deze logs bevatten informatie over de behandelde zaken en kunnen voor verschillende typen analyse gebruikt worden in ProM. De resultaten van de analyse kunnen gebruikt worden om inzicht te krijgen in het gebruik van het systeem en in mogelijke verbeteringen.

De bovengenoemde, op regels gebaseerde, aanpak en alle principes uit dit proefschrift die daarmee samenhangen zijn geïmplementeerd in het prototype DECLARE. Dit prototype laat zien dat de aanpak inderdaad geïmplementeerd kan worden in een workflowmanagementsysteem. Om dit prototype zo flexibel mogelijk te houden, ondersteunt het verschillende talen waarin regels opgesteld kunnen worden, zoals ConDec, DecSerFlow en CIGDec.

Acknowledgements

First of all, I would like to thank my supervisors prof.dr.ir. Wil van der Aalst and dr. Frans van Eijnatten for giving me the opportunity to work on this project. I am very grateful to them for sharing their knowledge and experience with me during the last four years. I also thank prof.dr. Christel Rutte for supervising me in the first three years of this PhD and making me aware of the multidisciplinary aspects of consequences of my research, namely with respect to the field of organizational psychology and the satisfaction of end-users.

Although only the names of my promotor prof.dr.ir. Wil van der Aalst and copromotor dr. Frans van Eijnatten are mentioned in the first pages of this thesis, more people were involved in the process of writing the manuscript. I thank all members of my examination committee, i.e., prof.dr. Paola Mello, prof.dr. Manfred Reichert, prof.dr. Paul de Bra, prof.dr. Stefan Jablonski, and prof.dr.ing. Thomas Andreas Herrmann, for their comments/suggestions for improving the thesis and the time they invested in the examination.

I would also like to thank all my colleagues from the Information Systems group at the department of Technology Management. They all helped me countless times with solving daily work-related problems. Moreover, I have enjoyed the pleasant social environment in our group. To be honest, I have never seen such a pleasant work environment, and I hope that, in the future, I will have as nice colleagues as they are! I would also like to thank colleagues at the department of Mathematics and Computer Science: Helen Schonenberg for sharing her experience in radiology to help me with the illustrative example and for reading the thesis and giving me useful feedback; and dr.ir. Boudewijn van Dongen for his help with the cover page, formalizations in Chapter 4, and translating the summary to Dutch (i.e., the Samenvatting of this thesis).

In the year before starting my work on this PhD, I did internship in Tilburg, the Netherlands. The internship was a part of an international exchange program, so I met many nice people from all around the world who also came to Tilburg. In particular, I would like to thank two of them: Moise Komlan Egoh for discussions about our mutual dilemma about whether to work in industry or in academia after our internships; and Amar Sahoo for informing me about the possibilities to do a PhD as a foreigner in the Netherlands and many nice hours of tasting special Belgium beers (Amar) and eating peanuts (I) at Kandinsky on Sunday

afternoons. I would also like to thank Milica Kovačević-Milivojević for supplying me with an enormous amount of useful advices about how to search for a PhD position in the Netherlands.

Last, but definitely the most, I thank my family. First, I thank Boris for being a good ‘sport’ in my belly and not making it difficult for me while writing this thesis; handling ‘occasional’ periods of my stress so well; and keeping me company in, what would otherwise be, endless lonely hours of writing the thesis. Second, I thank my mother Radosna for believing in me every single day since 28th of May 1977, no matter what; my father Zoran for calling me almost every day during my first ‘homesick’ year in Tilburg; my brother Djordje for making me laugh so often; and the lovely Boudewijn for showing me the right way whenever I couldn’t see the way out.

Maja Pešić
Eindhoven, August 11, 2008

Curriculum Vitae

Maja Pešić was born on 28th of May 1977 in Belgrade, nowadays Serbia. From 1992 to 1996 she attended high school at the First Belgrade Gymnasium (Prva Beogradska Gimnazija).

After finishing high school, she studied at the Department of Organizational Sciences at the University of Belgrade, where she graduated in 2002 in the Information Systems group with the thesis entitled “Software Development Using Craig Larman’s Method in C++ Environment”.

After graduation, Maja did an internship until 2004 as a software engineer in Smits Info Systems, a software company in Veldhoven, the Netherlands. Her internship was part of the AIESEC¹ international exchange program.

In June 2004 Maja started her doctoral studies in the area of user-centric workflow management systems at the Technology Management department of Eindhoven University of Technology, under the supervision of prof.dr.ir. Wil van der Aalst, prof.dr. Christel Rutte and dr. Frans van Eijnatten. She completed her doctoral studies in October 2008 with the thesis “Constraint-Based Workflow Management Systems: Shifting Control to Users”.

At the moment, she is working as a Postdoc at the Mathematics and Computer Science department of the Eindhoven University of Technology, within the STW project “Controlling Dynamic Real Life Workflow Situations with Demand Driven Workflow Systems”.

¹<http://www.aiesec.org>