

UNIVERSITÀ DEGLI STUDI DI PERUGIA

DOTTORATO DI RICERCA IN

MATEMATICA E INFORMATICA PER LA RAPPRESENTAZIONE E LA
ELABORAZIONE DELL'INFORMAZIONE E DELLA CONOSCENZA – XXIV CICLO

Product and Production Process Modeling and Configuration



Dario Campagna

Relatore

Prof. Andrea Formisano

Coordinatore del corso di dottorato

Prof. Giulianella Coletti

Febbraio 2012

To Jennifer, Rosalia and Gaetano.

Summary

Many companies nowadays offer products in different variants and options, i.e., they deploy the mass customization strategy. Such companies face a series of management difficulties involving different functional areas. An emerging technology that supports companies in deploying the mass customization strategy is represented by *product configuration systems*.

Most of the scientific studies completed on configuration focus on *product configuration*. As mass customization needs to cover the management of the whole customizable product cycle, extending the use of configuration techniques from product configuration to *process configuration* may avoid or reduce planning impossibilities due to product configuration, and configuration impossibilities due to production planning.

In this dissertation we present PRODPROC, a declarative constraint-based framework for defining models of both configurable products and their production processes. Most of the modeling elements of the framework have a graphical representation, thus allowing a user to easily create product and process structures. PRODPROC models result to be executable specifications of products and processes. In fact, Constraint Programming techniques can be exploited to guide the configuration of a product and its production process given the respective PRODPROC model.

Acknowledgements

In this dissertation I report the results of more than three years of work, during which I had the chance to collaborate with different people. I take this opportunity to express my gratitude towards them.

I wish to thank, first and foremost, my supervisor prof. Andrea Formisano. Andrea let me choose my topic freely, and then guided me through my doctoral studies with invaluable discussions and suggestions.

I owe my deepest gratitude to prof. Agostino Dovier, prof. Angelo Montanari, and prof. Carla Piazza. If they had not me involved in a research project after my Master degree, this dissertation would have remain unwritten.

It gives me great pleasure in acknowledging the help of prof. Tom Schrijvers, who supervised me during the months I spent in Belgium at the Katholieke Universiteit Leuven.

Last but not least, I am indebted with my parents, Rosalia and Gaetano, who gave me support throughout my life, and with my girlfriend Jennifer, who has always been by my side, even when I was far away from her.

Contents

Summary	ii
Acknowledgements	iii
1 Introduction	1
1.1 The Thesis	2
1.2 Overview	3
I Preliminary	5
2 Configuration	7
2.1 Mass Customization and Configuration Systems	7
2.2 Product Configuration	9
2.2.1 Rule-based reasoning	10
2.2.2 Case-based reasoning	11
2.2.3 Model-based reasoning	12
2.3 Process Configuration	16
2.3.1 Production Process Configuration	16
2.3.2 Business Process Configuration	17
2.4 Coupling Product with Process Configuration	19
3 Constraint Programming	21
3.1 Constraint Satisfaction Problems	21
3.2 Solving Constraint Satisfaction Problems	23
3.2.1 Search	23
3.2.2 Constraint Propagation	24
3.3 Constraint Logic Programming	27
3.3.1 Modeling and Solving CSPs	27
3.3.2 Constraint Solvers	29
4 Morphos Configuration Engine	33
4.1 Product models and Configuration Process	34
4.1.1 Morphos Product models	34

4.1.2	MCE Configuration Process	36
4.2	Encoding of the Product Configuration Problem	39
4.3	Assignment Revision Process	40
II	The PRODPROC Framework	43
5	Product and Production Process Modeling with PRODPROC	45
5.1	Preliminary Notions	45
5.2	The PROMo Language	47
5.2.1	Product Model Graph	48
5.2.2	Constraints on Product Characteristics	51
5.2.3	PROMo Model	54
5.3	The MART Language	54
5.3.1	Activities and Temporal Constraints	54
5.3.2	MART language elements	58
5.3.3	MART Model and Composite Activities	64
5.4	PRODPROC Models and Instances	66
5.4.1	ProMo Instances	66
5.4.2	MART Instances	73
5.4.3	PRODPROC Instances	76
6	Case Study: Building Construction	79
6.1	Prefabricated Component Building	79
6.2	Building Construction	79
6.2.1	Preparation and Development of the Building Site	80
6.2.2	Building Shell and Building Envelope Works	81
6.2.3	Building Services Equipment and Finishing Works	83
6.3	Building PROMo Model	85
6.3.1	Building Product Model Graph	85
6.3.2	Model Constraints	91
6.4	Building Construction MART Model	91
6.4.1	<i>Preparation and development of the building site</i> MART Model	92
6.4.2	<i>Building shell and building envelope works</i> MART Model	95
6.4.3	<i>Building services equipment</i> MART Model	98
6.4.4	<i>Finishing works</i> MART Model	100
6.5	Coupling of the Models	103
7	CSP Encoding of the Product/Process Configuration Problem	105
7.1	A CSP-based Configuration System	105
7.2	Encoding of the Product Configuration Problem	106
7.3	Encoding of the Process Configuration Problem	110

7.4	Encoding of the Product/Process Configuration Problem	114
8	GCSP Encoding of PRODPROC Models	115
8.1	Generative Constraint Satisfaction Problems	115
8.1.1	Generative CSPs and Configuration Problems	115
8.1.2	Applications of Generative CSPs	116
8.2	Encoding of PRODPROC Models as Generative CSPs	117
8.3	Custom Product Manufacturing and Generative CSPs	125
9	Language Comparisons	127
9.1	PROMo Language	127
9.1.1	ASP-based Systems	127
9.1.2	BDD-based Systems	128
9.1.3	CSP-based Systems	129
9.1.4	Ontologies and UML-based Approaches	129
9.1.5	Feature Diagrams	130
9.1.6	SysML Structural Constructs	131
9.2	MART Language	132
9.2.1	Temporal Constraints	132
9.2.2	Modeling Processes with MART	132
9.2.3	SysML Behavioral Constructs	139
9.3	PRODPROC Framework	139
9.4	Comparison Summary	140
III	PRODPROC Implementation	141
10	Prolog Implementation of PRODPROC Models	143
10.1	Modeling System Architecture	144
10.2	PRODPROC Variable Domains as Types	145
10.3	PRODPROC Constraints	146
10.4	PROMo Modeling Features	147
10.4.1	Product Model Graph	147
10.4.2	Model Constraints	148
10.5	MART Modeling Features	149
10.5.1	MART Model Variables	149
10.5.2	Resource constraints	150
10.5.3	Activities	150
10.5.4	Temporal Constraints	151
10.5.5	Coupling Constraints	152

11 CSP-based Automatic Instance Generation	153
11.1 Generation of Product Instances	155
11.2 Dealing with Resource Constraints	159
11.3 Generation of Process Instances	164
12 ProdProc Modeler	167
12.1 Creation of Product Models	171
12.1.1 Drawing a Product Model Graph	171
12.1.2 Defining Constraints on Product Characteristics	173
12.2 Creation of Process Models	176
12.3 Syntax and Validity Checking	183
13 Conclusions	187
13.1 Summary and Main Contributions	187
13.2 Future Research	188
A Syntax of PRODPROC Languages	191
Bibliography	197

Chapter 1

Introduction

In the past years many companies started to operate according to the *mass customization* strategy. Such strategy aims at selling products that satisfy customer's needs, preserving as much as possible the advantages of *mass production* in terms of efficiency and productivity. The products offered by such companies, usually called *configurable products*, have a predefined basic structure that can be customized by combining a series of available components and options (modules, accessories, etc.) or by specifying suitable parameters (lengths, tensions, etc.). Actually, a configurable product does not correspond to a specific physical object, it identifies sets of (physical) objects that a company can realize. A *configured product* is a single variant of a configurable product, obtained by specifying each of its customizable attributes, which corresponds to a fully-specified physical object.

The mass customization operating mode involves a series of difficulties that companies struggle to resolve by using traditional software tools, designed for repetitive productions. As more companies started offering configurable products, different systems designed for supporting them in deploying the mass customization strategy appeared. These systems are called *product configuration systems* and allow one to effectively and efficiently deal with the configuration process [SW98]. They offer functionality for the representation of configurable products through *product models*, and for organizing and managing the acquisition of information about the product variants to be realized.

Mass customization needs to cover the management of the whole customization product cycle, from customer order to final manufacturing. Current product configuration systems focus only on the support to product configuration, and do not cover aspects related to the production process planning. Extending the use of configuration techniques from products to processes, may avoid or reduce planning impossibilities due to constraints introduced in the product configuration phase, as well as configuration impossibilities due to production planning requirements. Existing languages/tools for process modeling, such as BPMN [WM08] and YAWL [tHvdAAR10], do not offer suitable features for specifying production processes and process configuration. Also, they lack the capability of modeling, in a single uniform setting, product models and their corresponding process models.

1.1 The Thesis

The thesis of this dissertation is that a product modeling framework natively supporting features for modeling process aspects otherwise difficult to describe, may lead to the definition of richer configurable product models, that allow the propagation of consequences of product configuration decision toward the planning of its production process, and the propagation of consequences of process planning decision toward the product configuration.

This dissertation presents a declarative constraint-based framework, called **PRODPROC**, for modeling configurable products (whose producible variants can be represented as trees), and their production processes. In particular, it extends the first results on such a framework we reported in [Cam11] and [CF11]. **PRODPROC** is inspired by the works of Aldanondo et al. (see, e.g., [AV08]) on the coupling of product configuration and process configuration, and reuses knowledge from the Morphos MCE system [CRD⁺10]. The proposed framework natively supports features for modeling temporal relations between process activities, and activity resource production and consumption. Moreover, it defines a graphical representation for most of its modeling elements, and allows one to easily define mixed product/process models without the need to know how the configuration systems works “under the hood”.

In order to show the effectiveness of the framework presented in this dissertation, a configuration system has been implemented on top of **PRODPROC**. The system, called **PRODPROC Modeler**, allows one to define mixed product/process models using the **PRODPROC** framework, and uses **PRODPROC** models as executable specifications to automatically generate product and process instances through an algorithm exploiting Constraint Programming techniques.

Contributions

The central contributions of this dissertation can be summarized as follows.

- This dissertation presents **PRODPROC**, a declarative constraint-based framework for modeling products and their production processes. The framework allows one to define product and process models, and to couple a product model with a process model. Graphical representations of modeling elements allows a user to easily draw products and processes structures.
- A **PRODPROC** semantics in terms of model instances is given. A Constraint Satisfaction Problem encoding of the product/process configuration problem, and a Generative Constraint Satisfaction Problem encoding of **PRODPROC** models are proposed.
- The **PRODPROC** framework is compared with existing product configuration systems and process modeling tool, to point out its strength and limitation.
- A configuration system implemented on top of **PRODPROC** and based on Constraint Logic Programming is described.

1.2 Overview

The content of this dissertation is subdivided in three part. The first part introduces preliminary notions used throughout the dissertation. The second part presents the PRODPROC graphical framework. The third and last part describes PRODPROC Modeler.

The first part comprises Chapter 2 to Chapter 4. Chapter 2 introduces the notion of *mass customization* and summarizes the state of the art of research on *product configuration* and *process configuration*. Chapter 3 recapitulates the fundamental ideas behind *constraint programming*. Chapter 4 presents the Morphos MCE system, a product configuration system based on Constraint Logic Programming.

Chapter 5 to Chapter 9 make up the second part of the dissertation. Chapter 5 presents the PRODPROC framework, and gives it a semantics in terms of model instances. A building case study is exploited in Chapter 6 to show a complete example of mixed product/process model. Chapter 7 describes a Constraint Satisfaction Problem encoding of the product/process configuration problem, while Chapter 8 shows a possible encoding of PRODPROC models into Generative Constraint Satisfaction Problems. A comparison of PRODPROC with existing systems and tools for product configuration and process modeling is given in Chapter 9. The syntax of PRODPROC languages is given in Appendix A.

The third part consists of Chapter 10 to Chapter 12. A Prolog encoding of PRODPROC models is defined in Chapter 10. Chapter 11 describes an algorithm for the automatic generation of PRODPROC instances. A detailed description of PRODPROC Modeler interface is given in Chapter 12.

Chapter 13 concludes this dissertation with a discussion of the presented material and an outlook on future work.

Part I

Preliminary

Chapter 2

Configuration

Many companies nowadays offer products in different variants and options, i.e., they deploy the mass customization strategy. Such companies face a series of management difficulties involving different functional areas. Software configurators are an emerging technology that supports companies in adopting the mass customization strategy.

In this chapter, we first introduce mass customization and configuration systems (cf. Section 2.1). Then, we summarize the state of the art of research about methodologies and techniques for product and process configuration (cf. Section 2.2 and Section 2.3). Finally, we focus on the coupling of product configuration with process configuration (cf. Section 2.4).

2.1 Mass Customization and Configuration Systems

During 1980s and 1990s, the increasing competition among companies lead manufacturers to focus on incrementing product and production process efficiency and quality. In particular, companies invested on the optimization of internal activity efficiency, adopting also Information Technology instruments. Although this objective is still of importance for manufacturers, operation efficiency alone no more guarantees a sufficient competitive advantage, durable over time.

Nowadays, the competitive situation of companies is characterized by a very strong orientation towards product individualization. The industrial goods' markets reached a saturation situation, where the offer exceeds the demand, due to a change from a seller to a buyer market. The customer's power increased after this major change, and companies started to differentiate their products from those of competitors by offering individualized problem solutions.

The main cause of this individualization trend is represented by social changes. One century ago, a key factor for the emergence of the *mass production* system was the high grow of population. But nowadays, especially in the industrial nations, the demographic development shows the population to be steadily decreasing, while wealth and the demand for luxury continue to increase. The need for change and novelty is becoming as important as survival for human beings. Moreover, the possession of an object is no longer interesting

and loses attractiveness if more and more people possess it. All of these reasons have contributed to a need for individualization and the demand of products that exactly meet the individual expectations of customers [BF07].

The ability of fulfilling individual customer needs requires the capability of producing a large number of product variants, which induces high costs at both operations- and manufacturing-related tasks. In contrast to the mass production system, in which the economies of scale can be fully utilized, the individualization of customer requirements usually involves a loss of efficiency. The challenge that manufacturing companies have to face is to provide individualized products and services while maintaining a high cost efficiency. To be successful, companies have to address both these perspective, which are necessary for gaining a competitive advantage [BF07]. The manufacturing of products according to individual customer needs is referred to as product customization. Product customization and cost efficiency are the main ingredient of *mass customization*, a new business paradigm that is very challenging for manufacturing companies.

Mass customization is a business strategy that aims at fulfilling individual customer needs with near mass production efficiency [PD99]. Companies that want to adopt this strategy need a set of practical tools in order to make mass customization work efficiently. The main challenge is about how to be able to produce a large number of customer-oriented product variants by simultaneously providing prices that do not considerably differ from those of mass products.

Products subject to customization are products with a predefined basic structure that can be customized by combining together a series of available components and options (modules, accessories, etc.) or by specifying suitable parameters (lengths, tensions, etc.). The class of “products subject to configuration” includes products of different types like, for example, computers, cars, industrial machineries, shoes, etc. With the term *configurable product* one usually refers to a type of product offered by a company. Hence, it does not correspond to a specific (physical) object, but it identifies a set of (physical) objects that the company can realize. A *configured product* is a single variant of the configurable product, which corresponds to a fully-specified (physical) object. A configured product is obtained by customizing a configurable product, that is, by specifying the value of each customizable attribute of the configurable product. The *configuration process* consists of a series of activities and operations ranging from the acquisition of information about the specific variant of the product requested by the customer to the generation of data for its realization. In general, it requires a complex interaction with the customer to find, among the available characteristics and functions, those that best meet his/her needs. Such an interaction can result in combinations that have never been realized before.

Customers generally accept paying premium prices for customized products compared to standard products because they honor the additional benefits they can get. Therefore, if mass customization fails in providing customers with an optimal or a better solution than any mass products, then the product resulting from the configuration process will have, from the customer’s perspective, no more additional value than any other standard product. As a result, an optimal understanding of customer needs is a necessary requirement for the success of the strategy [BF07]. Moreover, during the pursuit of mass customization, customers have to be seen as partners in the value creation, which implies a deeper

customer-supplier relationship.

Since it is necessary to satisfy the customer, the only chance to meet this challenge is to reduce the customizing costs during the configuration process. Researches conducted on mass customization put in evidence that companies can exploit different instruments to achieve this objective. However, these researches also pointed out that traditional solutions adopted for the acquisition of customizable product orders are not, in general, satisfactory. In particular, information systems developed for mass production shown limitations on the management of large number of product variants. The universal remedy for reducing the costs of product customization is to design, implement, and use a supporting computerized *configuration system*. Such a system is, once implemented, the best way to cope with the problem, because it automates main parts of product designing and producing. This reduces complexity and human efforts, which in the end lead to lower costs. Furthermore, the advances realized in Information Technology are critical enablers, which make this strategy function efficiently. Information systems can be implemented to support diverse activities in the mass-customization value chain. They assist customers during the product specification phase in order to lead them in a fast-paced manner to the product variants corresponding to their individual requirements. Modern information systems, which support open innovation, even enable customers to participate actively in the product design. In addition, mass customization information systems contribute to helping companies mitigate excessive product variety and increase cost efficiency on the shop floor and logistics through optimal product modeling, production planning, and scheduling [BF07].

2.2 Product Configuration

Informally, product configuration is a special case of design activity with two key features: (i) the product being configured is assembled from instances of a fixed set of well defined component types; (ii) components interact with each other in predefined ways. Selecting and arranging combinations of parts that satisfy given specifications form the core of a configuration process. The solution must produce the list of selected components, as well as the configured product's structure and topology [SW98].

Systems supporting the configuration of customizable products are usually referred to as *product configuration systems* (or *product configurators*). As early as 1982, Digital Equipment Corporation used the R1/XCON system [McD82] to configure computer systems. Since then, a wealth of configuration expert systems have been built for configuring computers, communication networks, cars, trucks, operating systems, buildings, circuit boards, keyboards, printing presses, and so forth [SW98]. Like any problem-solving activity, solving a configuration task implies two different steps. First, a representation of the problem has to be devised. Second, algorithms that, based on the problem representation, will produce a solution have to be implemented. The core of a product configuration system is the *product model*. It is a logical structure that formally represents the features of the types of product offered by a company. Moreover, it specifies a number of constraints on the relationships among these features. The main modules of a product configurator are the *modeler* and the *configuration engine*. The modeler supports the modeling process.

It allows one to create and to modify product models by defining features and constraints of configurable products. The configuration engine implements algorithm supporting the configuration process. It facilitates the work of the seller, helping him/her in organizing and managing the acquisition of information about the product variant to be realized. In particular, it makes it possible to immediately check the validity and consistency of inserted data.

The problem of product configuration has been addressed in a number of different ways. According to [Jun06], three main approaches can be identified: rule-based reasoning [McD82], case-based reasoning, and model-based reasoning [MF89].

2.2.1 Rule-based reasoning

Early configuration applications were based on rule-based configuration methods. Following Digital's landmark RI/XCON system, the history of configuration systems has continued with numerous systems developed by other organizations. These systems, also known as expert systems, use *production rules* as a uniform mechanism for representing both domain knowledge and control strategy [SW98].

XCON uses OPS5, a production-rule programming language. Two key issues in configuration are: (i) what actions must be performed to obtain a valid configuration; (ii) when an action can appropriately occur in relation to other actions. The production-rule programming paradigms explicitly provides the dynamic, runtime decision-making essential for copying with these characteristics.

The fundamental activity of any program written in OPS5 is the execution of the *recognize/act* cycle [BS89]. A production rule has the form **if** *condition* **then** *consequence*. A working memory holds global state information, i.e., inputs and results from executing actions. The conditional part of the rule specifies tests on the working memory. If the condition is satisfied, then the system executes the rule's consequence, possibly modifying the working memory. The system derives solutions in a forward-chaining manner. At each step, the system examines the entire set of rules and considers only the rules it can execute next. Each rule carries its own complete triggering context, which identifies its scope of applicability. The system then selects and executes one of the rules under consideration by performing its action part.

Although adherents of rule-based reasoning initially claimed that it allows incremental development, it soon became apparent that large rule-based systems incur enormous maintenance problems. Rules specify both *directed relationships* and *actions*. A directed relationship represents domain knowledge (compatibilities, dependencies, etc.), while an action represents (procedural) knowledge controlling the computation of a solution. This lack of separation between domain knowledge and control strategy, and the spread of knowledge about a single entity over several rules make the knowledge-maintenance task extremely difficult. XCON is a good illustration of the complexity of knowledge maintenance. In 1979, XCON had 100 components and 250 rules. In 1989, its knowledge base grew to more than 31.000 components and approximately 17.500 rules [BOBS89]. The change rate of this knowledge averaged about 40% per year. At the same time, the number of XCON developers increased from 2 to 59. To be able to manage the system, a

new analyst or programmer had to spend a learning period of at least 12 months. As a matter of fact, rule-based systems can be successfully exploited to deal with very simple configuration problems involving a small number of choices only.

2.2.2 Case-based reasoning

A major part of human expertise is based on past experiences. Case-based configuration provides a model for representing the experiences and using them for problem solving. Case-based reasoning (CBR) [AP94] is a problem solving paradigm proposed in Artificial Intelligence (AI) that differs from standard AI paradigms in many respects. Rather than relying on general knowledge about a problem domain or establishing associations between problem descriptions and solutions by means of suitable general relationships, CBR takes advantage of specific knowledge about concrete problem instances it already dealt with in the past (*cases*). A new problem instance is addressed and possibly solved by finding a similar past case and reusing the machinery that has been used to solve it. Moreover, additional knowledge provided by the solution of a new instance of the problem becomes immediately available for the management of future problem instances.

The CBR paradigm covers a range of different methods for organizing, retrieving, utilizing and indexing the knowledge retained in past cases. Cases may be kept as concrete experiences, or a set of similar cases may form a generalized case. Cases may be stored as separate knowledge units, or split up into subunits and distributed within the knowledge structure. Cases may be indexed by a prefixed or open vocabulary, and within a flat or hierarchical index structure. The solution from a previous case may be directly applied to the present problem, or modified according to differences between the two cases. The matching of cases, adaptation of solutions, and learning from an experience may be guided and supported by a deep model of general domain knowledge, by more shallow and compiled knowledge, or be based on an apparent, syntactic similarity only. CBR methods may be purely self-contained and automatic, or they may interact heavily with the user for support and guidance of its choices. Some CBR methods assume a rather large amount of widely distributed cases in their case base, while others are based on a more limited set of typical ones. Past cases may be retrieved and evaluated sequentially or in parallel.

Actually, “case-based reasoning” is just one of a set of terms used to refer to systems of this kind. This has led to some confusion, particularly since case-based reasoning is a term used both as a generic term for several types of more specific approaches, as well as for one such approach. Case-based approaches can be classified in five different classes: *exemplar-based reasoning*, *instance-based reasoning*, *memory-based reasoning*, *case-based reasoning*, *analogy-based reasoning*. CBR methods that address the learning of concept definitions (i.e. the problem addressed by most of the research in machine learning), are sometimes referred to as *exemplar-based*. *Instance-based reasoning* is a specialization of exemplar-based reasoning into a highly *syntactic* CBR-approach. The *memory-based* approach emphasizes a collection of cases as a large memory, and reasoning as a process of accessing and searching in this memory. Although case-based reasoning is used here as a generic term, the typical *case-based reasoning* methods have some characteristics that distinguish them from the other approaches listed in this section. First, a typical case

is usually assumed to have a certain degree of richness of information contained in it, and a certain *complexity* with respect to its internal organization. Typical case-based methods also have another characteristic property: they are able to *modify*, or adapt, a retrieved solution when applied in a different problem solving context. Paradigmatic case-based methods also utilize *general background knowledge* - although its richness, degree of explicit representation, and role within the CBR processes varies. The term *analogy-based reasoning* is sometimes used, as a synonym to case-based reasoning, to describe the typical case-based approach just described. However, it is also often used to characterize methods that solve new problems based on past cases from a *different domain*, while typical case-based methods focus on indexing and matching strategies for single-domain cases.

The application of case-based reasoning to the configuration problem has been explored in various contributions. An interesting example can be found in [TCC05]. The main weaknesses of such an approach are connected to the lack of a complete model of the application domain, the complexity of the knowledge acquisition problem, the low quality of the proposed solutions, and various inefficiencies at execution time. To overcome these weaknesses, hybrid approaches combining case-based reasoning with other reasoning methods (e.g., rule-based reasoning, constraint satisfaction techniques) have been proposed.

2.2.3 Model-based reasoning

Model-based reasoning approaches address the deficiencies of rule-based and case-based reasoning ones. Firstly, they separate the description of the problem from the algorithm that solves it, thus allowing an analysis of the problem which actually does not depend on the chosen approach. Secondly, the description of the problem is based on a model of the system to be configured. Such a model consists of a set of decomposable entities and a specification of the interactions between them. Modularity and compositionality are well-supported since components can easily be added or removed without changing the whole model. Thirdly, they require the description of the problem to be complete by defining a closed space of possible configurations. If no element of this space satisfies the given requirements, then the problem has no solution.

Different proposals can be situated in the setting of model-based reasoning, including approaches based on description logic [MW98], Constraint Programming [MF90, SFH98], and resource models [JH98]. Additional solutions have been recently proposed based on SAT [SKK03], Binary Decision Diagrams [HSJ⁺04], integer programming [TO02], and answer-set programming [SNS02]. Finally, there have been various attempts to combine different solutions in order to enhance the modeling and solving capabilities of model-based reasoning configurators. In the following, we present some of the most interesting existing systems respectively based on description logic, answer-set programming, Binary Decision Diagrams, and Constraint Programming.

Description logic. One prominent family of logic-based approaches to configuration is based on *description logic* (DL) [BCM⁺03]. Description logic is not monolithic, the DL community has implemented a great assortment of description logics and extensions, each one best suited to a specific type of applications. DLs are formalisms for representing

and reasoning with knowledge. They unify and provide a logical basis for the well-known traditions of name-based systems, semantic networks, object-oriented representations, semantic data models, and type systems. DLs have three fundamental elements: *individuals*, representing objects in the application domain; *concepts*, representing sets of individuals; *roles*, binary relations between individuals.

DL systems reason about the intensional description of concepts and their instances (individuals). Constructors - for example, *and*, *or*, *at-least*, *all*, etc. - allow the creation of complex, composite descriptions. The main inference mechanism in DLs is *subsumption*, i.e., the decision whether one concept (description) is more general than another. The semantics of subsumption is defined by the subset relationship between two concepts as sets of individuals. Most other forms of inference performed in DL systems can be expressed using subsumption. Examples are *classification* and *recognition*. Classification is the process of integrating a new concept into the concept hierarchy, and recognition determines whether an individual instantiates a particular concept.

Description logics are well-suited to describe product component types and their relations. They organize component types in a taxonomy and allow one to define complex types out of primitive ones. Moreover, DLs are able to express specialization relations among complex types and to test type consistency by classification. Classification can actually be exploited to solve configuration problems provided that knowledge about configuration can be completely expressed in a DL. Unfortunately, in many cases the expressiveness of a DL is not sufficient and a rule-based or constraint-based engine must be used to cope with complex compatibility and numerical constraints. A significant example of such a hybrid approach is given by the CLASSICS project [MW98].

Answer-set programming. *Answer-set programming* [SNS02] is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. Answer-set programming (ASP) is based on the stable model (answer set) semantics of logic programming [GL88], which applies ideas of autoepistemic logic and default logic to the analysis of negation as failure. In ASP, search problems are reduced to computing stable models, and answer set solvers - programs for generating stable models - are used to perform search. The search algorithms used in the design of many answer set solvers are enhancements of the Davis-Putnam-Logemann-Loveland procedure, and they are somewhat similar to the algorithms used in efficient SAT solvers.

ASP expresses configuration knowledge by means of suitable default rules, while ensuring well-justified configurations through groundedness conditions [SGN99]. A significant representative of the ASP approach to configuration is Kumbang Configurator [MAMS05], a tool supporting the configuration task for configurable software product families. In Kumbang Configurator, a configurable software product family is modeled from two independent, yet mutually related, points of view. *Features* are abstractions from the set of requirements of the product family. A feature may include a number of sub-features as its constituent elements and it can be characterized by a number of attributes. The technical aspects of the product family are captured by its architecture, which is defined in terms of *components*. A component may have a number of other components as its parts and it

can be characterized by a number of attributes. Furthermore, the interaction possibilities for a component are specified by its interfaces and bindings between such interfaces. In addition, Kumbang Configurator supports the user in the configuration task by providing a graphical user interface through which the user can enter his/her specific requirements for a product individual. Moreover, it checks the configuration for consistency and completeness after each step and it derives the consequences of the selections made so far. The derivation process is implemented using *smodels* [SNS02], a general-purpose inference tool based on the stable model semantics for logic programs.

In general, ASP-based configuration systems provide a number of features that are well suited for the modeling of software product families. On the one hand, this makes these systems appealing for a relevant set of application domains. On the other hand, it results in a lack of generality, which is probably the major drawback of this class of systems.

Binary Decision Diagrams. A *Binary Decision Diagram* (BDD) [Bry86, Bry92] is a rooted directed acyclic graph representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0 and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges, *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is *true*, if the label of the reached terminal node is 1, otherwise it is *false*. A BDD is *ordered* if all paths in the graph respect a given ordering of the variables. A BDD is *reduced* if no pair of distinct nodes u and v are associated with the same variable and low and high successors, and no variable node u has identical low and high successors. Due to these reductions, the number of nodes in a BDD for many functions encountered in practice is much smaller than the number of truth assignments of the function.

BDDs have been successfully exploited to support configuration systems for interactive sales. These systems have to be able to simultaneously solve the configuration problem for a number of customers, guaranteeing short response times. As the considered problem instances only differ in the users' requirements, and not in the product model, preprocessing techniques can be used to compute configurations in advance. For problems involving a fixed number of parts with small finite domains, it may be possible to represent the whole configuration space compactly by a BDD [HSJ⁺04]. A well-known representative of this class of configuration systems is the commercial system Configit Product Modeler [Con09]. This system implements a two-phase approach to interactive product configuration. In the first phase, the product model is compiled into a reduced ordered BDD representing the set of valid configurations (the solution space). In the second phase, a fast specialized BDD operation is used to prune variable domains. The worst-case response time only grows polynomially with the size of the BDD. Since the size of the compiled BDD is small, the computationally hard part of the configuration problem is confined to the offline phase.

Constraint Programming. *Constraint Programming* [RBW06] is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from

artificial intelligence, computer science, databases, programming languages, and operations research. Constraint Programming (CP) is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics. The basic idea in Constraint Programming is that the user states the constraints and a general purpose constraint solver is used to solve them. Constraints are just relations, and a *Constraint Satisfaction Problem* (CSP) states which relations should hold among the given decision variables. For example, in scheduling activities in a company, the decision variables might be the starting times and the durations of the activities and the resources needed to perform them, and the constraints might be on the availability of the resources and on their use for a limited number of activities at a time. Constraint solvers take a real-world problem like this, represented in terms of decision variables and constraints, and find an assignment to all the variables that satisfies the constraints.

The adequacy of CP as a tool for product configuration is witnessed by a significant number of contributions (see, for instance, [FM87, MF87]). A configuration problem can be naturally formulated as a CSP whose solutions represent the possible configured products. An enlightening example of such an encoding can be found in [FLM⁺03], where the authors take advantage of the classical problem of the N queens to illustrate the distinctive features of a product configurator based on CP with preferences. Two representative configuration systems based on CP are ILOG Configurator [Jun03] and the Lava system [FFH⁺98].

A fundamental issue in CP-based configuration systems is that of computational efficiency. Backtrack-free configuration algorithms for CSP-based systems are often inefficient, while non backtrack-free ones need to explicitly deal with dead ends. Various techniques and heuristics have been proposed in the literature to improve the efficiency of constraint solving in the configuration process. Extensions to the basic CP paradigm have been investigated to support advanced desirable features (of configuration systems), such as the ability to organize system components in a hierarchy, the availability of tools that provide explanations on system behaviors, and the presence of interactive methods to express preferences and modify choices already made. To support at least some of these features, a number of variants of the standard CSP model have been developed. An extension of CSP, called *Dynamic CSP*, that makes it possible to manage optional components during the modeling phase in an effective way, has been proposed in [MF90]. Another variant of CSP, close to Dynamic CSP, called *Generative CSP*, has been developed in [FFH⁺98]. The handling of CSP at various levels of abstraction, called *Composite CSP*, has been illustrated in [SF96]. A configurator that uses CSP and soft constraints to implement an iterative approach to the configuration problem has been described in [FLM⁺03]. Finally, the integration of the CSP technology with that of *Truth Maintenance Systems* [Mar91], which results in the so-called *assumption-based CSP*, has been explored in [AFM02] (as a matter of fact, such a paper contains an interesting discussions of advantages and limitations of configurators based on CSPs).

2.3 Process Configuration

The term *process configuration* may refer to both the management of processes for customizable product manufacturing, and the problem of managing families of business processes. In this section, we summarize the state of the art of research about production process configuration (cf. Section 2.3.1) and business process configuration (cf. Section 2.3.2).

2.3.1 Production Process Configuration

The direct consequence of product customization on production is evident in an exponential increase in the number of process variations, including diverse machines, tools, fixtures, setups, cycle times, labors, and so on [WMT96]. Configuring production processes for product families, referred to as process configuration, by exploiting similarities among variants, has been recognized as an effective means to enable product customization while maintaining near mass production efficiency [Sch01].

Process variety is manifested in a set of generic items, which essentially are the items of the same type, and a set of variants involving changes of operations sequences (namely, process structural changes), and their relationships. Process variety introduces significant constraints to production planning and control, e.g., preventing make-to-order systems from building customization capabilities [Kol00].

A large amount of effort on product design and platform-based product development has been made over the last decade [JSS07]. While the majority of the researches focused on product design, some effort has been reported in the process planning phase. Schierholt presented the novel concept of process configuration where product configuration methods were utilized to support the process-planning task. Consequently, a process configuration system including an interactive and automatic process configuration component has been developed [Sch01]. Nielsen and Kimura presented a resource capability model for automotive product family analysis to consider resource reuse during generic and detailed process planning [NK06]. Azab et al. developed a semi-generative macro process planning system for reconfigurable manufacturing. Precedence graphs depicting the relationships between features/operations are reconfigured by adding and removal of nodes [APEU07]. Kulvatunyou et al. presented an integration framework for process planning to facilitate collaborative manufacturing. Hierarchical process modeling, process planning decomposition and an augmented AND/OR directed graph are used in their proposed Integrated Product and Process Data (IPPD) representation called a Resource Independent Operation Summary (RIOS) [KWCJ04]. Martinez et al. presented a new method for assembly process planning for product families with multiple operations and variants, based on the concept of meta-product and meta-sequence. The proposed meta-sequence is represented by a hyper-graph and is able to simplify assembly plan generation [MFG00]. Bley and Zenner proposed a novel approach toward an integrated consideration of all required product and process variants. In order to consider process variants in assembly planning, a decision element is integrated into the process graph. The implementation of so-called variant junctions allows several variants to be considered within the planning of the required assembly processes [BZ06]. Jiao et al. developed a formal modeling of process variety using

Petri nets. Object-oriented Petri nets (PNs) with changeable structures have been applied to deal with the issues of generic representation, constraints compliance, and operational sequence requirements. Such object-oriented PN (OPN) models facilitate generic representation of product and process variety as well as their instantiations [JZP04]. Alizon et al. proposed an approach for searching and reusing existing design and manufacturing knowledge. In addition to the efficiency assessment for knowledge retrieval, the method accounts for the variety of the product family during the reuse of existing process planning information [ASS06]. Zheng et al. presented a systematic knowledge model for manufacturing process cases. The model represents process knowledge at different levels of granularity and facilitates process configuration. Based on the systematic knowledge model, Zheng et al. also proposed an approach to achieve rapid process configuration by reusing the process knowledge and by performing the involved configuration rules so as to create an appropriate, reliable and effective process plan [ZDV⁺08].

2.3.2 Business Process Configuration

Each product that a company provides to the market is the outcome of a number of activities performed. *Business processes* are the key instrument to organizing these activities and to improving the understanding of their interrelationships. A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations [Wes07].

Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes. The basis of business process management is the explicit representation of business processes with their activities and the execution constraints between them. Once business processes are defined, they can be subject to analysis, improvement, and enactment [Wes07].

Traditionally, business processes are enacted manually, guided by the knowledge of the company's personnel and assisted by the organizational regulations and procedures that are installed. Enterprises can achieve additional benefits if they use software systems for coordinating the activities involved in business processes. These software systems are called *business process management systems* [Wes07]. During the last years, there has been an increase in the adoption of business process management (BPM) tools by enterprises as well as emerging standards for business process specification and execution (e.g., BPMN [WM08]). A *business process model* captures the activities an organization has to perform to achieve a particular business goal. Thus, it implements a process type (e.g., handling of a credit request or medical treatment of a patient) by describing its activities as well as their execution constraints (i.e., control flow), resources required (e.g., humans or computer systems), and information processed [HBR10]. For creating and maintaining such business process models, there exists a multitude of tools, supporting different modeling techniques including UML Activity Diagrams [RJB98], Business Process Modeling Notation (BPMN), and event-process chains (EPCs) [vdA99].

When creating business process models, one of the fundamental challenges the process designers face is to cope with business process variability. Typically, for a particular process type, a multitude of *process variants* exists, each of them being valid in a particular context, i.e., the configuration of a particular process variant depends on concrete requirements building the *process context* [HBR10]. Process models in commercial use lack a representation of variation points and configuration decisions. In other words, they focus on the common traits found in recurrent business processes, but they do not capture possible variations in a systematic manner. As a result, analysts are given little guidance as to which model elements need to be removed, added or modified to meet a given set of requirements, a practice known as *individualization* [LDtHM11].

The concept of *configurable process model* has been put forward as a means to address this limitation [RvdA07]. A configurable process model is a model that captures multiple variants of a business process in a consolidated manner. Given a configurable process model, analysts are able to define a *configuration* of this model by assigning values to its variation points based on a set of requirements. Once a configuration is defined, the model can be individualized automatically, thereby relieving analysts of this tedious and error-prone step [LDtHM11]. The control-flow perspective of process models is commonly captured in terms of activities and events related by control-flow arcs and connectors. Such concepts can be found in virtually any process modeling notation. The resource perspective, on the other hand, is commonly modeled in terms of associations between activities and roles, where a role represents a set of capabilities and/or an organizational group. The flow of data and physical artifacts in a business process is generally captured by associating objects with activities. In addition to task-role associations, resource modeling in business processes also encompasses resource allocation and role-based access control.

According to [LDtHM11], existing methods for capturing variability in process models can be classified into four categories, based on their underlying variability mechanism: *configurable nodes*, *model projection*, *annotation*, and *hiding and blocking of elements*. *Configurable nodes* are used as a configuration mechanism in Configurable EPCs [RvdA07]. The key idea is that configuration options are associated with particular types of process model elements. *Model projections* are introduced as a variability mechanism in [BDK07]. Since a reference process model typically contains information on multiple application scenarios, it is possible to create a projection for a specific scenario (e.g. a particular class of users) by fading out those process branches that are not relevant to the scenario in question. The configuration is conducted by setting configuration parameters defined in the form of simple attributes or logical terms over characteristics. These can be elements of a model, but also elements of the meta-model. In this way, whole element types can be excluded. Those elements whose parameters evaluate to false are hidden. Different approaches have been defined to achieve configuration by means of *annotations*. The PESOA (Process Family Engineering in Service-Oriented Applications) project [SP06] defines so-called variant-rich process models as process models extended with stereotype annotations to accommodate variability. These stereotypes are applied to both UML Activity Diagrams and BPMN models. In [GAJV07] the authors investigate a set of process configuration operators based on *hiding* (skipping) and *blocking of model elements*. These operators are applied to Labeled Transition Systems (LTSs). LTSs are a formal abstraction of computing

processes, therefore any process model with a formal semantics (e.g. Petri nets [RR98] or YAWL [tHvdAAR10]) can be mapped to an LTS. The blocking operator corresponds to disabling the execution of an atomic action. In the LTS this means that a blocked edge cannot be taken anymore. Meanwhile, hiding corresponds to abstraction, i.e. the execution of an atomic action becomes unobservable. In the LTS a hidden edge is simply skipped, but the corresponding path is still taken. This approach has been subsequently applied to define a configurable extension of YAWL, namely C-YAWL [LaR09].

Only two of these approaches take aspects beyond the configuration of control-flow into consideration. The approach by Becker et al. [BDK07] can be used to hide elements and element types in extended EPCs for configuration purposes, which includes non-control flow elements. However, this only affects the view on the EPC, not its underlying behavior. Also, this approach does not enable fine-grained configuration of task-role and task-object associations (beyond hiding). A second approach by Razavian et al. [RK08], based on the PESOA project, is restricted to the configuration of simple forms of role-task and object-task associations that are present in UML Activity Diagrams. As a result, both offer only basic features to configure resources and objects, such as removing a role or an object. In [LDtHM11] the authors addressed the lack of support for the resource and object perspectives of existing configurable process modeling notations. They proposed a notation for configurable process modeling, namely C-iEPCs, that extends the EPC notation with mechanisms for representing a range of variations along multiple perspectives. While embodied in the EPC notation, the proposed extensions are defined in an abstract manner, so that they can be transposed to other notations, such as BPMN and UML Activity Diagrams.

2.4 Coupling Product with Process Configuration

Mass customization needs to cover the management of the whole customization product cycle, from customer order to final manufacturing. On the one hand, product configuration decisions may have strong consequences on the planning of its production process. On the other hand, planning decisions can provide hard constraints to product configuration. Dealing with product and production process in an independent way can cause inconsistencies, while managing them in a sequential way can cause additional delays. To overcome these problems, Aldanondo et al. proposed to couple together interactive product configuration tools with process planning tools in order to pass decisions made from one to the other [AVDG08, AV08]. In particular, they proposed to associate product configuration and production planning in order to allow: (i) the propagation of the consequences of each product configuration decision toward the planning of its production process; (ii) the propagation of the consequences of each process planning decision towards the product configuration. This should reduce or avoid planning impossibilities due to product configuration, and configuration impossibilities due to production planning.

In [AVDG08], configuration and planning problems are considered simultaneously as two Constraint Satisfaction Problems. In order to propagate decision consequences between the two problems, the two constraint based models are linked by means of constraints.

Configurable products are modeled in [AVDG08] as set of components. Each component has associated a set of properties representing its configurable characteristics. Moreover, a set of constraints restricts possible combinations of components and property values. Dynamic CSPs [MF90] are used to implement product models. Each group of components and product property is associated with a variable. Each component and each property value corresponds with one value of the variable. The constraints represents the allowed or excluded combinations of components and property values. Dynamic CSPs extend CSPs by introducing initial variables, i.e., variables that exists in any configured product, and activity constraints, i.e., constraints allowing to control variable existence. Dynamic CSPs allows to modulate the existence of any variable corresponding with a group of components or a product property.

Production planning is addressed in [AVDG08] considering a production process as a set of task entities. A task entity is defined with: temporal parameters (start time, finishing time, and duration), resource parameters (required resource, and quantity of required resource), compatibility constraints (linking duration with required resource and/or required resource quantity). Temporal parameters are numerically defined with intervals, while resource parameters can remain symbolic since infinite resource capacity is considered. The production process is defined with: a set of tasks, a set of precedence constraints between tasks expressing that task Y is after task X (these kinds of constraints have certain similarities to Allen’s primitives [All83]). The production process can gather sequential tasks and parallel tasks with “AND” and “OR” nodes. Planning decisions can correspond with: (i) temporal parameters, value selection or domain restriction; (ii) resource and/or resource quantity selection or domain restriction. Numeric CSPs [Lho93] are exploited in order to implement process models.

Coupling constraints [AVDG08] can be used to link a product with a process model. A coupling constraint is a compatibility constraint that links a variable of the configuration model with a variable of the planning model. Any variable of the configuration model can belong to a coupling a constraint. On the planning model side, resource parameters and duration variables can be involved in coupling constraints. A resource parameter in a coupling constraint allows to propagate the impact of a configuration decision on the selection of the required resource and/or resource quantity (reverse behavior from resource selection to product configuration is also possible). A temporal parameter duration in a coupling constraint allows to propagate the impact of a configuration decision on the modulation of the duration of a task (reverse behavior from duration modulation to product configuration is also possible).

The system proposed by Aldanondo et al. in [AVDG08] shows that it is possible to manage interactions between product configuration and production planning. However, it has to be considered as a primary result on the study of coupling process with product configuration, that need to be consolidated with structured (or multi-level) planning, finite resource capacity planning, and less routine design.

Chapter 3

Constraint Programming

Constraint Programming is an alternative approach to programming which relies on a combination of techniques that deal with *reasoning* and *computing* [Apt03]. It has been successfully applied in a number of fields including molecular biology, electrical engineering, operations research and numerical analysis.

In Section 3.1 we introduce the notion of Constraint Satisfaction Problem (CSP). Section 3.2 presents inference and search techniques for CSP solving. Finally, Section 3.3 is devoted to the presentation of Constraint Logic Programming.

3.1 Constraint Satisfaction Problems

The central notion in Constraint Programming is that of a constraint. Informally, a *constraint* on a sequence of variables is a relation on their domains. It can be viewed as a requirement that states which combinations of values from the variable domains are admitted. In turn, a *Constraint Satisfaction Problem* consists of a finite set of constraints, each on a subsequence of a given set of variables. The classic definition of a Constraint Satisfaction Problem (CSP) is the following.

Definition 1. *Constraint Satisfaction Problem (CSP)* A CSP \mathcal{P} is a triple $\mathcal{P} = \langle X, D, C \rangle$ where X is an n -tuple of variables $X = \langle x_1, x_2, \dots, x_n \rangle$, D is a corresponding n -tuple of domains $D = \langle D_1, D_2, \dots, D_n \rangle$ such that $x_i \in D_i$, C is a t -tuple of constraints $C = \langle C_1, C_2, \dots, C_t \rangle$.

A constraint C_j is a pair $\langle R_{S_j}, S_j \rangle$ where R_{S_j} is a relation on the variables in $S_j = \text{var}(C_j)$, i.e., the set of variables occurring in C_j . In other words, R_{S_j} is a subset of the Cartesian product of the domains of the variables in S_j .

A solution to the CSP \mathcal{P} is an n -tuple $A = \langle a_1, a_2, \dots, a_n \rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of A onto the set S_j .

In a given task one may be required to find the set of all solutions of a CSP \mathcal{P} , $\text{sol}(\mathcal{P})$, to determine if that set is non-empty or just to find any solution, if one exists. If the set of solutions is empty the CSP is *unsatisfiable*. This simple but powerful framework captures a

wide range of significant applications in fields as diverse as artificial intelligence, operations research, scheduling, supply chain management, to name but a few.

To solve a given problem by means of constraint programming one first formulates it as a CSP. To this end one: (i) introduces some variables ranging over specific domains and constraints over these variables; (ii) chooses some language in which the constraints are expressed (usually a small subset of first-order logic) [Apt03]. This part of the problem solving is called *modeling*.

As an example of modeling with a CSP, let us consider a so-called cryptarithmic problem, i.e. $SEND + MORE = MONEY$. Cryptarithmic problems are mathematical puzzles in which the digits are replaced by letters of the alphabet or other symbols. In the problem under consideration we are asked to replace each letter by a different digit so that the following sum is correct.

$$\begin{array}{r} SEND \\ + \quad MORE \\ \hline = \quad MONEY \end{array}$$

Here the variables are S, E, N, D, M, O, R, Y . Because S and M are the leading digits, the domain for each of them consists of the integer interval $[1..9]$. The domain of each of the remaining variables consists of the integer interval $[0..9]$. This problem can be formulated as the equality constraint

$$\begin{aligned} & + \quad 1000 \cdot S \quad + \quad 100 \cdot E \quad + \quad 10 \cdot N \quad + \quad D \\ & + \quad 1000 \cdot M \quad + \quad 100 \cdot O \quad + \quad 10 \cdot R \quad + \quad E \\ = & \quad 10000 \cdot M \quad + \quad 1000 \cdot O \quad + \quad 100 \cdot N \quad + \quad 10 \cdot E \quad + \quad Y \end{aligned}$$

combined with 28 disequality constraint $x \neq y$ for x, y ranging over the set of variables $\{S, E, N, D, M, O, R, Y\}$ with x preceding y in, say, the presented order. The resulting CSP is $\mathcal{P} = \langle X, D, C \rangle$, where:

$$\begin{aligned} X &= \langle S, E, N, D, M, O, R, Y \rangle \\ D &= \langle [1..9], [0..9], [0..9], [0..9], [1..9], [0..9], [0..9], [0..9] \rangle \\ C &= \langle 1000 \cdot S + 100 \cdot E + 10 \cdot N + D + 1000 \cdot M + 100 \cdot O + 10 \cdot R + E = \\ &\quad = 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y, \\ &\quad S \neq E, S \neq N, S \neq D, S \neq M, S \neq O, S \neq R, S \neq Y, \\ &\quad E \neq N, E \neq D, E \neq M, E \neq O, E \neq R, E \neq Y, N \neq D, \\ &\quad N \neq M, N \neq O, N \neq R, N \neq Y, D \neq M, D \neq O, D \neq R, \\ &\quad D \neq Y, M \neq O, M \neq R, M \neq Y, O \neq R, O \neq Y, R \neq Y \rangle \end{aligned}$$

In general, more than one representation of a problem as a CSP exists. For example, a minor variation of the above representation of the $SEND + MORE = MONEY$ problem as a CSP is obtained assuming that the domains of all variables are the same, i.e., the interval $[0..9]$, and by adding to the tuple C two disequality constraints, i.e., $S \neq 0$, $M \neq 0$.

3.2 Solving Constraint Satisfaction Problems

The Constraint Satisfaction Problem being NP-complete, it is usually solved by backtrack search procedures that try to extend a partial instantiation to a global one that is consistent. Algorithms for solving CSPs can be partitioned into two broad categories: inference and search, and various combinations of those two approaches [FM06]. If the domains D_i are all finite then the finite search space for putative solutions is the Cartesian products Ω of the domains of the variables of the CSP, i.e., $\Omega = D_1 \times D_2 \times \dots \times D_n$. Ω can, in theory, be enumerated and each n -tuple tested to determine if it is a solution. This blind enumeration technique can be improved upon using two distinct orthogonal strategies: constraint propagation and search. In constraint propagation, local consistency techniques can eliminate large subspaces from Ω on the grounds that they must be devoid of solutions. Search systematically explores Ω , often eliminating subspaces with a single failure. The success of both strategies hinges on the simple fact that a CSP is conjunctive: to solve it, all of the constraints must be satisfied so that local failure on a subset of variables rules out all putative solutions with the same projection onto those variables. These two basics strategies are usually combined in most applications.

3.2.1 Search

Backtrack is the fundamental “complete” search method for constraint satisfaction problems, in the sense that one is guaranteed to find a solution if one exists. Basic backtrack search builds up a partial solution by choosing values for variables until it reaches a dead end, where the partial solution cannot be consistently extended. When it reaches a dead end it undoes the last choice it made and tries another. This is done in a systematic manner that guarantees all the possibilities will be tried. Backtrack improves on simply enumeration and testing of all candidate solutions by brute force in that it checks to see if the constraints are satisfied each time it makes a new choice, rather than waiting until a complete solution candidate containing values for all variables is generated.

The naive backtracking algorithm (BT) is the starting point for all of the more sophisticated backtracking algorithms. The backtrack search process is often represented as a search tree. In the BT search tree, the root node at level 0 is the empty set of assignments, and a node at level j is a set of assignments $\{x_1 = a_1, \dots, x_j = a_j\}$. At each node in the search tree, an uninstantiated variable is selected and the branches out of this node consist of all possible ways of extending the node by instantiating the variable with a value from its domain. The branches represent the different choices that can be made for that variable. In BT, only constraints with no uninstantiated variables are checked at a node. If a constraint check fails - a constraint is not satisfied - the next domain value of the current variable is tried. If there are no more domain values left, BT backtracks to the

most recently instantiated variable. A solution is found if all constraint checks succeed after the last variable has been instantiated.

In the naive backtracking algorithm (BT), a node $p = \{x_1 = a_1, \dots, x_j = a_j\}$ in the search tree is a set of assignments and p is extended by selecting a variable x and adding a branch to a new node $p \cup \{x = a\}$, for each a in the domain of x . The assignment $x = a$ is said to be posted along a branch. As the search progresses deeper in the tree, additional assignments are posted and upon backtracking the assignments are retracted. However, this is just one possible branching strategy, and several alternatives have been proposed and examined in the literature. Usually, branching strategies consist of posting unary constraints. In this case, a *variable ordering heuristic* is used to select the next variable to branch on and the ordering of the branches is determined by a *value ordering heuristic*. Three popular branching strategies involving unary constraints are the following.

Enumeration. The variable x is instantiated to each value in its domain. A branch is generated for each value a in the domain of the variable, and a constraint $x = a$ is posted on each branch.

Binary choice points. The variable x is instantiated to some value in its domain. Assuming the value a is chosen, two branches are generated and the constraints $x = a$ and $x \neq a$ are posted, respectively. This branching strategy is often used in constraint programming languages for solving CSPs (see, e.g., [ILO03, Smo95]).

Domain splitting. Here the variable is not necessarily instantiated, but rather the choices for the variable are reduced in each subproblems. For ordered domains, this could consist of posting a constraint of the form $x \leq a$ on one branch and positing $x > a$ on the other branch.

Branching strategies that consist of posting non-unary constraints have also been proposed, as have branching strategies that are specific to a class of problems (see, e.g., [CL94]). There has been further work on branching strategies that has examined the relative power of the strategies and proposed new strategies (see, e.g., [Hen89, MH02]).

3.2.2 Constraint Propagation

Analysis of using backtracking to solve CSPs shows that it almost always displays pathological *thrashing* behaviors [BR74]. Thrashing is the repeated exploration of failing subtrees of the backtrack search tree that are essentially identical - differing only in the assignments to variables irrelevant to the failure of the subtree. Because there is typically an exponential number of such irrelevant assignments, thrashing is often the most significant factor in the running time of backtracking.

A fundamental insight in improving the performance of backtracking algorithms on CSPs is that local inconsistencies can lead to thrashing or unproductive search [Gas74, Mac77]. A *local inconsistency* is an instantiation of some of the variables that satisfies the relevant constraints but cannot be extended to one or more additional variables and so cannot be part of any solution. If we are using a backtracking search to find a solution, such an inconsistency can be the reason for many dead-ends in the search and cause much futile

search effort. This insight has led to: the definition of conditions that characterize the level of local consistency of a CSP (e.g., [Fre78, Mac77, Mon74]), the development of constraint propagation algorithms - algorithms which enforce these levels of local consistency by removing inconsistencies from a CSP (e.g., [Mac77, Mon74]), and effective backtracking algorithms for finding solutions to CSPs that maintain a level of local consistency during the search (e.g., [Gas74, HE79]).

A generic scheme to maintain a level of local consistency in a backtracking search is to perform constraint propagation at each node in the search tree. Constraint propagation algorithms remove local inconsistencies by posting additional constraints that rule out or remove the inconsistencies. When used during search, constraints are posted at nodes as the search progresses deeper in the tree. But upon backtracking over a node, the constraints that were posted at that node must be retracted. When used at the root node of the search tree - before any instantiations or branching decisions have been made - constraint propagation is sometimes referred to as a preprocessing stage.

Backtracking search integrated with constraint propagation has two important benefits. First, removing inconsistencies during search can dramatically prune the search tree by removing many dead-ends and by simplify the remaining subproblem. In some cases, a variable will have an empty domain after constraint propagation, i.e., no value satisfies the unary constraints over that variable. In this case, backtracking can be initiated as there is no solution along this branch of the search tree. In other cases, the variables will have their domains reduced. If a domain is reduced to a single value, the value of the variable is forced and it does not need to be branched on in the future. Thus, it can be much easier to find a solution to a CSP after constraint propagation or to show that the CSP does not have a solution. Second, some of the most important variable ordering heuristics make use of the information gathered by constraint propagation to make effective variable ordering decisions. As a result of these benefits, it is now standard for a backtracking algorithm to incorporate some form of constraint propagation.

Arc consistency [Mac77] is the oldest and most well-known way of propagating constraints. This is indeed a very simple and natural concept that guarantees every value in a domain to be consistent with every constraint. Given a constraint c the notation $t \in c$ denotes a tuple t - an assignment of a value to each of the variables occurring in c - that satisfies the constraint c . The notation $t[x]$ denotes the value assigned to variable x by the tuple t . $vars(c)$ denotes the set of variable occurring in c , and $dom(x)$ denotes the domain of variable x .

Definition 2 (Arc consistency). *Given a constraint c , a value $a \in dom(x)$ for a variable $x \in vars(c)$ is said to have a support in c if there exists a tuple $t \in c$ such that $a = t[x]$, and $t[y] \in dom(y)$, for every $y \in vars(c)$. A constraint c is said to be arc consistent if for each variable $x \in vars(c)$, each value $a \in dom(x)$ has a support in c .*

A constraint can be made arc consistent by repeatedly removing unsupported values from the domains of its variables. Enforcing arc consistency on a CSP means iterating over the constraints until no more changes are made to the domains. Algorithms for enforcing arc consistency have been extensively studied. An optimal algorithm for an arbitrary constraint has $O(rd^r)$ worst case time complexity, where r is the arity of the constraint

and d is the size of the domains of the variables [MM88]. Fortunately, it is almost always possible to do much better for classes of constraints that occur in practice.

The MAC algorithm [SF94] maintains arc consistency on constraints with at least one uninstantiated variable during backtracking search. At each node of the search tree, an algorithm for enforcing arc consistency is applied to the CSP. Since arc consistency was enforced on the parent of a node, initially constraint propagation only needs to be enforced on the constraint that was posted by the branching strategy. In turn, this may lead to other constraints becoming arc inconsistent and constraint propagation continues until no more changes are made to the domains. If, as a result of constraint propagation, a domain becomes empty, the branch is a dead-end and is rejected. If no domain is empty, the branch is accepted and the search continues to the next level.

When maintaining arc consistency during search, any value that is pruned from the domain of a variable does not participate in any solution to the CSP. However, not all values that remain in the domains necessarily are part of some solution. Hence, while arc consistency propagation can reduce the search space, it does not remove all possible dead-ends. Let us say that the domains of a CSP are *minimal* if each value in the domain of a variable is part of some solution to the CSP. Clearly, if constraint propagation would leave only the *minimal domains* at each node in the search tree, the search would be backtrack-free as any value that was chosen would lead to a solution. Unfortunately, finding the minimal domains is at least as hard as solving the CSP. After enforcing arc consistency on individual constraints, each value in the domain of a variable is part of some solution to the constraint considered in isolation. Finding the minimal domains would be equivalent to enforcing arc consistency on the conjunction of the constraints in a CSP, a process that is worst-case exponential in n , the number of variables in the CSP. Thus, arc consistency can be viewed as approximating the minimal domains.

In general, there is a tradeoff between the cost of the constraint propagation performed at each node in the search tree, and the quality of the approximation of the minimal domains. One way to *improve* the approximation, but with an increase in the cost of constraint propagation, is to use a stronger level of local consistency such as *singleton consistency* [PSW00]. One way to *reduce* the cost of constraint propagation, at the risk of a poorer approximation to the minimal domains and an increase in the overall search cost, is to restrict the application of arc consistency. One such algorithm is called forward checking. The *forward checking* algorithm (FC) [HE79] maintains arc consistency on constraints with exactly one uninstantiated variable. On such constraints, arc consistency can be enforced in $O(d)$ time, where d is the size of the domain of the uninstantiated variable.

Forward checking is just one way to restrict arc consistency propagation. An alternative to restricting the application of arc consistency is to restrict the definition of arc consistency. One important example is *bounds consistency* [CHLS06]. Suppose that the domains of the variables are large and ordered and that the domains of the variables are represented by intervals (the minimum and the maximum value in the domain). With bounds consistency, instead of asking that each value $a \in \text{dom}(x)$ has a support in the constraint, we only ask that the minimum value and the maximum value each have a support in the constraint. Although in general weaker than arc consistency, bounds consistency has been shown to be useful for arithmetic constraints and global constraints [Pug98, QGLOB05] as it can

sometimes be enforced more efficiently.

3.3 Constraint Logic Programming

Constraint Logic Programming (CLP) is the merger of two declarative paradigms: constraint solving and logic programming. As both constraint solving and logic programs are based on mathematical relations the merger is natural and convenient [MSW06]. CLP encourages experimentation and fast algorithm development by narrowing the gap between the logic and the solving algorithms. This is because CLP can express both conceptual and design models and, even more importantly, CLP can also express mappings from conceptual to design models. A *conceptual* model of a problem is its precise formulation in logic, while the *design* model of the problem is its algorithmic formulation, which maps to a sequence of steps for solving it.

3.3.1 Modeling and Solving CSPs

The first important characteristic of constraint logic programs is that they allow succinct, natural conceptual modeling of constraint satisfaction problems. For example, the cryptarithmic problem presented in Section 3.1, is naturally conceptually modeled by the following CLP program (we will use the concrete syntax of SWI Prolog [Uni11] throughout this section).

```

1: smm(S,E,N,D,M,O,R,Y) :-
2:   [S,M] ins 1..9,
3:   [E,N,D,O,R,Y] ins 0..9,
4:   S #\= E, S #\= N, S #\= D, S #\= M, S #\= O, S #\= R, S #\= Y,
5:   E #\= N, E #\= D, E #\= M, E #\= O, E #\= R, E #\= Y, N #\= D,
6:   N #\= M, N #\= O, N #\= R, N #\= Y, D #\= M, D #\= O, D #\= R,
7:   D #\= Y, M #\= O, M #\= R, M #\= Y, O #\= R, O #\= Y, R #\= Y,
8:   1000 * S + 100 * E + 10 * N + D
9:   + 1000 * M + 100 * O + 10 * R + E
10:  #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

Line 1 initiates a rule to define a new predicate (or user-defined constraint) `smm` which has the variables of the problem as arguments. The remainder of the rule defines `smm` in terms of other constraints. Line 2 defines variables `S` and `M` as integer in the range 1 to 9, while line 3 defines that the other variables are integer in the range 0 to 9. Constraints in lines 4 to 7 encode the 28 disequalities ensuring that all the variables take different values. Finally, the lines 8 to 10 constrain the variables to satisfy the cryptarithmic constraint.

The standard class of CSPs admits a fixed finite set of variables X , upon which are imposed a number of constraints. A unary *domain* constraint is imposed upon each variable, restricting the set of values it can take. The remaining constraints involve more than one variable. Each n -ary constraint has a definition, formally expressed as a set of n -tuples, and a scope which is a sequence of n variables from the set X . CLP offers a direct conceptual model for all CSPs in this class. CLP systems typically provide equality

and disequality as well as standard mathematical functions and relations as built-in, i.e., primitive constraints. Moreover, as we have seen in the above example, CLP provides a very natural way to express logical combinations of constraints. Briefly, if the constraint $C1$ is the conjunction of $C2$ and $C3$, this can be expressed in CLP as $C1 :- C2, C3$. If, on the other hand, the constraint $C1$ is the disjunction of $C2$ and $C3$ then in CLP we can write

```
C1 :- C2.  
C1 :- C3.
```

One of the strengths of CLP languages is that they inherit constraint solving over finite trees (i.e. terms) from logic programming languages. These provide standard data structures such as records, lists, and trees. Such data structures are important because they allow the conceptual model and other constraints to be parametric in the number of variables. This is important because for many CSP problems and many constraints, the number of variables depends on the runtime data.

As a matter of fact, CLP languages are much more expressive than most other approaches for defining constraint problems, in particular the standard framework of constraint satisfaction problems (CSPs) or mathematical modeling languages such as GAMS or OPL. They allow local variables and recursive definitions that together allow one to express problems with an unbounded number of variables. They can also represent solutions without necessarily fixing all variables. This enables CP languages to support interactive problem solving. For example the user can control search, by posting search decisions one at a time, and observing the resulting partial solution calculated by the CLP program, before deciding what to do next.

The second important characteristic of CLP languages is that they allow the programmer to define search strategies for solving their model. This is a core component of the design model. This is possible because CLP languages inherit backtracking search from logic programming. When combined with reflection predicates that provide information about the current solver state, this allows the programmer to specify sophisticated, efficient, problem specific search strategies.

The most basic search procedure in a CLP system is called *labeling* and can be defined by the following two rules [MSW06]:

```
labeling([]).  
labeling([V|Rest]) :-  
    indomain(V),  
    labeling(Rest).
```

This predicate recurses through a list of variables and uses the predicate `indomain(V)` to non-deterministically set each variable V to each of its possible values in turn.

We can solve the $SEND + MORE = MONEY$ cryptarithmic problem by combining this search predicate with the model above in the goal

```
smm(S,E,N,D,M,O,R,Y), labeling([S,E,N,D,M,O,R,Y]).
```

What we obtain is a design model for our problem assuming the existence of an underlying finite domain constraint solver. When this goal is evaluated by a CLP system it will return the answer $S = 9$, $E = 5$, $N = 6$, $D = 7$, $M = 1$, $O = 0$, $R = 8$, $Y = 2$.

In SWI Prolog [Uni11], as in other CLP languages (e.g., SICStus Prolog [Swe08], ECLiPS^e [AW06]), the predicate `labeling` is a built-in predicate with two arguments. One is the list of variables to label, the other is a list of options that allow the user to specify some parameters of the labeling procedure. Usually, the parameters that can be set are the variable selection strategy (i.e., which variable to label next), the value selection strategy (i.e., which value in the domain to select), and the branching strategy.

3.3.2 Constraint Solvers

One of the distinguishing features of CLP languages is that constraints are generated dynamically, and tests for satisfaction on the partially generated constraints controls subsequent execution and constraint generation. This contrasts to say mathematical modeling languages in which constraint solving only takes place after the constraints are generated.

Incremental constraint solvers keep an internal solver state which represents the constraints encountered so far in the derivation. As new constraints are added the solver state is updated and checked for unsatisfiability. When the solver detects that the current state is unsatisfiable execution returns to the last state with an unexplored child state. The solver-state must be restored to a state equivalent to the solver-state at that point and continue execution on the unexplored derivation, i.e., the solver has to *backtrack*.

Constraint logic programming has been the birthplace of a number of constraint solving algorithms for supporting incremental backtracking constraint solving. In Prolog II an incremental algorithm for solving equations and disequations was developed [Col84].

Finite domain solvers utilizing artificial intelligence techniques were first incorporated in CHIP [DHSA88]. Essentially the finite domain solver maintains a record of the possible values of each variable, its *domain*, and implements constraints as *propagators* (also known as *filtering algorithms*) that are executed when the domains of the variables involved change. Each propagator possibly reduces the domains of the variables involved in the constraint, enforcing some level of local-consistency. The process is repeated until no propagator can change a domain of a variable. The solvers are incremental since adding a new constraint simply involves scheduling its propagators, and then remembering the propagators for later rescheduling. Finite domain solvers are guaranteed to be complete when every variable is fixed to a unique value.

Another important class of domain constraints are interval domains over floating point numbers. In this case propagation based techniques are also used to solve arithmetic constraints over the floating point numbers. The use of floating point intervals for constraint solving was suggested by Cleary [Cle87] and independently by Hyvönen [Hyv89]. The first implementation in a CLP system was in BNR Prolog, and is discussed in [OV90, OV93].

Typically, CLP solvers support backtracking by trailing (inherited from the WAM machine of Prolog [War83, Ait91]) or copying [Sch99]. Trailing records changes made to the solver state, and then undoes the changes on backtracking. Copying simply copies the solver state, and backtracks by moving back to an old copy.

Reified Constraints

Most finite domain constraint solvers allow the user to add an extra Boolean argument to the simple primitive constraints, that enforces or prohibits the constraint according to the value of the Boolean. For example, the constraint $1 \#<==> (X \# = Y)$ enforces the constraint $X = Y$ and is equivalent to the constraint $X \# = Y$. On the other hand, the constraint $0 \#<==> (X \# = Y)$ prohibits $X = Y$ and is equivalent to $X \# \neq Y$. Let us consider the following constraint definition.

```
soft_diff(X,Y,0) :- X #\= Y.
soft_diff(X,X,1).
```

We have that the constraint `soft_diff(X,Y,B)` is equivalent to $B \#<==> (X \# = Y)$, but the reified constraint does not use multiple rules to define the “disjunctive” constraint.

CLP languages introduced reified constraints to finite domain constraint solvers, originally using a cardinality combinator [HD93]. They are now a standard feature of finite domain constraint solvers.

Let us represent a reified constraint as $c[B]$, where c is the original constraint, and B is the extra Boolean variable. The ideal behavior of the reified constraint $c[B]$ is to propagate domain reductions on its variables as follows.

- If, with the current domains, c is unsatisfiable, then propagate $B = 0$;
- If the constraint is satisfied by every combination of values from the domains of its variables, then propagate $B = 1$;
- If $B = 0$, then impose the constraint $\neg c$;
- If $B = 1$, then impose c .

Some reified constraints in some CLP systems have weaker propagation than this, but all systems propagate a fixed Boolean value as soon as all the variables in c are instantiated.

Reified primitive constraints can be straightforwardly extended to reify logical combinations of primitive constraints using conjunction, disjunction, and implication. The power of reified constraints is that they allow logical combinations of constraints to be expressed.

Global Constraints

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables. An example is the constraint `alldifferent`(x_1, \dots, x_n) [vH01], which specifies that the values assigned to the variables x_1, \dots, x_n must be pairwise distinct. Typically, a global constraint is semantically redundant in the sense that the same relation can be expressed as the conjunction of several simpler constraints. Having shorthands for frequently recurring patterns clearly simplifies the programming task. What may be less obvious is that global constraints also facilitate the work of the constraint solver by providing it with a better view of the structure of the problem. Other example of global constraints are: `table` [LS06], for extensional representation of constraints; `global_cardinality` [Rég96],

a generalization of `alldifferent`; `cumulatives` [BC02], for modeling scheduling problems. Many other global constraints have been proposed in the past year, the Global Constraint Catalog [BCR10] lists about 354 constraints. Existing CLP languages offer a consistent number of global constraints as built-in, some systems (e.g., SICStus Prolog [Swe08]), also have mechanisms for the creation of user defined global constraints.

As an example of usage of global constraints, let us consider the CSP for the *SEND + MORE = MONEY* problem introduced in Section 3.1. The 28 disequalities modeling the fact that each letter has to be replaced by a different digit, can be substituted by the constraint `alldifferent(S, E, N, D, M, O, R, Y)`. The following is a CLP program modeling the cryptarithmic problem using the `alldifferent` constraint (in SWI Prolog the `alldifferent` constraint is named `all_different`).

```
1: sum(S,E,N,D,M,O,R,Y) :-  
2:   [S,M] ins 1..9,  
3:   [E,N,D,O,R,Y] ins 0..9,  
4:   all_different(S,E,N,D,M,O,R,Y)  
5:       1000 * S + 100 * E + 10 * N + D  
6:       + 1000 * M + 100 * O + 10 * R + E  
7:   #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

Notice that the constraints in lines 4 to 7 of the CLP program of Section 3.3.1 are replaced by a single `all_different` constraint (line 4) in the above program.

Chapter 4

Morphos Configuration Engine

In this chapter, we give an example of configuration system adopting the model-based approach to configuration, and exploiting Constraint Logic Programming. In particular, we present Morphos Configuration Engine (MCE) [CRD⁺10], a general CLP-based (product) configuration engine developed for the existing commercial configuration platform Morphos. Morphos has been developed by Acritas S.r.l., and successfully used by various companies, whose names are omitted for privacy reasons. It has been designed to be simple to use and easily integrable with the information system of a company.

In MCE, the product model consists of a tree, whose nodes represent the components of a product, and a set of constraints. Each node is paired with a set of variables, which express configurable features of the component it represents. Each variable is endowed with a finite domain (typically, a finite set of integers or strings), which represents the set of its possible values. The set of constraints defines the compatibility relations between properties of product components. MCE takes advantage of a model of the product and of user's choices about property values to create a program in SWI Prolog [Uni11], which encodes a Constraint Satisfaction Problem (CSP). Each variable of the CSP represents a variable of the product being configured, while the constraints of the CSP encode the constraints that specify the relationships among product variables. Once created, the program is executed using the finite domain constraint solver of SWI Prolog. If all constraints are satisfiable, the execution of the solver on the given program returns for each variable the associated, possibly restricted, domain; otherwise, it certifies the inconsistency of the encoded CSP. Once the solver computation (successfully) ends, for each variable whose domain has been restricted, MCE communicates to Morphos the restricted domain computed by the solver. Exploiting these pieces of information, Morphos is able to prevent user's choices that violate the given constraints. In case of a failure, MCE supports the user in the process of assignment revision.

In Section 4.1, we first describe the main features of Morphos product models and then we illustrate the structure of the MCE configuration process. Section 4.2 presents the encoding of the product configuration problem as a constrain satisfaction problem. Section 4.3 is devoted to the process of assignment revision.

4.1 Product models and Configuration Process

In this section, we first present the structure of Morphos product models (cf. Section 4.1.1), then we describe the organization of the configuration process in MCE (cf. Section 4.1.2).

4.1.1 Morphos Product models

A *Morphos product model* consists of the following components.

- A tree, called *product model tree*, whose nodes represent well-defined components of a product; it basically defines the *has-part/is-part-of* relation over product components.
- For each node N of the product model tree, a set \mathcal{V} of *node variables* that define the configurable characteristics of the corresponding component; each node variable $V \in \mathcal{V}$ is endowed with a domain $D(V)$ (i.e., a finite set of integers or strings), which specifies the set of values V can assume.
- A set of constraints on node variables, called *compatibility relations pool*, that define the compatibility relations between configurable characteristics of product components, e.g., some configurations of components are incompatible, other ones depend on each other.

The elements of the compatibility relations pool are constraint on node variables, integers, and strings, built using arithmetic operators (i.e., $+$, $-$, $*$, $/$, **mod**), comparison operators (i.e., $<$, \leq , $=$, \geq , $>$, \neq), and Boolean operators (i.e., \neg , \vee , \wedge , \Rightarrow). Three syntactical subclasses of the set of constraints that turn out to be particularly useful in product modeling are defined.

- *Rule constraints.* A rule constraint is a constraint of the form $C_1 \Rightarrow C_2$, where C_1 and C_2 are constraints. C_1 is called *condition*, C_2 is called *action*.
- *Integrity constraints.* An integrity constraint is a constraint C , whose outermost connective (if any) is not \Rightarrow .
- *Disjunctive Normal Form (DNF) constraints.* A DNF constraint over a set of variables $\mathcal{V} = \{X_1, X_2, \dots, X_k\}$ is a disjunction of the form $C_1 \vee \dots \vee C_h$, where each disjunct C_i is a conjunction of the form $X_{i,1} = v_{i,1} \wedge \dots \wedge X_{i,k_i} = v_{i,k_i}$, with $k_i \leq k$ and $X_{i,j} \neq X_{i,j'}$ for $j \neq j'$, where $X_{i,j}$ is a variable belonging to $\{X_1, X_2, \dots, X_k\}$ and $v_{i,j}$ is an integer number or an ASCII string.

A DNF constraint is *complete* if it is of the form:

$$(X_1 = v_{1,1} \wedge X_2 = v_{1,2} \wedge \dots \wedge X_k = v_{1,k}) \vee \dots \vee (X_1 = v_{h,1} \wedge X_2 = v_{h,2} \wedge \dots \wedge X_k = v_{h,k})$$

The distinction between rules and integrity constraints is merely syntactic as $A \Rightarrow B$ (rule constraint) is the same as $\neg A \vee B$ (integrity constraint). The introduction of the class

of rule constraints is motivated by the fact that they make it possible to adopt a rule-based style in the specification of the dependencies among product features, which is quite common in existing configuration systems (Morphos included).

As an example of Morphos product model let us consider a simple model for a bicycle. The structure of the product model tree is depicted in Figure 4.1. Notice that *has-part* relation cardinality constraints are hard coded in the Morphos product model tree.

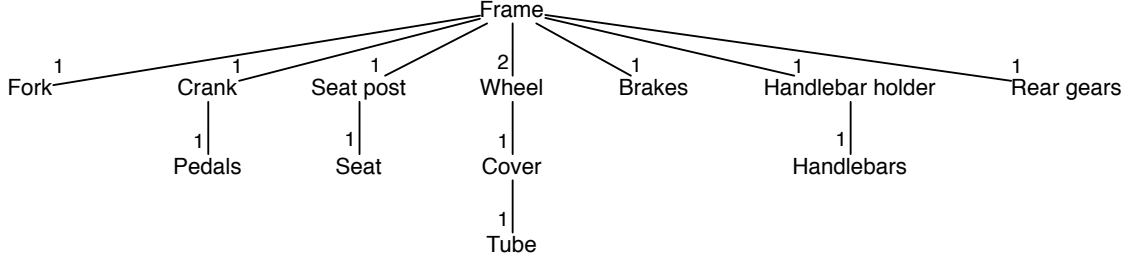


Figure 4.1: The product model tree of a bicycle.

The set of variables for the nodes Fork, Crank, Wheel, and Cover are the following (the set of variables for the other nodes in the tree are omitted as they are not relevant to the purpose of this example).

$$\begin{aligned}
 \mathcal{V}_{\text{Fork}} &= \{\text{Type} \in \{\text{Racing}, \text{MTB}\}, \\
 &\quad \text{ShockAbsorber} \in \{80\text{mm}, 100\text{mm}, 85\text{--}135\text{mm}, \text{SingleChassis}\}\} \\
 \mathcal{V}_{\text{Crank}} &= \{\text{Type} \in \{\text{Racing}, \text{MTB}\}, \\
 &\quad \text{Gears} \in \{46/36, 48/38, 50/34, 44/32/33\}\} \\
 \mathcal{V}_{\text{Wheel}} &= \{\text{Type} \in \{\text{Racing}, \text{MTB}\}, \\
 &\quad \text{Diameter} \in \{22, 24, 26, 28, 29\}, \\
 &\quad \text{Cover} \in \{\text{Traditional}, \text{Tubeless}\}\} \\
 \mathcal{V}_{\text{Cover}} &= \{\text{Type} \in \{\text{Racing}, \text{MTB}\}, \\
 &\quad \text{Typology} \in \{\text{Traditional}, \text{Tubeless}\}, \\
 &\quad \text{Diameter} \in \{22, 24, 26, 28, 29\}\}
 \end{aligned}$$

The following are a rule constraint, an integrity constraint, and a complete DNF constraint that define compatibility relations between node variables of the bicycle product model. We use a table as a shorthand for the DNF constraint. We refer to node variables using a

dot-notation. For example, `Fork.Type` denotes the variable `Type` of the node `Fork`.

Rule constraint : **Condition :** `Fork.Type = Racing`

Action : `Fork.ShockAbsorber = SingleChassis`

Integrity constraint : **Condition :** `Wheel.Cover = Cover.Typology`

Complete DNF constraint :

Crank.Type	Crank.Gears
<i>Racing</i>	46/36
<i>Racing</i>	48/38
<i>Racing</i>	50/34
<i>MTB</i>	44/32/22

4.1.2 MCE Configuration Process

The configuration task consists of finding a configured product (*configuration*), that is, a set of customized components together with a description of their relationships, that satisfies all user's preferences on product characteristics and all the relations defined by the product model. This task is accomplished by the user through the *configuration process*. During this interactive process, the user (i) selects the components that will compose the configured product and (ii) chooses suitable values for their configurable characteristics. MCE checks the validity of user's choices with respect to the product model and reacts to user's inputs by propagating their effects.

Given the product model defined in Section 4.1.1, a configuration can be viewed as a set of node instances, whose variables have been assigned a value belonging to their domain, and a tree, called *instance tree*, that specifies the pairs of node instances in the relation *has-part*. In the following, we will use the term *partial configuration* to indicate the set of node instances and the instance tree at an intermediate state of the configuration process, that is, to refer to a configuration under construction. Figure 4.2 shows the instance tree of a partial configuration of a bicycle, where different instances of the same node are identified by a different *id*.

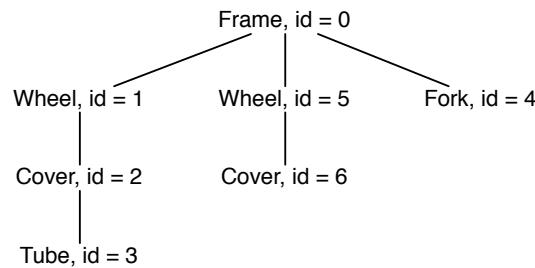


Figure 4.2: Bicycle: an example of instance tree.

The general structure of the configuration process in Morphos, with a special attention to the role of MCE, is pictorially described in Figure 4.3. First, Morphos initializes MCE

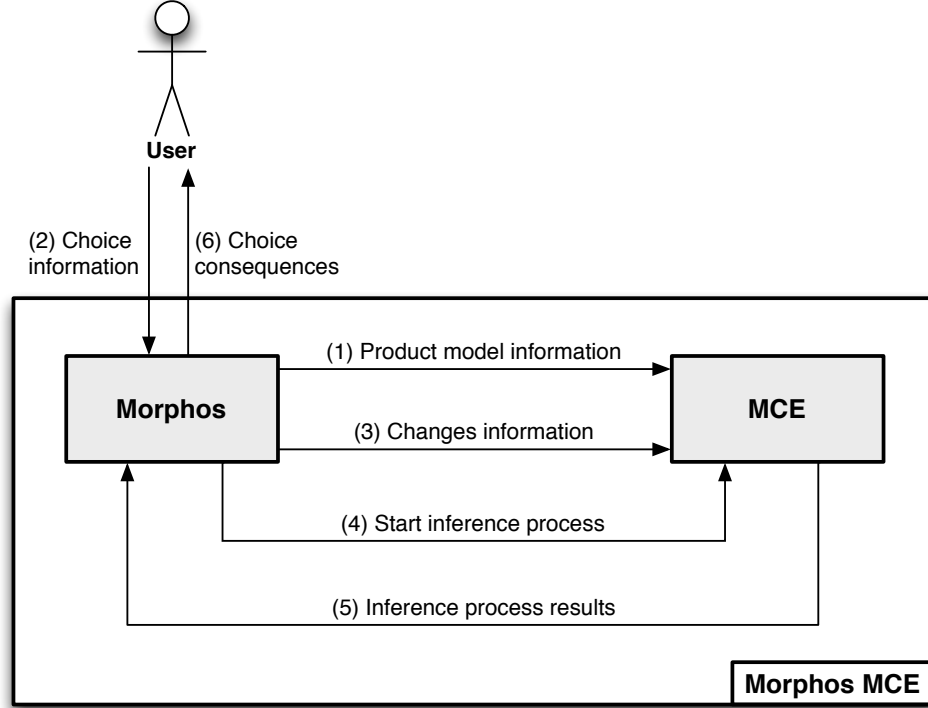


Figure 4.3: Morphos MCE: the configuration process.

(1) by sending to it information about the model of the product to be configured, that is, nodes, variables, and constraints defined by the product model. After such an initialization phase, the interaction with the user starts. The user makes her/his choices using the Morphos interface (2). She/he can add or remove node instances (product components) and set the values of node variables (product component characteristics). When adding an instance n_i of a node N_i having a node N_j as its father in the product model tree, the user specifies the instance n_j of N_j that will be the father of n_i in the instance tree. Morphos communicates to MCE each data variation specified by the user (3) and MCE updates the current partial configuration accordingly. Whenever an update of the partial configuration takes place, the user, through the Morphos interface, can activate the MCE inference process (4). MCE encodes the product configuration problem in a CLP program (encoding a CSP), and it uses the finite domain solver of SWI Prolog to propagate the logical effects of user's choices. Once the inference process ends, MCE returns to Morphos the results of its computation (5). In its turns, Morphos communicates to the user the consequences of her/his choices on the (partial) configuration (6).

Figure 4.4 shows the interface of the Morphos MCE system. It consists of a series of controls for the configuration and the assignment revision processes (the assignment revision process will be analyzed in Section 4.3), together with areas for showing data of

the current partial configuration. The *Product Model* text box and the *Select* button allow one to select and load the file encoding the product model of the product to be configured. Buttons *Add Node*, *Remove Node*, *Set Value*, and *Unset*, together with the associated text boxes, allow one to add a node instance, to remove a node instance, to assign a value to a node instance variable, and to remove a variable assignment, respectively. The button *Run* activates the MCE inference process. Buttons *Allowed Values*, *Backjump*, and *Recompute*, together with the associated text boxes, allow one to perform the assignment revision process. The list box *Nodes* shows the node instance created, the list box *Variables* shows the instance variables that have a value assigned. The main box is used to show the results of the inference process, that is, the restricted domains of node instance variables. Product models are encoded in XML files. The interface and the methods realizing the MCE configuration and assignment revision processes have been implemented using the C# language. XML messages are used for data exchanges between the interface and the MCE methods. Files are used for communication purposes between MCE and SWI-Prolog.

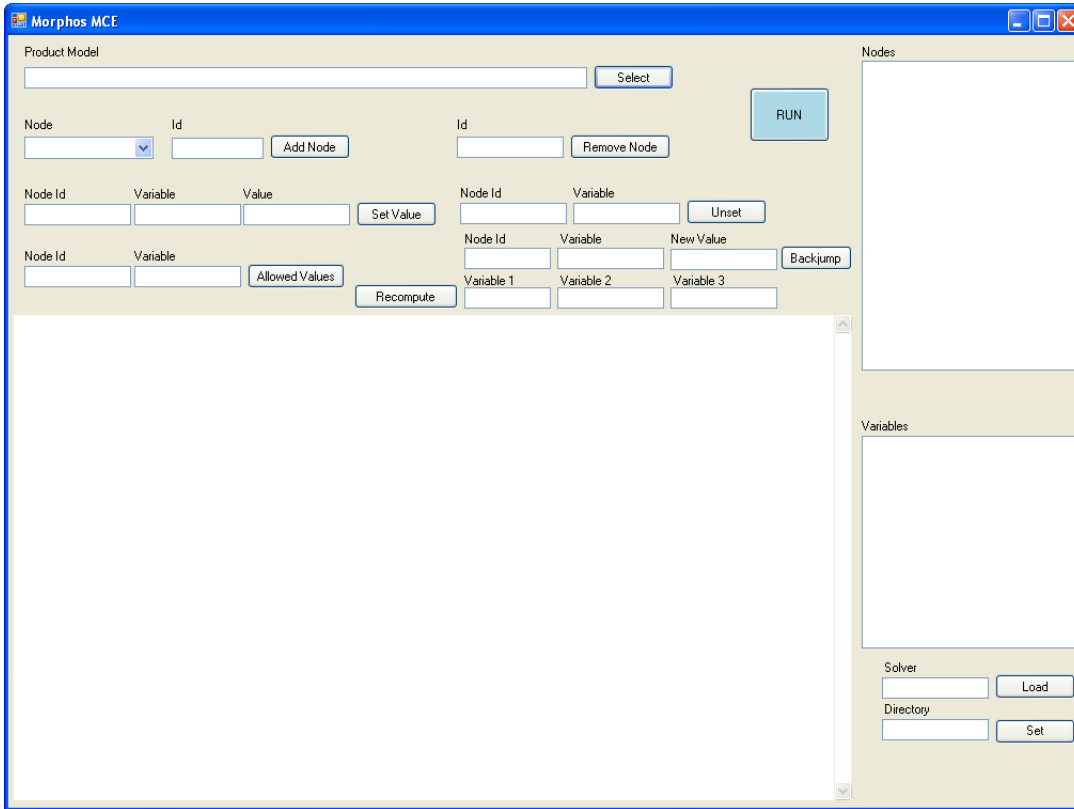


Figure 4.4: Interface of the Morphos MCE system.

The way in which the configuration process takes place makes it clear that the proposed product configuration system is a software tool *supporting* the product configuration activity and not a system for *automatic* product configuration. Hence, it goes without saying that solving a configuration problem requires a constant human-machine interaction.

4.2 Encoding of the Product Configuration Problem

Given the product model and a partial configuration (node instances, identified by a node name N and an id i , user's choices for the values of the variables associated with them, and the instance tree), MCE generates an SWI Prolog program, encoding a CSP, to compute the consequences of user's choices on the (partial) configuration. Basically, it applies a transformation function τ that maps variables and constraints into the corresponding entities of a CSP formulated in Constraint Logic Programming.

Constraints on finite domains and node variables domains are formulated according to the concrete syntax of SWI Prolog. Variables of the CSP correspond to variables of nodes instances belonging to the partial configuration. They are identified by the concatenation of three elements, namely, the variable p of the instance with id i of a node N .

Constraints defining compatibility relations among properties of product components are mapped into SWI Prolog constraints using the transformation function τ as follows:

- a rule constraint $C_1 \Rightarrow C_2$ is translated into $\tau(C_1) \#==> \tau(C_2)$ ($\#==>$ is the SWI Prolog operator for logical implication);
- an integrity constraint C is translated into $\tau(C)$;
- a DNF constraint $C_1 \vee \dots \vee C_h$ is translated into $\tau(C_1) \# \vee \dots \vee \tau(C_h)$ ($\# \vee$ is the SWI Prolog operator for logical disjunction). A complete DNF constraint is translated using the built-in combinatorial global constraint of SWI Prolog `tuples_in/2`.

The constraints of the CSP are the elements of the compatibility relation pool, appropriately instantiated with variables of node instances in the partial configuration (as a matter of fact, the choices of the values for the instance variables can be viewed as primitive constraints and thus they are constraints of the CSP as well). The (constraint) instantiation mechanism works as follows. Let c be an element of the compatibility relation pool, involving variables of different nodes. If the nodes whose variables are involved in c all belong to the same path from the root to a leaf of the product model tree, then c must hold for all the tuples of instance variables that belong to the same path from the root to a leaf of the instance tree (notice that multiple paths in the instance tree may correspond to a single path in product model tree). Otherwise, c must hold for all the ordered tuples of instance variables it involves. For a detailed description of this instantiation mechanism we refer the reader to [CRD⁺10].

Given a program with the above-described characteristics, the finite domain solver of SWI Prolog can be used to reduce the domains associated with variables, preserving satisfiability, or to detect the inconsistency of the encoded CSP (due to user's assignments that violate the set of constraints or to inconsistencies of the original product model). When the computation of the solver ends, MCE communicates to Morphos the reduced domains and, by exploiting information about (restricted) domains, Morphos can prevent user's choices that would violate the constraints, that is, subsequent user's choices incompatible with previous ones.

4.3 Assignment Revision Process

In this section, we describe the process of assignment revision that plays an important role in the Morphos MCE system. This process copes with two situations that may occur during the configuration process: (i) the user changes her/his preferences; (ii) a dead end is reached because constraint propagation has not removed all inconsistent values from variable domains. To deal with such situations, Morphos MCE allows the user to modify past choices about property values. Such a functionality is supported by the assignment revision process that allows the user to *jump back* from a dead end or a (partial) configuration that no longer satisfies her/his preferences. Figure 4.5 shows how Morphos MCE manages the assignment revision process.

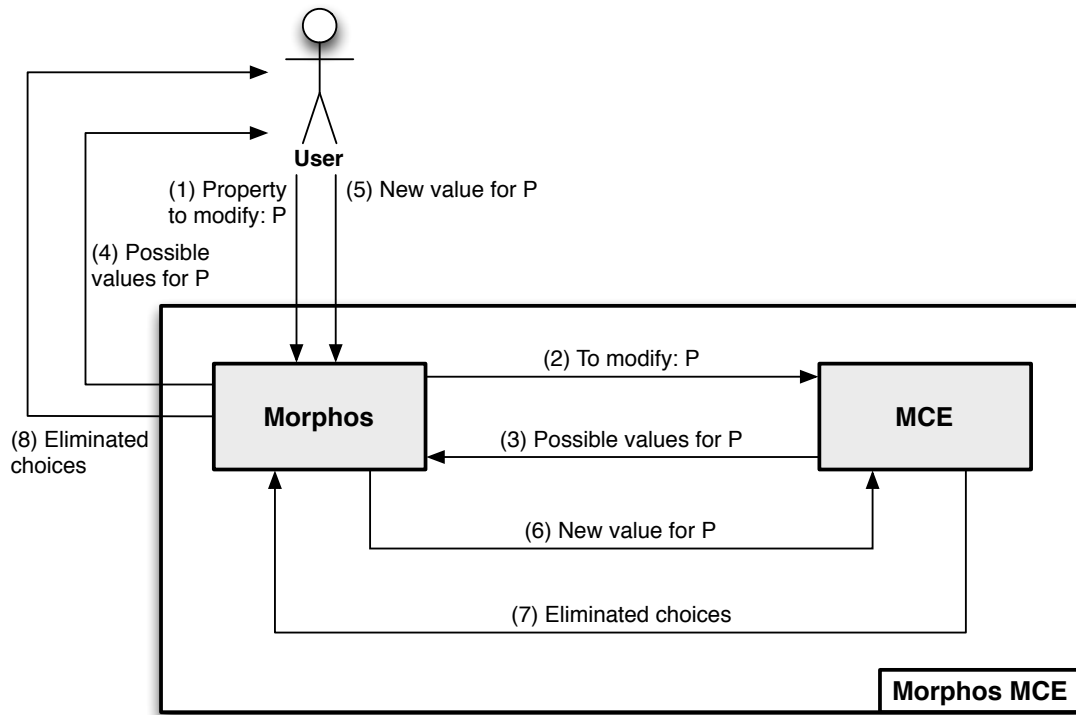


Figure 4.5: Morphos MCE: the assignment revision process.

We illustrate such a capability of the system by means of a simple example. Let us assume that the user would like to modify the value he/she previously assigned to variable P. The user selects variable P through the Morphos interface (1). Morphos communicates to MCE that the user would like to modify the value of P (2). MCE computes the set of values that can be assigned to P without generating any conflict with constraints and choices made before the one concerning variable P, and it communicates the result to Morphos (3). Morphos shows to the user the values he/she can assign to P (4). The user chooses the new value for P (5). Morphos communicates it to MCE (6). MCE computes a maximal subset of the choices made after the one concerning P that can be maintained

without introducing any conflict with constraints, previous choices, and the new choice for P , and it communicates to Morphos the choices that have to be withdrawn (7). Morphos shows to the user which choices must be withdrawn to keep the configuration consistent (8). The user can change the subset of maintained choices according to her/his own preferences. In doing that, he/she can impose to the system to reintroduce some choices it withdrawn, thus forcing it to recompute the subset of choices to withdraw.

We now describe how MCE takes advantage of the finite domain solver of SWI Prolog during the assignment revision process. Let us consider again the case when the user would like to modify the value he/she assigned to variable P . To compute the set of alternative values for P that generate no conflict with constraints and choices made before the one concerning P , MCE defines an SWI Prolog program encoding a CSP as in the configuration process, but without considering the primitive constraints representing choices on variable values made after the one the user would like to modify. To compute a maximal subset of the choices made after the one concerning P that can be maintained without generating conflicts with constraints, previous choices, and the new choice for P , MCE defines an SWI Prolog program encoding a CSP as in the configuration process, but with the following additional features. A variable B_i with domain $\{0,1\}$ is associated with each assignment to variable values A_i made after the one for P . Let m be the number of these assignments. For each of them, a constraint of the form $B_i \# ==> A_i$, where A_i is the translation of the primitive constraint representing A_i , is inserted in the program (instead of A_i). A maximal subset of the assignments A_i that maintains the configuration consistent is computed using a predicate of the form `labeling([max(B_1 + B_2 + ... + B_m)], [B_1, B_2, ..., B_m])`. Such a predicate, through a branch-and-bound algorithm, allows one to compute an assignment of values for variables B_1, B_2, \dots, B_m that maximizes the sum $B_1 + B_2 + \dots + B_m$. The computed assignment determines one maximal subset among all possible ones.

Part II

The PRODPROC Framework

Chapter 5

Product and Production Process Modeling with PRODPROC

In this chapter we present PRODPROC, a declarative constraint-based framework for product and production process modeling. Its purposes are: (i) to allow a user to model a configurable product in terms of components, relations between components, and compatibility relations between component configurable characteristics; (ii) to allow a user to model aspects of the production process for a product that may affect product configuration; (iii) to allow a user to couple a product and a process description, in order to avoid or reduce planning impossibilities due to product configuration, and configuration impossibilities due to production planning. The PRODPROC framework consists of two languages:

- A language for *product modeling* called PROMO;
- A language for *production process modeling* called MART.

The PRODPROC framework has been presented for the first time in [Cam11], in this chapter we describe in details all of its features, give a graphical representation to most of its modeling elements, and define an instance-based semantics for it. In Section 5.1 we introduce some basic notions that are used throughout the document. Section 5.2 describes the language PROMO, while Section 5.3 presents the language MART. In Section 5.4 we define the notion of PRODPROC model, and give a description of the PRODPROC semantics in terms of product and process instances. Throughout the following sections, we will exploit a bicycle example to present the different features of our framework. The syntax of PRODPROC languages is presented in Appendix A.

5.1 Preliminary Notions

PRODPROC supports two basic sorts: the sort `int` for integer numbers and the sort `string` for ASCII strings. We assume that there is a denumerable set of variables \mathcal{V}_{int} of sort `int` and a denumerable set of variables $\mathcal{V}_{\text{string}}$ of sort `string`. Moreover, a domain $D(V)$ may

be associated with a variable V , which contains the set of admissible values for V . We consider as domains sets of integers and sets of ASCII strings.

Both integer and string domains are defined by explicitly enumerating their elements; in addition, whenever possible, an integer domain is compactly represented as an interval:

- $D(V) = \{i_1, i_2, \dots, i_n\}$, with $i_1 \in \mathbb{Z}, \dots, i_n \in \mathbb{Z}$, for $V \in \mathcal{V}_{\text{int}}$;
- $D(V) = [\min, \max]$ ($= \{i \in \mathbb{Z} \mid \min \leq i \leq \max\}$), with $\min \in \mathbb{Z}$ and $\max \in \mathbb{Z}$, for $V \in \mathcal{V}_{\text{int}}$;
- $D(V) = \{s_1, \dots, s_m\}$, where, for $i = 1, \dots, m$, s_i is an ASCII string, for $V \in \mathcal{V}_{\text{string}}$.

We denote with the couple $\langle V, D(V) \rangle$ a variable V endowed with a domain $D(V)$, we refer to such a variable as *domain endowed variable*. Also, we refer to V in $\langle V, D(V) \rangle$ as (*domain endowed*) *variable name*.

The notion of *sorted term* can be recursively defined in the usual way:

- a variable of \mathcal{V}_{int} is a term of sort **int** and a variable of $\mathcal{V}_{\text{string}}$ is a term of sort **string**;
- an integer number is a (ground) term of sort **int** and an ASCII string is a (ground) term of sort **string**;
- if t_1 and t_2 are terms of sort **int**, then $t_1 \oplus t_2$ is a term of sort **int**, where $\oplus \in \{+, -, *, /, \text{mod}\}$, also $|t_1|$ (i.e., absolute value of t_1) is a term of sort **int**;

Terms can be used to build constraints as follows:

- if t_1, t_2 are terms of sort **int** and $op \in \{<, \leq, =, \geq, >, \neq\}$, then $t_1 \text{ op } t_2$ is a *primitive constraint*;
- if t_1, t_2 are terms of sort **string** and $op \in \{=, \neq\}$, then $t_1 \text{ op } t_2$ is a *primitive constraint*;
- primitive constraints are *constraints*; any Boolean combination of constraints is a *constraint*, that is, if C_1 and C_2 are constraints, then $\neg C_1$, $C_1 \vee C_2$, $C_1 \wedge C_2$, and $C_1 \Rightarrow C_2$ are *constraints*.

A DNF constraint over a set of variables $\mathcal{V} = \{X_1, X_2, \dots, X_k\}$ is a disjunction of the form $C_1 \vee \dots \vee C_h$, where each disjunct C_i is a conjunction of the form:

$$X_{i,1} = v_{i,1} \wedge \dots \wedge X_{i,k_i} = v_{i,k_i},$$

with $k_i \leq k$ and $X_{i,j} \neq X_{i,j'}$ for $j \neq j'$, where $X_{i,j}$ is a variable belonging to the set $\{X_1, X_2, \dots, X_k\}$ and $v_{i,j}$ is an integer number or an ASCII string (depending on the sort of $X_{i,j}$). A DNF constraint is *complete* if it is of the form:

$$(X_1 = v_{1,1} \wedge X_2 = v_{1,2} \wedge \dots \wedge X_k = v_{1,k}) \vee \dots \\ \dots \vee (X_1 = v_{h,1} \wedge X_2 = v_{h,2} \wedge \dots \wedge X_k = v_{h,k}).$$

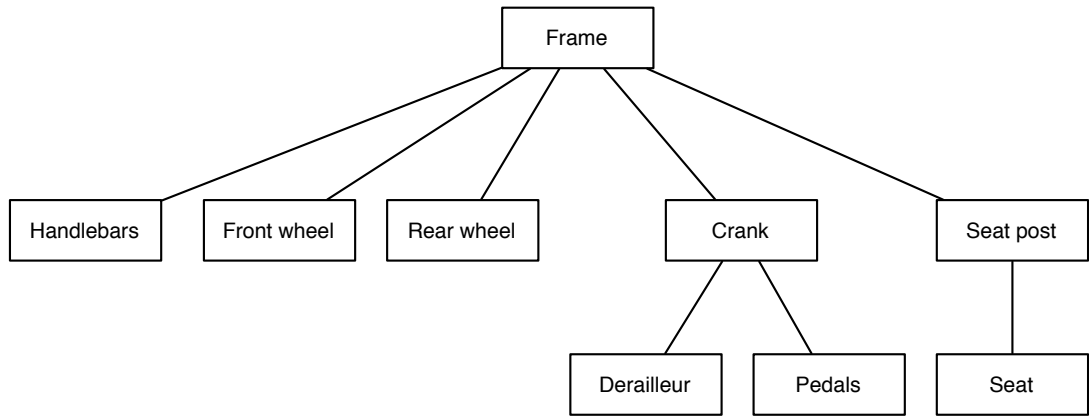
We use

$$\text{valid_tuples}(\mathcal{V}, \{\langle v_{1,1}, v_{1,2}, \dots, v_{1,k} \rangle, \dots, \langle v_{h,1}, v_{h,2}, \dots, v_{h,k} \rangle\}),$$

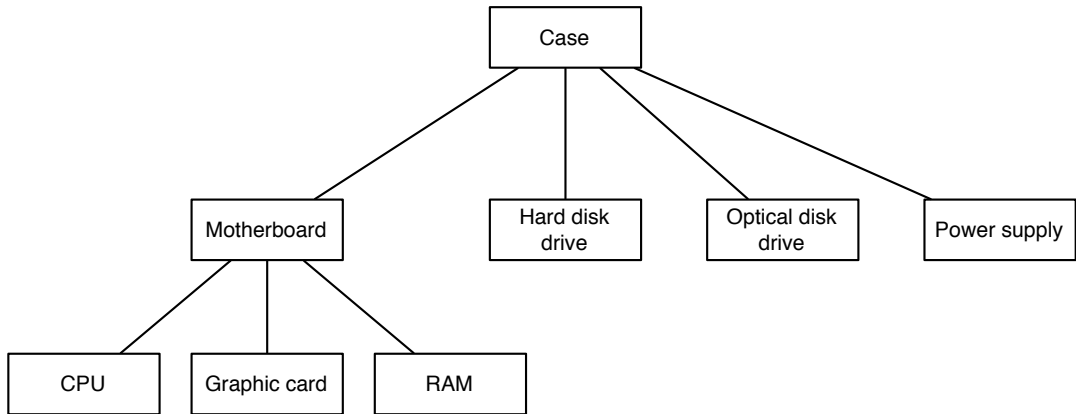
as a shorthand for such a constraint. The notion of constraint we just presented will be used throughout this dissertation.

5.2 The PROMo Language

We are interested in modeling configurable products whose corresponding (producible) variants can be represented as trees. Nodes of these trees correspond to physical components, whose characteristics are all determined. The tree structure describes how the single components taken together define a configured product. Examples of such products are: bicycles, cars, computers, etc. The tree structure of a variant of a bicycle and of a computer case are depicted in Figure 5.1.



(a) Tree structure of a bicycle.



(b) Tree structure of a computer case.

Figure 5.1: Tree structure of configurable product variants.

The PROMo (*Product Modeling*) language allows one to model configurable products in terms of components, relations between components, configurable characteristics, and compatibility relations between configurable characteristics. In particular, a configurable product is represented as a multi-graph, called *product model graph*, and a set of constraints. The nodes of such multi-graph represent well-defined components of a product (e.g., the

frame of a bicycle). The edges model *has-part/is-part-of* relations between product components. Constraints represent compatibility relations between product configurable characteristics, e.g., some configurations of components are incompatible, other ones depend on each other. PROMO defines three types of constraints: node constraints, cardinality constraints, and model constraints.

5.2.1 Product Model Graph

In this section we first introduce the notion of PROMO *nodes* and *edges*. Then, we formally define *product model graphs*.

PROMO nodes. A PROMO *node* is a tuple $\langle N, \mathcal{V}, \mathcal{C} \rangle$ representing a well-defined component, named N , of a product. \mathcal{V} is a set of *node variables* defining the configurable characteristics of the corresponding component (e.g., the number of spokes of a bicycle wheel), and \mathcal{C} is a set of *node constraints*. The following are two PROMO nodes representing well-defined components of a bicycle:

$$\langle \text{Frame}, \mathcal{V}_{\text{Frame}}, \mathcal{C}_{\text{Frame}} \rangle, \quad \langle \text{Wheel}, \mathcal{V}_{\text{Wheel}}, \mathcal{C}_{\text{Wheel}} \rangle.$$

A *node variable* for a node N is a domain endowed variable $\langle V, D(V) \rangle$, where $V \in \mathcal{V}_{\text{int}} \cup \mathcal{V}_{\text{string}}$ and $D(V)$ is a domain of integer or strings that defines the domain for V . $\langle V, D(V) \rangle$ represents a configurable characteristics of the component the node N represents. The following are three node variables for the nodes *Frame* and *Wheel*.

$$\begin{aligned} \langle \text{FrameType}, \{\text{Racing bike}, \text{City bike}\} \rangle &\in \mathcal{V}_{\text{Frame}}, \\ \langle \text{FrameMaterial}, \{\text{Steel}, \text{Aluminum}, \text{Carbon}\} \rangle &\in \mathcal{V}_{\text{Frame}}, \\ \langle \text{WheelType}, \{\text{Racing bike}, \text{City bike}\} \rangle &\in \mathcal{V}_{\text{Wheel}}, \\ \langle \text{SpokeNumber}, [18, 24] \rangle &\in \mathcal{V}_{\text{Wheel}}. \end{aligned}$$

The variables *FrameType* and *WheelType* represent the type of the bicycle frame and of a bicycle wheel, respectively. *FrameMaterial* models the material the bicycle frame is made of, while *SpokeNumber* represent the number of spokes of a wheel.

PROMO edges. An *edge* is a tuple $\langle \text{label}, N, M, \text{Card}, \mathcal{CC} \rangle$ representing an *has-part* relation over product components. *label* is a string representing a description of the relation. N and M are the parent node and the child node in the relation, respectively. *Card* is the *cardinality* of the relation, it is a constant $\text{Card} \in \mathbb{N}^+$, or a domain endowed variable $\langle \text{Card}, D(\text{Card}) \rangle$ where $\text{Card} \in \mathcal{V}_{\text{int}}$, and $D(\text{Card})$ is an integer domain. \mathcal{CC} is a set of *cardinality constraints*. The following are two PROMO edges defining the relations “Front wheel” and “Rear gears” for a bicycle:

$$\begin{aligned} \langle \text{front wheel}, \text{Frame}, \text{Wheel}, 1, \emptyset \rangle, \\ \langle \text{rear gears}, \text{Frame}, \text{Gears}, \langle \text{CG}, \{0, 1\} \rangle, \mathcal{CC} \rangle. \end{aligned}$$

The first edge represents the *has-part* relation between the bicycle frame and its front wheel, it states that a frame has exactly one front wheel. The second edge models the *has-part* relation between the bicycle frame and the rear gears, it states that the frame may have a rear gears component as one of its parts.

Product model graph. PROMO nodes and edges are used to define directed multi-graphs, called *product model graphs*, representing the structure of configurable products. A directed multi-graph is a directed graph which is permitted to have multiple arcs, i.e., arcs with the same source and target nodes. We require the presence of a node without entering edges in product model graphs.

Definition 3 (Product model graph). *A product model graph is a multi-graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$ having the following characteristics:*

- \mathcal{N} is a set of PROMO nodes;
- \mathcal{E} is a set of PROMO edges between nodes in \mathcal{N} ;
- $\exists R \in \mathcal{N}$ such that $\nexists e \in \mathcal{E} \ e = \langle \text{label}, N, R, CC \rangle$ for some $N \in \mathcal{N}$. Such a node R is called root node.

Figure 5.2 shows the graphical representation of PROMO nodes and edges. A PROMO node is depicted as a box with three sections. One for node name, one for node variables, and one for node constraints. A PROMO edge is represented as an arrow from the parent node to the child node, labeled with its label and cardinality, and with an ellipse containing cardinality constraints attached to it.

A fragment of the product model graph of a bicycle is depicted in Figure 5.3. It consists of 5 nodes, each representing a bicycle component, and 5 edges, each representing an *has-part* relation between bicycle components. Notice that two edges connect the node *Frame* with the node *Wheel*, they represent two different relations, i.e., the “Front wheel” relation and the “Rear wheel” relation. The two edges state that a frame has associated one front wheel and one rear wheel. This product structure could be represented also defining a node for the front wheel and a node for the rear wheel. Since the two nodes would have the same characteristics of the node *Wheel* in Figure 5.3, this solution leads to the definition of two copies of the same node. Using two edges with different labels we avoid this redundancy in the product model graph. In general, multiple edges between nodes may allow one to represent components that differs only by their position in the product using only one node, and one edge for each possible position.

In the description of a configured product, physical components are represented by instances of nodes of the product model graph. In particular, an instance of a node *NodeName* consists of a unique id (typically, an integer number) and inherits the name and the set of variables of *NodeName*. Moreover, each variable has a value assigned. An example of instance of the node *Frame* of the bicycle is the triple

$$\langle \text{Frame}, 1, \{ \text{FrameType} = \text{Racing bike}, \text{FrameMaterial} = \text{Carbon} \} \rangle.$$

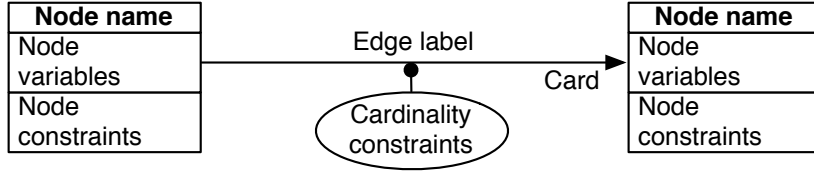


Figure 5.2: PROMO nodes and edges.

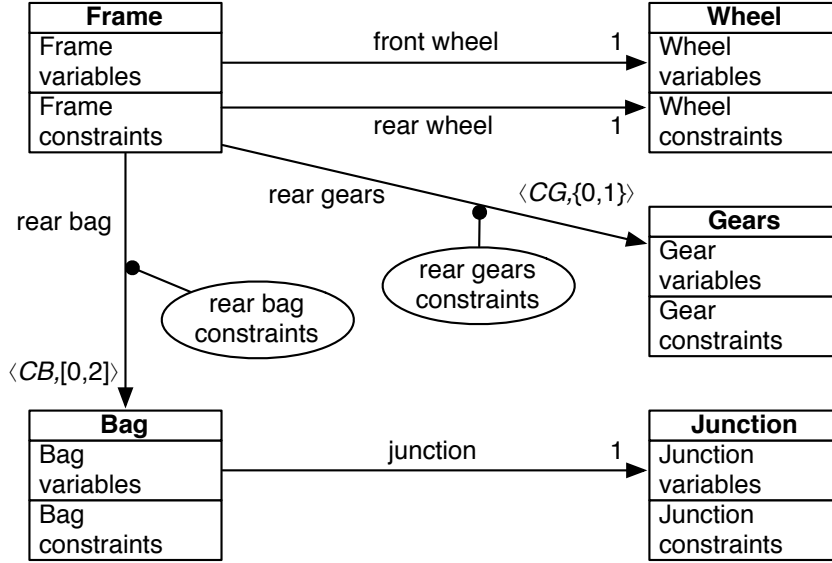


Figure 5.3: Fragment of the product model graph for a bicycle.

Plainly, an instance of an edge labeled *label* connecting a node *N* with a node *M*, is an edge connecting instances of *N* and *M*. It is labeled *label* too. Node and edge instances constitute a tree called *instance tree*, representing a configured product. The *root node* will have only one instance, it will be the root of the instance tree. An example of instance tree for the fragment of product model graph depicted in Figure 5.3, is shown in Figure 5.4.

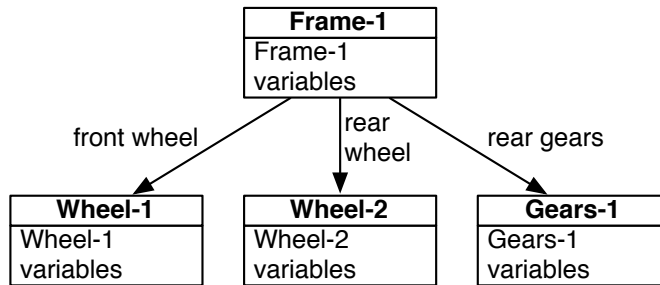


Figure 5.4: Instance tree for a bicycle.

5.2.2 Constraints on Product Characteristics

In this section we present the three types of constraints that can be used in PROMo to define compatibility relations between configurable characteristics. These types are: *node constraints*, *cardinality constraints*, and *model constraints*. Node and edge constraints allow one to model compatibility relations that are “local” with respect to *has-part* relations, while model constraints allow one to describe “global” compatibility relations.

Node constraints. A *node constraint* for the node $\langle N, \mathcal{V}, \mathcal{C} \rangle$ is a constraint on variables in \mathcal{V} , and node variables of N ’s ancestors in the product model graph. Such a constraint represents a compatibility relation between configurable characteristics of components related by *has-part* relations, and will have to hold for instances of N .

Since we may have more than one edge entering in a node in the product model graph, there could be instances of the same node reached by different paths in the instance tree. For example, in the instance tree depicted in Figure 5.4, an instance of the node *Wheel* is reached from the root by the path (i.e., a list of edge labels) $[front\ wheel]$, while the other one is reached from the root by the path $[rear\ wheel]$. Hence, when defining a node constraint we may want to refer to a variable of a particular ancestor of a node, i.e., a node constraint that will have to hold only for particular instances of a node. We can do this using *meta-paths*. A *meta-path*, represents a set of path connecting two nodes in the graph. It is a list of edge labels and wild-cards $_$ and \star . Where $_$ represents an arbitrary edge label, and \star represents an arbitrary number of edge labels.

Each node variable not belonging to N is represented in a node constraint as a tuple $\langle V, M, p \rangle$, where $\langle V, D(V) \rangle$ is a node variable of M , M is an ancestor of N , and p is a meta-path representing paths connecting M with N . In the following, we will refer to tuples of the form $\langle V, M, p \rangle$ as *meta-variables*. Each variable $\langle V, D(V) \rangle$ of N appears as V in a node constraint. For example, in a constraint for the node *Junction* the variable *FrameMaterial* of node *Frame* would be represented as follows

$$\langle FrameMaterial, Frame, [rear\ bag, junction] \rangle.$$

Intuitively, a node constraint for a node N must hold for each instance of N such that it has ancestors connected with it through paths matching with meta-paths occurring in the constraints.

Let us consider for example the node *Frame* in Figure 5.3. The section *Frame variables* may contain the variables introduced in Section 5.2.1, while *Frame constraints* may include a constraint stating that a frame of type “Racing bike” cannot be made of steel:

$$FrameType = \text{Racing bike} \Rightarrow FrameMaterial \neq \text{Steel}.$$

The values assigned to variables of the instance of node *Frame* presented in Section 5.2.1 satisfy this node constraint. For the node *Wheel* (also shown in Figure 5.3) we may have these constraints on variable *WheelType* and *SpokeNumber* (both introduced in

Section 5.2.1).

$$\begin{aligned}
 WheelType &= \langle FrameType, Frame, [_] \rangle, \\
 \langle FrameType, Frame, [rear\ wheel] \rangle &= \text{Racing bike} \Rightarrow \\
 &\Rightarrow SpokeNumber > 20.
 \end{aligned}$$

Notice that these constraints involve features belonging to an ancestor of the node *Wheel*, namely, the node *Frame*. References to variables in ancestor nodes are made by using meta-variables. Meta-paths are used to define constraints that have effect only on particular instances of a node. For example, the second constraints for the node *Wheel* has to hold only for those instances of *Wheel* that are connected to an instance of *Frame* through an edge labeled *rear wheel*. Since wildcards may appear in a meta-path, several paths in the model may match the same meta-path. Hence, several conditions among pairs of nodes may be imposed through a single constraint of this form.

Cardinality constraints. A *cardinality constraint* for an edge connecting a node *N* with a node *M*, whose cardinality is a domain endowed variable $\langle Card, D(Card) \rangle$, is a constraint on *Card*, node variables of *N*, and node variables of *N*’s ancestors in the product model graph. Such a constraint represents a compatibility relation between configurable characteristics of components related by *has-part* relations, and the structure of the product. Each domain endowed variable $\langle V, D(V) \rangle$ of *N* appears as *V* in the constraint. Each variable not belonging to *N* is represented as a meta-variable $\langle V, M, p \rangle$ in the constraint, where *M* is a node, $\langle V, D(V) \rangle$ is a node variable of *M*, and *p* is a meta-path representing paths connecting *M* with *N*. Intuitively, a cardinality constraint for an edge *e* must hold for each instance of the parent node *N* in *e*, such that *N* has ancestors connected with it through paths matching with meta-paths specified in the constraint.

As an example, let us consider the edges labeled *front wheel* and *rear gear* depicted in Figure 5.3. The former is the edge relating the frame with the front wheel, its cardinality is imposed to be 1, and there is no cardinality constraint. Hence, there must be (only) one instance of the node *Wheel* connected to an instance of the node *Frame* through an edge labeled *front wheel*. The edge *rear gears* represents the *has-part* relation between the frame and the rear gears of a bicycle. Its cardinality is a variable *CG* which takes values in the domain $\{0,1\}$. Hence, we may have an instance of the node *Gears* connected to an instance of the node *Frame* through an edge labeled *rear gears*. Among the cardinality constraints for the edge we may have the following one:

$$FrameType = \text{Racing bike} \Rightarrow CG = 1.$$

This constraint states that if the bicycle frame has type “Racing bike”, then it must have a rear gears component.

Model constraints. Node constraints and cardinality constraints allows one to define compatibility relations between configurable characteristic of product components related

by *has-part* relations only. Besides these forms of constraints, PRODPROC supports *model constraints* that may involve variables of nodes not necessary related by the *has-part* relation (*node model constraints*), or cardinalities of different edges exiting from a node (*cardinality model constraints*).

A *node model constraint* is a constraint on variables of nodes not necessary related by *has-part* relations. Each variable is represented as a meta-variable $\langle V, M, p \rangle$ in the constraint, where M is a node, $\langle V, D(V) \rangle$ is a node variable of M , and p is a meta-path representing paths reaching M . In this case, meta-paths are used to refer to a particular node in the product model graph. In general, a node model constraint must hold for all the tuples of node instances reached by paths matching the meta-paths occurring in the constraint. An example of model constraints for the bicycle is the following:

$$\begin{aligned} \langle GearType, Gears, [rear\ gears] \rangle = \text{Special} \Rightarrow \\ \Rightarrow \langle SpokeNumber, Wheel, [rear\ wheel] \rangle = 26. \end{aligned}$$

This node model constraint states that if the type of rear gears is “Special”, then the rear wheel must have 26 spokes.

Node model constraints include also the *allDifferent* constraint [vH01] and *aggregation* constraints. The *allDifferent* constraint is defined as follows:

$$allDifferent(L),$$

where L is a list of meta-variables. It is satisfied if all the involved variables assume a different values. An *aggregation* constraint has the following form:

$$aggConstraint(f, L, op, n),$$

where $f \in \{sum, avg\}$, $op \in \{<, \leq, =, \geq, >, \neq\}$, L is a list of meta-variables, and $n \in \mathbb{Z}$. It is satisfied if $f(L) op n$ holds. Intuitively, *allDifferent* and *aggregation* constraints will have to hold for the list of node instances reached by path matching with meta-paths occurring in the constraint. For example,

$$allDifferent([\langle SpokeNumber, Wheel, [_] \rangle]),$$

states that the variable *SpokeNumber* of instances of node *Wheel* must take different values.

A *cardinality model constraint* is a constraint on cardinality variables of edges exiting from a node N . Each cardinality variable is represented as a quadruple $\langle label, N, M, Card \rangle$, where $Card$ is the name of a cardinality involved in the constraint and $label$ is the string representing a description of the relation between N and M the cardinality belongs to. Cardinality model constraints may be used to express mutual exclusion relations between *has-part* relations.

5.2.3 PROMo Model

Now that we have defined product model graphs, node constraints, edge constraints, and model constraints, we are ready to give the definition of PROMo *model*.

Definition 4 (PROMo model). A PROMo model M_{Prod} is a tuple $\langle G, MC \rangle$ where:

- G is a product model graph;
- $MC = \mathcal{NMC} \cup \mathcal{CMC}$ is set of node (\mathcal{NMC}) and cardinality (\mathcal{CMC}) model constraints.

A PROMo model represents a configurable product. Its configuration can lead to the definition of different (producible) product variants. Node constraints, cardinality constraints and model constraints constitute a set called *compatibility relation pool*, that defines all the compatibility relations between characteristics of product components.

5.3 The MART Language

One of the main purposes of the PRODPROC framework is to allow the user to model aspects of a production process that may affect product configuration. In our opinion, at least two aspects must be considered: process execution times, and process resource consumption and production. Modeling these aspects one can discover during the configuration of a product the impossibility to realize the desired variant, due to lack of resources, and/or due to too strict delivery/production deadlines. PRODPROC covers these process aspects with the MART language.

The MART (*Modeling of Activities Resources and Time*) language allows one to model configurable processes through MART models, in terms of activities and temporal relations between them. Moreover, it allows one to model process resource production and consumption, and to mix product modeling with process modeling. Analogously to product models, a process model represents a configurable production process. Its configuration can lead to the definition of different executable processes. As we will see, a MART model can also be viewed as a mixed planning and scheduling problem, whose solutions represent executable processes.

5.3.1 Activities and Temporal Constraints

Prior to define the different elements of the MART language, we introduce the notion of *atomic activity*, and define *temporal constraints* between activities.

Atomic Activities. An *atomic activity* named A is an event occurring in a time interval. It has associated the following parameters.

- Two integer decision variables, t_A^{start} and t_A^{end} , denoting the start and end time of the activity. They define the time interval $[t_A^{\text{start}}, t_A^{\text{end}}]$, and are subject to the implicit condition $0 \leq t_A^{\text{start}} \leq t_A^{\text{end}}$.

- A decision variable $d_A = t_A^{end} - t_A^{start}$ denoting the duration of the activity, endowed with an integer domain $D(d_A)$.
- A flag $exec_A$ with domain $\{0,1\}$, called *activity execution flag*.

When $d_A = 0$ we say that A is an *instantaneous activity*. When $exec_A = 1$ holds A is *executed*, i.e., an instance of A will be part of an executable process. A is *not executed* if $exec_A = 0$ holds. Figure 5.5 shows the graphical representation of an activity, i.e., a box with two sections, one for the activity name and one for constraints on its duration.

Activity name
Activity duration constraints

Figure 5.5: Graphical representation for activities.

Temporal constraints. Temporal constraints between activities define temporal relations between the time intervals represented by activities. They are inductively defined starting from *atomic temporal constraints*. Let A and B be two activities. We consider the following *atomic temporal constraints*.

1. A *before* B to express that A is executed before B .
2. B *after* A to express that B is executed after A .
3. A *meets* B to express that the execution of A ends at the instant in which the execution of B starts.
4. B *met_by* A to express that the execution of B starts at the instant in which the execution of A ends.
5. A *overlaps* B to express that the execution of A ends after the execution of B has started.
6. B *overlapped_by* A to express that the execution of B starts before the execution of A ends.
7. A *during* B to express that A is executed during the execution of B .
8. B *includes* A to express that during the execution of B , A is executed.
9. A *starts* B to express that A and B start at the same instant and A ends before B ends.
10. B *started_by* A to express that B and A start at the same instant and B ends after A ends.

11. *A finishes B* to express that *A* and *B* ends at the same time instant and *A* starts after *B* has started.
12. *B finished_by A* to express that *B* and *A* end at the same time instant and *B* starts before *A* starts.
13. *A equals B* to express that *A* and *B* execute at the same time.
14. *A must_be_executed* to express that *A* must be executed.
15. *A is_absent* to express that *A* can never be executed.
16. *A not_co_existent_with B* to express that either *A* or *B* can be executed (i.e., it is not possible to execute both *A* and *B*).
17. *A succeeded_by B* to express that when *A* is executed than *B* has to be executed after *A*.

An *temporal constraint* is inductively defined as follows:

- an atomic temporal constraint is a temporal constraint;
- if φ and ϑ are temporal constraints then φ and ϑ is a temporal constraint;
- if φ and ϑ are temporal constraints then φ or ϑ is a temporal constraints;
- if φ is a temporal constraint, and c is a constraint on *process model variable* (i.e., elements of the MART language for modeling process characteristics) and activity execution flags, then:
 - $c \rightarrow \varphi$ is an *if-conditional* temporal constraint, stating that φ has to hold if c holds;
 - $c \leftrightarrow \varphi$ is an *iff-conditional* temporal constraint, stating that φ has to hold if and only if c holds.

We can associate to each atomic temporal constraint a propositional formula on activity durations, starting times, finishing times, and activity execution flags, as shown in Table 5.1.

Let φ and ϑ be temporal constraints, let $\varphi^{\mathcal{P}}$ and $\vartheta^{\mathcal{P}}$ the corresponding propositional formulas, then we have that:

- φ and ϑ corresponds to $\varphi^{\mathcal{P}} \wedge \vartheta^{\mathcal{P}}$;
- φ or ϑ corresponds to $\varphi^{\mathcal{P}} \vee \vartheta^{\mathcal{P}}$;
- $c \rightarrow \varphi$ corresponds to $c \Rightarrow \varphi^{\mathcal{P}}$;
- $c \leftrightarrow \varphi$ corresponds to $c \Leftrightarrow \varphi^{\mathcal{P}}$.

Atomic temporal constraint	Propositional formula
<i>A before B</i>	$t_A^{start} < t_B^{start} \wedge t_A^{end} < t_B^{start}$
<i>B after A</i>	
<i>A meets B</i>	$t_A^{start} < t_B^{start} \wedge t_A^{end} = t_B^{start}$
<i>B met_by A</i>	
<i>A overlaps B</i>	$t_A^{start} < t_B^{start} \wedge t_A^{end} > t_B^{start} \wedge t_A^{end} < t_B^{end}$
<i>B overlapped_by A</i>	
<i>A during B</i>	$t_A^{start} > t_B^{start} \wedge t_A^{start} < t_B^{end} \wedge t_A^{end} < t_B^{end}$
<i>B includes A</i>	
<i>A starts B</i>	$t_A^{start} = t_B^{start} \wedge t_A^{end} < t_B^{end}$
<i>B started_by A</i>	
<i>A finishes B</i>	$t_A^{start} > t_B^{start} \wedge t_A^{start} < t_B^{end} \wedge t_A^{end} = t_B^{end}$
<i>B finished_by A</i>	
<i>A equals B</i>	$t_A^{start} = t_B^{start} \wedge t_A^{end} = t_B^{end}$
<i>A must_be_executed</i>	$exec_A = 1$
<i>A is_absent</i>	$exec_A = 0$
<i>A not_co_existent_with B</i>	$exec_A + exec_B \leq 1$
<i>A succeeded_by B</i>	$exec_A = 1 \Rightarrow exec_B = 1 \wedge t_B^{start} > t_A^{end}$

Table 5.1: Atomic temporal constraints and corresponding propositional formulas.

Atomic temporal constraints 1 to 13 correspond to the thirteen mutually exclusive binary relations which capture all the possible ways two intervals can be related, such relations have been introduced by Allen in [All83]. Atomic temporal constraints 14 to 17 are inspired by some of the constraint templates of the language ConDec [PSvdA07].

Since the execution of an activity A may depend on an activation condition, i.e., we may have $Assembly = true \rightarrow A \text{ must-be-executed}$ (where $Assembly$ is a process variable), it is necessary to specify when an atomic formula from 1 to 13 has to hold. We may consider the following condition: atomic formulas 1 to 13 have to hold whenever $exec_A = 1 \wedge exec_B = 1$ holds.

An *temporal constraint network* \mathcal{CN} is a couple $\langle \mathcal{A}, \mathcal{C} \rangle$, where \mathcal{A} is a set of activities, and \mathcal{C} is a set of temporal constraints on activities in \mathcal{A} . A temporal constraint network $\mathcal{CN} = \langle \mathcal{A}, \mathcal{C} \rangle$ can be represented with a hyper-graph $G_{\mathcal{CN}} = \langle V, E \rangle$, where each activity in \mathcal{A} corresponds to a vertex in V , and each formula in \mathcal{C} corresponds to a hyper-edge in E ,

connecting the vertex corresponding to the activities involved in the formula, and labeled with the formula itself. More precisely, we can give a graphical representation to each temporal constraint, as shown in Figure 5.6. In particular, we represent binary atomic temporal constraints as arrows labeled with atomic constraints operators. Examples of such graphical representations are given in Figure 5.6a. Figure 5.6b and Figure 5.6c depict the *must_be_executed* constraint and the *is_absent* constraint, respectively. Moreover, Figure 5.6c and 5.6d show the graphical representation for iff-conditional and if-conditional temporal constraints with condition c , respectively. A conjunction of atomic constraints between two activities can be graphically represented simply representing each constraint in the conjunction. The graphical representation for a non-atomic temporal constraint φ is shown in Figure 5.6e.

Figure 5.7 shows examples of atomic activities and temporal constraints for the bicycle production process. In it, activities are depicted as boxes containing only activity names. We will use this graphical representation when not interested in showing constraints on activity durations. The process depicted in Figure 5.7 consists of three atomic activities, namely, “Construction of bicycle components”, “Bicycle assembly”, and “Delivery”. “Construction of bicycle components” is executed before “Bicycle assembly”, and “Bicycle assembly” is executed before “Delivery”. Finally, “Bicycle assembly” is not executed when the condition $Assembly = 0$ holds, otherwise it is executed.

5.3.2 MART language elements

Atomic activities and temporal constraints are not the only elements of the MART language. In the following we describe all the other features that can be used to model production processes.

Model variables. A *model variable* is a domain endowed variable, i.e., it has the form $\langle V, D(V) \rangle$. We consider two classes of model variables:

- *Resource variables* are integer variables, i.e., $V \in \mathcal{V}_{\text{int}}$, modeling resources (e.g., number of workers, quantity of steel, etc.);
- *Process variables* are variables modeling process characteristics (e.g., flags for controlling the presence or absence of activities).

The variable *Assembly* that appears in Figure 5.7 is an example of model variable. In particular, it is a process variable.

Resource constraints. A *resource constraint* [Lab03] defines how a given activity A will require and affect the availability of a given resource R (R is a resource variable name). It consists of a tuple $\langle A, R, q, TE \rangle$, with $q \in \mathbb{Z}$, or $\langle A, R, \langle q, D(q) \rangle, TE \rangle$, with $q \in \mathcal{V}_{\text{int}}$. q defines the quantity of resource R consumed (if $q < 0$) or produced (if $q > 0$) by activity A , and TE is a time extent that defines the time interval where the availability of resource R is affected by the execution of activity A . The possible time extents are: *FromStartToEnd*, *AfterStart*, *AfterEnd*, *BeforeStart*, *BeforeEnd*, *Always*, with the obvious intuitive

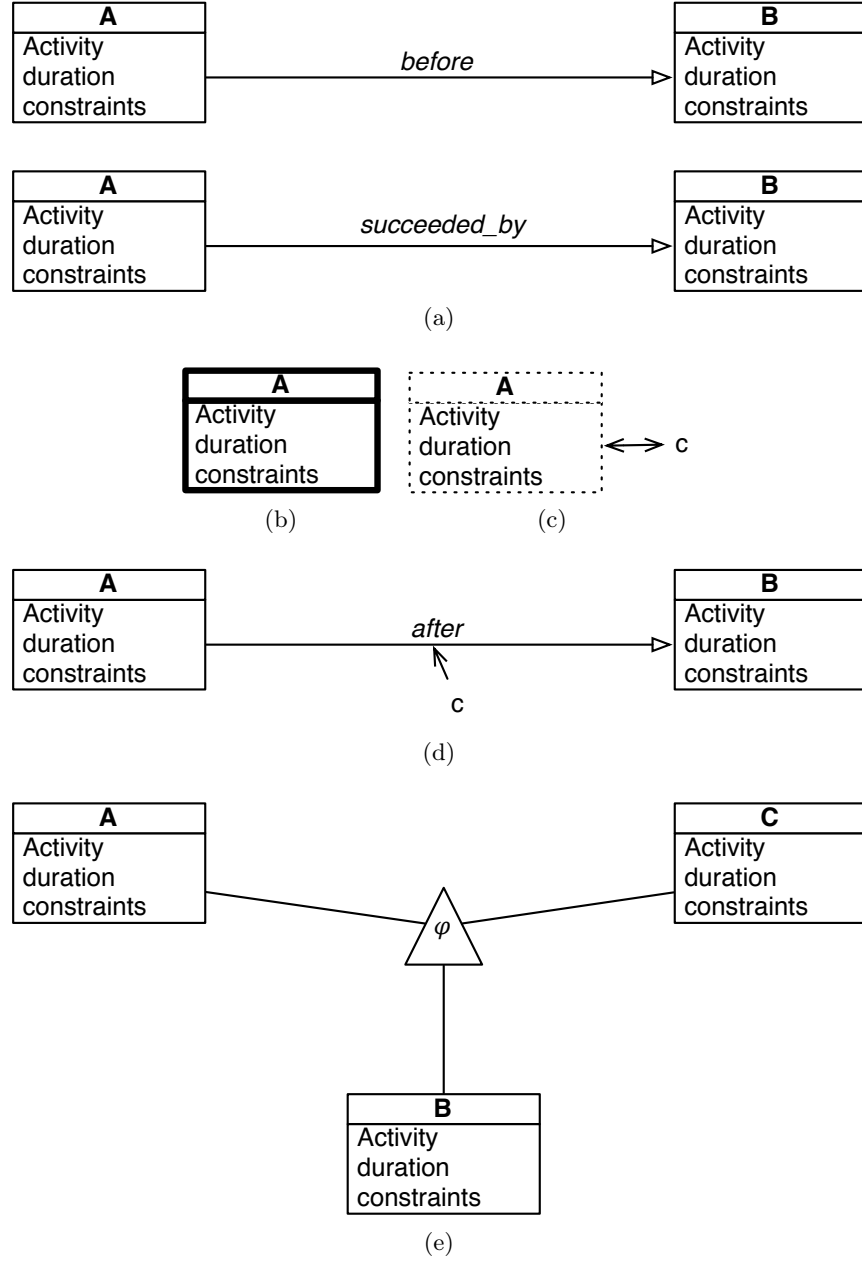


Figure 5.6: Graphical representations for temporal constraints.

meaning. Given a domain endowed variable $\langle q, D(q) \rangle$ occurring in a resource constraint, we refer to q as *resource quantity variable*. We assume that activities involved in resource constraints are not instantaneous, i.e., their duration is strictly greater than zero. The

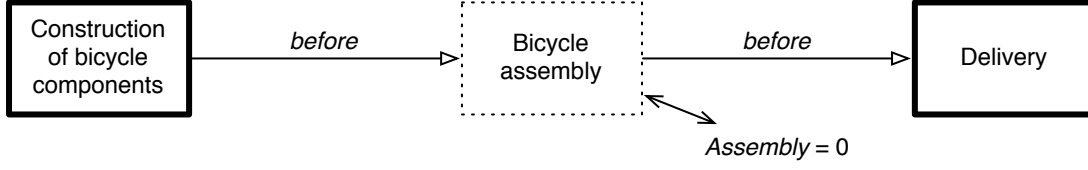


Figure 5.7: Examples of atomic activities and temporal constraints.

following are two resource constraints for the process depicted in Figure 5.7:

$$\begin{aligned} &\langle \text{Bicycle assembly}, \text{Worker}, \langle q_{\text{Worker}}, [-2, -1] \rangle, \text{FromStartToEnd} \rangle, \\ &\langle \text{Delivery}, \text{Worker}, -1, \text{FromStartToEnd} \rangle, \end{aligned}$$

where *Worker* is a resource. The first constraint states that the activity “Bicycle assembly” will require one or two unit of resource *Worker* between its start time and its end time. Thus, the availability of *Worker* will decrease at the start time of “Bicycle assembly”, and will increase at its end time, when “Bicycle assembly” releases *Worker*. The second constraints states that activity “Delivery” will require exactly one unit of resource *Worker* between its start time and its end time.

An *initial level constraint* for a resource *R* defines the quantity of resource *R* that will be available at the time origin of the process. It has the form $\text{initialLevel}(R, iv)$, where $iv \in \mathbb{N}$.

Analogously to the case of temporal constraints, *if-conditionals* (of the form $c \rightarrow \langle A, R, q, TE \rangle$) and *iff-conditionals* (of the form $c \leftrightarrow \langle A, R, q, TE \rangle$) are introduced for resource constraints as well.

Resource constraints may implicitly define temporal constraints between activities. Let us consider the following couple of resource constraints:

$$\langle A, R, +2, \text{AfterEnd} \rangle, \quad \langle B, R, -2, \text{AfterStart} \rangle.$$

Activity *B* can start only when 2 units of resource *R* are available, if *R* is produced only by *A* and its initial value is 0, then the execution of *B* can start only after the end of the execution of *A*.

Figure 5.8 shows the graphical representation for a resource *R*, and for the constraints $\text{initialLevel}(R, iv)$ and $\langle A, R, q_A, TE_A \rangle$. The resource is represented as a polygon with six sides containing *R* and *iv*. A resource constraint is depicted as a straight line, labeled with the quantity and the time extent, connecting the activity with the resource. The graphical representations of if- and iff- conditional resource constraints are similar to the ones defined for temporal constraints in Figure 5.6.

Activity duration constraints. An *activity duration constraint* for the activity *A* is a constraint on d_A , process variable names, and resource quantity variables of resource constraints for *A*. The following is an example of duration constraint for the activity

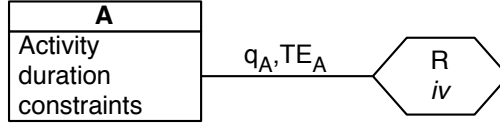


Figure 5.8: Graphical representations for resources and resource constraints.

“Bicycle assembly”.

$$d_{\text{Bicycle assembly}} = \frac{3 \cdot \text{Components}}{|q_{\text{Worker}}|},$$

where *Components* is a process variable name.

Multiple instance activities. A *multiple instance activity* named *A* is an event corresponding to a time interval, that may occur multiple times. It has associated the following parameters:

- A domain endowed variable $\langle inst_A, D(inst_A) \rangle$, representing the number of instances of *A* that will be executed, i.e., that will be part of an executable process.
- $inst_A$ couple of decision variables, t_A^{start} and t_A^{end} , denoting the start and end time of activity instances, and defining the interval $[t_A^{start}, t_A^{end}]$ ($t_A^{end} \geq t_A^{start} \geq 0$).
- $inst_A$ decision variables $d_A = t_A^{end} - t_A^{start}$ denoting the duration of activity instances, endowed with an integer domain $D(d_A)$.
- A flag $exec_A \in \{0,1\}$.

$exec_A = 1$ holds when $inst_A > 0$ and $inst_A$ instances of *A* are executed, if $exec_A = 0$ holds then no instance of *A* is executed. Clearly, an atomic activity *B* can be viewed as a multiple instance activity with $D(inst_B) = \{0,1\}$. Figure 5.9 shows the graphical representation for multiple instance activities.

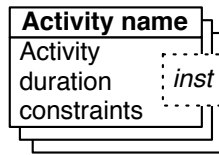


Figure 5.9: Graphical representation for multiple instance activities.

Resource constraints may implicitly define constraints on the number of instances of multiple instance activities. Let us consider the following couple of resource constraints on the multiple instance activity *A*, and the atomic activity *B*.

$$\langle A, R, +1, AfterEnd \rangle, \quad \langle B, R, -4, AfterStart \rangle.$$

Activity *B* needs 4 units of resource *R* to be executed, if *R* is produced only by *A* and its initial value is 0, then 4 instances of *A* must be executed to execute *B*. In general, given

$\langle A, R, q, TE \rangle$, where $q \in \mathbb{N}^+$ or $q = \langle V, D(V) \rangle$, $D(V) \subset \mathbb{N}^+$, and A is a multiple instance activity, we can consider an implicit constraints of the following form:

$$inst_A \geq \frac{\sum_{\langle I_C, c \rangle \in Cons(R)} (I_C \cdot c) - v - \sum_{\langle I_P, p \rangle \in Prod(R)} (I_P \cdot p)}{k},$$

where $k = q$ if $q \in \mathbb{N}^+$, or $k = \min(D(V))$ if $q = \langle V, D(V) \rangle$, $v = iv$ if there exists a constraint $initialLevel(R, iv)$, or $v = \min(D(R))$, and the sets $Cons(R)$ and $Prod(R)$ are defined as follows:

$$\begin{aligned} Cons(R) &= \{ \langle I_C, c \rangle \mid \exists \langle C, R, con, TE \rangle \wedge \\ &\quad \wedge ((con \in \mathbb{Z}^{<0} \wedge c = |con|) \vee (con \subseteq \mathbb{Z}^{<0} \wedge c = |\max_{v \in con}(v)|)) \wedge \\ &\quad \wedge ((C \text{ multiple instance activity} \wedge I_C = inst_C) \vee I_C = exec_C) \}, \\ Prod(R) &= \{ \langle I_P, p \rangle \mid \exists \langle P, R, prod, TE \rangle \wedge \\ &\quad \wedge ((prod \in \mathbb{N}^+ \wedge p = prod) \vee (prod \subseteq \mathbb{N}^+ \wedge c = \min_{v \in prod}(v))) \wedge \\ &\quad \wedge ((P \text{ multiple instance activity} \wedge I_P = inst_P) \vee I_P = exec_P) \}. \end{aligned}$$

Coupling constraints. *Coupling constraints* allows one to define constraints involving product model variables and process model variables.

Let M_{Prod} be a product model. Let $\mathcal{V}_{M_{Prod}} = \mathcal{V}_{Cardinalities} \cup \mathcal{V}_{Nodes}$, where $\mathcal{V}_{Cardinalities}$ is the set of cardinality variables in M_{Prod} , and \mathcal{V}_{Nodes} is the set of node variables in M_{Prod} . Let \mathcal{V}_{Proc} be a set of model variables and multiple instance activity instance number variables.

A *coupling constraint* is an equality constraint on variables in the set $\mathcal{V}_{M_{Prod}} \cup \mathcal{V}_{Proc}$. A node variable is represented as a tuple $\langle V, M, p \rangle$ in the constraint, where M is a node, $\langle V, D(V) \rangle$ is a node variable of M , and p is a meta-path representing paths reaching M . A cardinality variable is represented as a quadruple $\langle label, N, M, Card \rangle$, where $Card$ is the name of a cardinality involved in the constraint, $label$ is the string representing a description of the relation between N and M the cardinality belongs to, and N is the parent node in the relation. Variables in \mathcal{V}_{Proc} are referred to by their names in a constraint. The following is an example of a coupling constraint:

$$\langle FrameType, Frame, [] \rangle = \text{Racing bike} \Rightarrow Components = 8,$$

This constraint states that if the bicycle frame has type “Racing bike”, then the process variable *Components* takes value 8. A set of coupling constraints defines a coupling of M_{Prod} with M_{Proc} .

Product related constraints. *Product related constraints* allows one to define in a simple way resource constraints where the considered resources are product components. A *product related constraint* is a constraint on activities and product model nodes, that implicitly defines resource and other constraints. We consider two product related constraints, namely, *produces for* and *needs from*.

A *produces for* constraint is an expression of the following form:

$$A \text{ produces } n \text{ } N \text{ for } B,$$

where A and B are activities, $n \in \mathbb{N}^+$, and N is a node of the product model graph having (at least) one incoming edge having associated a cardinality variable. Such a constraint corresponds to the following constraints:

$$\begin{aligned} &\langle A, Resource_N, \langle q_A, D(Resource_N) \rangle, AfterEnd \rangle, \\ &\langle B, Resource_N, -n, AfterStart \rangle, \\ &initialLevel(Resource_N, 0), \\ &\sum_{C \in CE_N} C \geq \sum_{A \text{ produces } N} q_A, \end{aligned}$$

where $Resource_N$ is a resource variable whose domain $D(Resource_N)$ is defined as

$$D(Resource_N) = \left[0, \sum_{C \in CE_N} \max(D(C)) \right],$$

and CE_N is the list of cardinality variables associated to edges entering in N .

If A is a multiple instance activity, then each of its instances will contribute in producing $Resource_N$, i.e., each instance i of A will produce $q_i \in D(Resource_N)$ unity of $Resource_N$. If B is a multiple instance activity, then each of its instances will consume n unity of $Resource_N$.

The following is an example of *produces for* constraint:

Wheel construction *produces* 2 Wheel *for* Bicycle assembly ,

where “Wheel” is a product model node, and “Wheel construction” and “Bicycle assembly” are activities.

A *needs from* constraint is an expression of the following form:

$$B \text{ needs } n \text{ } N \text{ from } A,$$

where A and B are activities, $n \in \mathbb{N}$, and N is a node of the product model graph having (at least) one incoming edge having associated a cardinality variable. Such a constraint is equivalent to the constraint $A \text{ produces } n \text{ } N \text{ for } B$.

The following are two examples of *needs from* constraint:

Car wheel assembly *needs* 4 Wheel *from* Wheel construction,

Truck wheel assembly *needs* 4 Wheel *from* Wheel construction,

where “Wheel construction”, “Car assembly” and “Truck assembly” are activities, and “Wheel” is a product model node.

A graphical representation for the constraint $A \text{ produces } n \text{ } N \text{ for } B$ (and the equivalent constraint $B \text{ needs } n \text{ } N \text{ from } A$) is shown in Figure 5.10.

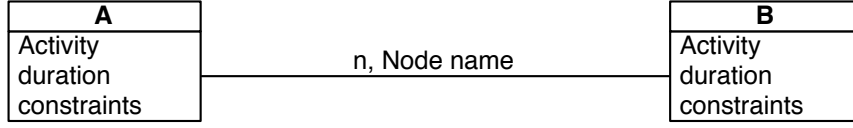


Figure 5.10: Graphical representation for product related constraints.

5.3.3 MART Model and Composite Activities

Now that we have presented all the features of the MART language we are ready to give the definition of MART *model*.

Definition 5 (MART model). A MART model M_{Proc} is a tuple $\langle \mathcal{A}, \mathcal{V}, \mathcal{CN}, \mathcal{P}, \mathcal{R}, \mathcal{D} \rangle$, where:

- \mathcal{A} is a set of atomic activities, multiple instance activities, composite activities, and multiple instance composite activities (composite activities will be defined next).
- $\mathcal{V} = \mathcal{V}_{\text{Resources}} \cup \mathcal{V}_{\text{Proc}}$ is a set of model variables.
- $\mathcal{CN} = \langle \mathcal{A}, \mathcal{C} \rangle$ is a temporal constraint network, where \mathcal{C} is a set of temporal constraint on activities in \mathcal{A} .
- \mathcal{P} is a set of product related constraint.
- $\mathcal{R} = \bigcup_{A \in \mathcal{A}} \mathcal{R}_A$, where \mathcal{R}_A is a set of resource constraints on activity A and resources in $\mathcal{V}_{\text{Resources}}$.
- $\mathcal{D} = \bigcup_{A \in \mathcal{A}} \mathcal{D}_A$, where \mathcal{D}_A is a set of duration constraints on d_A and variables in \mathcal{V} .

A MART model does not represent a single production process. Instead, it represents a configurable production process, whose configuration can lead to the definition of different executable processes. A MART model can be also thought as a mixed scheduling an planning problem. In fact, we may have activities whose execution is not certain, and activities may produce and/or consume resources.

Composite activities. A *composite activity* is an event described in terms of a sub-process. A composite activity A of a model $M_{\text{Proc}} = \langle \mathcal{A}, \mathcal{V}, \mathcal{CN}, \mathcal{R}, \mathcal{D} \rangle$ has associated the following elements:

- A MART model M_{Proc}^A that “inherits” the model variables of M_{Proc} , defined as $M_{\text{Proc}}^A = \langle \mathcal{A}_A, \mathcal{V}_A \cup \mathcal{V}, \mathcal{CN}_A, \mathcal{R}_A, \mathcal{D}_A \rangle$.
- Two integer decision variables, t_A^{start} and t_A^{end} , denoting the start and end time of the composite activity, and defining the time interval $[t_A^{\text{start}}, t_A^{\text{end}}]$. These time instants are defined as follows:

$$\begin{aligned} t_A^{\text{start}} &= \min_{B \in \mathcal{A}_A} t_B^{\text{start}}, \\ t_A^{\text{end}} &= \max_{B \in \mathcal{A}_A} t_B^{\text{end}}. \end{aligned}$$

- An integer decision variable $d_A = t_A^{end} - t_A^{start}$, with domain $D(d_A)$.
- A flag $exec_A \in \{0,1\}$.

When $d_A = 0$ we say that A is an *instantaneous composite activity*. When $exec_A = 1$ holds A is *executed*, A is *not executed* if $exec_A = 0$ holds. Figure 5.11 shows the graphical representation for composite activities.

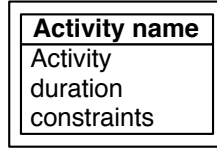


Figure 5.11: Graphical representation for composite activities.

A *multiple instance composite activity* A is a composite activity having associated also a variable $inst_A$ endowed with an integer domain $D(inst_A)$, representing the number of instances of A that will be executed.

Figure 5.12 shows a different version of the simple constraint network depicted in Figure 5.7, where the activity named “Construction of bicycle components” is modeled by a composite activity instead of an atomic activity. The constraint network of the MART model of the composite activity is the one inside the dashed line box. The absence of temporal constraints involving activities “Wheels construction” and “Construction of other components” means that the two activities can be executed independently to the others.

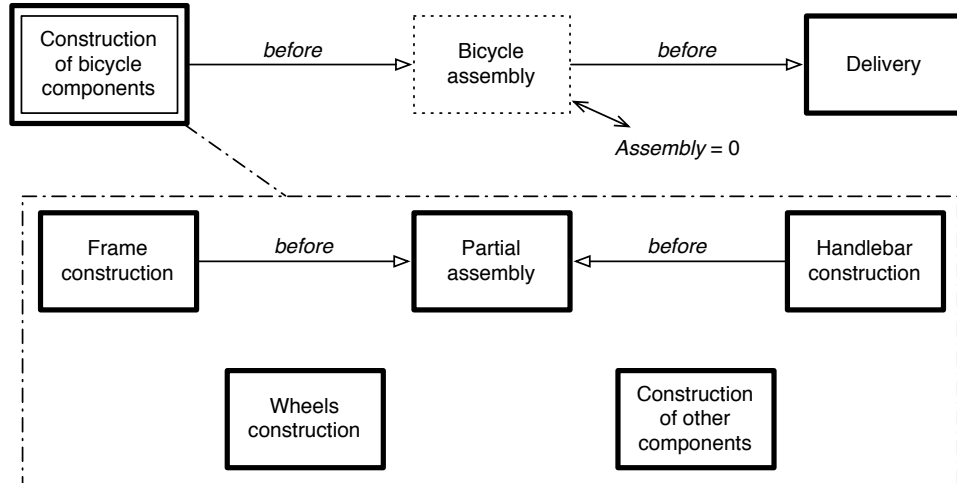


Figure 5.12: Example of composite activity usage.

5.4 PRODPROC Models and Instances

A PRODPROC model M_{ProdProc} is a tuple

$$\langle M_{\text{Prod}}, M_{\text{Proc}}, \mathcal{C}_{\text{Coupl}} \rangle$$

where M_{Prod} is a PROMO model, M_{Proc} is a MART model, and $\mathcal{C}_{\text{Coupl}}$ is a set of coupling constraint defining the coupling of M_{Prod} with M_{Proc} . A PRODPROC model represents the collection of single (producible) variants of a configurable product and the processes to produce them. A PRODPROC *instance* represents one of such product variants and its production process. In this section we describe the product and process instances that can be obtained from PROMO models and MART models, respectively, through the configuration task. Moreover, we show how such instances can be coupled to obtain a PRODPROC instance from a PRODPROC model.

The configuration task consists of finding a *configured product*, that is, a set of customized components together with a description of their relationships, that satisfies all user's preferences on product characteristics and all the relations defined by the product model, and an *executable process* for its production. This task may be accomplished by the user using a configuration system, through the *configuration process*. During this interactive process, the user (i) selects the components that will compose the configured product, (ii) chooses suitable values for their configurable characteristics, (iii) selects activities to be executed, (iv) chooses suitable values for process model parameters. The configuration system supports the user checking the validity of user's choices with respect to the product model, and reacting to user's inputs by propagating their effects. The configuration task may be automatically accomplished by the configuration system too.

5.4.1 ProMo Instances

Given the definition of product model presented in Section 5.2.3, a configured product can be represented with a PROMO *instance*. To define the notion of PROMO instance we need to introduce the notion of PROMO *candidate instance*. A PROMO candidate instance consists of the following components.

- A set \mathcal{N} of *node instances*, i.e., tuples of the form $\langle N, i, \mathcal{V} \rangle$ where N is a node in the product model tree, $i \in \mathbb{N}$ is an id (different for each instance of a node), \mathcal{V} is the set of node variables of N . In the following, we will denote with n an instance of a node N .
- A set $\mathcal{A}_{\text{Nodes}}$ of *assignments* for all the node instance variables, i.e., expressions of the form $V = \text{value}$ where V is a variable of node instance n and $\text{value} \in D(V)$.
- A tree, called *instance tree*, that specifies the pairs of node instances in the relation *has-part*. Such a tree is defined as $IT = \langle \mathcal{N}, \mathcal{E} \rangle$, where \mathcal{E} is a set of tuples of the form $\langle \text{label}, n, m \rangle$ such that there exists an edge $\langle \text{label}, N, M, \text{Card}, \mathcal{CC} \rangle$ in the product model tree, n is an instance of N and m is an instance of M .

- A set $\mathcal{A}_{\text{Cards}}$ of *assignments* for all the instance cardinality variables, i.e., expressions of the form $IC_n^e = k$, where: n is an instance of a node N , $e = \langle \text{label}, N, M, \text{Card}, \text{CC} \rangle$, $IC_n^e \equiv \text{Card}$, and $k \in D(\text{Card})$ is the number of edges $\langle \text{label}, n, m \rangle$, where m is an instance of M in the instance tree.

We represent a PROMO candidate instance with a couple $\langle IT, \mathcal{A} \rangle$, where $\mathcal{A} = \mathcal{A}_{\text{Nodes}} \cup \mathcal{A}_{\text{Cards}}$. In the following, we will use the term *partial candidate PROMO instance* to indicate a couple $\langle IT, \tilde{\mathcal{A}} \rangle$ where $\tilde{\mathcal{A}} = \tilde{\mathcal{A}}_{\text{Nodes}} \cup \mathcal{A}_{\text{Cards}}$ and $\tilde{\mathcal{A}}_{\text{Nodes}}$ is a set of assignments for a subset of the node instance variables, that is, to refer to a candidate instance under construction.

A PROMO instance is a candidate instance such that the assignments in \mathcal{A} satisfy all the elements of the compatibility relation pool, appropriately instantiated with variables of node instances in the candidate instance (as a matter of fact, the choices of the values for the instance variables can be viewed as primitive constraints).

The (constraint) instantiation mechanism yielding an instance, produces a set of constraints on candidate instance variables from each constraint in the PRODPROC model. A candidate instance must satisfy all these constraints to qualify as an instance. We give here an intuitive description of how the instantiation mechanism works on different constraint types. Let us begin with node and cardinality constraints. Let c be a constraint belonging to the node N , or a constraint for an edge e between nodes N and M . Let us suppose that N_1, \dots, N_k are ancestors of N whose variables are involved in c , and let p_1, \dots, p_k be meta-paths such that, for $i = 1, \dots, k$, p_i is a meta-path from N_i to N . We define L_{node} as the set of k -tuple of node instances $\langle n, n_1, \dots, n_k \rangle$ where: n is an instance of N ; for $i = 1, \dots, k$ n_i is an instance of N_i , connected with n through a path q_i in the instance tree that matches with p_i . For each k -tuple $t \in L_{\text{node}}$, we obtain a constraint on instance variables appropriately substituting variables in c with variables of node instances in t . For example, let us consider the fragment of the instance tree for a bicycle depicted in Figure 5.4. It consists of an instance of node *Frame*, an instance of the node *Gears*, and two instance of node *Wheel*. The instantiation of the constraints

$$\text{WheelType} = \langle \text{FrameType}, \text{Frame}, [_] \rangle, \quad (5.1)$$

$$\begin{aligned} \langle \text{FrameType}, \text{Frame}, [\text{rear wheel}] \rangle &= \text{Racing bike} \Rightarrow \\ &\Rightarrow \text{SpokeNumber} > 20, \end{aligned} \quad (5.2)$$

for the node *Wheel* of the bicycle product model graph, leads to the following constraints on variables of node instances in Figure 5.4. ($\langle V, n_i \rangle$ denotes the variable V of the instance with id i of node N).

$$\langle \text{WheelType}, \text{Wheel}_1 \rangle = \langle \text{FrameType}, \text{Frame}_1 \rangle, \quad (5.3)$$

$$\langle \text{WheelType}, \text{Wheel}_2 \rangle = \langle \text{FrameType}, \text{Frame}_1 \rangle, \quad (5.4)$$

$$\begin{aligned} \langle \text{FrameType}, \text{Frame}_1 \rangle &= \text{Racing bike} \Rightarrow \\ &\Rightarrow \langle \text{SpokeNumber}, \text{Wheel}_2 \rangle > 20. \end{aligned} \quad (5.5)$$

The instantiation of (5.1) leads to the constraints (5.3) and (5.4), since it can be instantiated on the couples of node instances appearing in Figure 5.4 $\langle \text{Wheel}_1, \text{Frame}_1 \rangle$ and

$\langle Wheel_2, Frame_1 \rangle$. Instead, the instantiation of (5.2) leads to only one constraint, i.e. (5.5), because it can be instantiated only on the couple $\langle Wheel_2, Frame_1 \rangle$.

Node model constraints are instantiated in a slightly different way. Let c be a node model constraint. Let us suppose that N_1, \dots, N_k are the nodes whose variables are involved in c , let p_1, \dots, p_k be meta-paths such that, for $i = 1, \dots, k$, p_i is a meta-path that ends in N_i . We define L_{nmc} as the set of ordered k -tuples of node instances $\langle n_1, \dots, n_k \rangle$, where for $i = 1, \dots, k$ n_i is an instance of N_i connected by a path q_i with one of its ancestors in the instance tree, such that q_i matches with p_i . For each k -tuple $t \in L_{nmc}$, we obtain a constraint on instance variables appropriately substituting variables in c with variables of node instances in t . If c is an *allDifferent* or an aggregation constraint, then we define an equivalent constraint on the list consisting of all the node instances of N_1, \dots, N_k , connected with one of their ancestors by a path matching with the corresponding meta-path. As an example, let us consider the following constraint for the bicycle

$$\begin{aligned} \langle GearType, Gears, [rear\ gears] \rangle = \text{Special} \Rightarrow \\ \Rightarrow \langle SpokeNumber, Wheel, [rear\ wheel] \rangle = 26. \end{aligned}$$

It can be instantiated on the couple of node instances $\langle Wheel_2, Gears_1 \rangle$ and leads to the constraint

$$\langle GearType, Gears_1 \rangle = \text{Special} \Rightarrow \langle SpokeNumber, Wheel_2 \rangle = 26.$$

The instantiation of cardinality model constraints is very simple. Let c be a cardinality model constraint for the cardinalities of the edges with labels e_1, \dots, e_k exiting from a node N . Let n_1, \dots, n_h be instances of N . For all $i \in \{1, \dots, h\}$, we instantiate c appropriately substituting the cardinality variables occurring in it, with the instance cardinality variables $IC_{n_i}^{e_1}, \dots, IC_{n_i}^{e_k}$.

Formally, we define a function μ_{Prod} that, given the instance tree IT and a constraint c of the compatibility relation pool, instantiates c by generating a set of constraints on variables of node instances. To define μ_{Prod} , we preliminarily need to introduce some basic notions. For any variable name V , let N_V (resp. n_V) be the PROMO (resp. instance) node such that V is a variable of N (resp. n). Moreover, for any constraint c , let $\text{vars}(c)$ be the list of node variable names and meta-variables occurring in c , and let $\text{nodesPaths}(c) = \{ \langle N, p \rangle \mid \langle V, N, p \rangle \in \text{vars}(c) \}$. If c is a cardinality model constraint $\text{cards}(c)$ is the list of cardinalities in c , $\text{edgect}(\langle label, N, M, Card \rangle) = e$ such that e is the edge $\langle label, N, M, Card, CC \rangle$. Finally, given a list L and an integer i , let $\text{nth}(L, i)$ be the i -th element of L . Given two list L_1 and L_2 , $L_1 \circ L_2$ denote the concatenation of the two lists. We denote with $[x|L]$ the list obtained prepending the element x to the list L . Given a node N of the product model graph and a node n of the instance tree, we say that $n \leftrightarrow_{\text{Node}} N$ if and only if n is an instance of N (correspondence between nodes and node instances). We say that $V \leftrightarrow_{\text{Var}} W$ if V is the name of a node instance variable of n , W represents a node variable W_N of the PROMO node N such that $\langle V, D(V) \rangle = W_N$, and $n \leftrightarrow_{\text{Node}} N$. Given an edge e between two nodes N and M of the product model graph, and an edge f between two nodes n and m of the instance tree, we say that $f \leftrightarrow_{\text{Edge}} e$ if

and only if $n \leftrightarrow_{\text{Node}} N$ and $m \leftrightarrow_{\text{Node}} M$ (correspondence between product model graph edges and instance tree edges). We define four functions on meta-paths and paths in the instance tree IT . The function **apath** returns the path in the instance tree IT connecting the node instance m with n , if the former is an ancestor of the latter, otherwise it returns *false*.

$$\text{apath}(n, m, IT) = \begin{cases} q & \text{if } m \text{ ancestor of } n \text{ in } IT \text{ and} \\ & q \text{ connects } m \text{ with } n \\ false & \text{otherwise} \end{cases}$$

The function **match** checks if a path q in the instance tree matches with a meta-path p .

$$\text{match}(q, p) = \begin{cases} true & \text{if } q = p \\ \text{match}(qs, mps) & \text{if } q = [label|qs] \wedge (p = [label|mps] \vee p = [_|mps]) \\ \text{match}(qs, mps) & \text{if } p = [\star, label|mps] \wedge \exists Sub.(q = Sub \circ [label|qs]) \\ false & \text{otherwise} \end{cases}$$

The function **mapath** computes the set of paths in the instance tree IT , matching with the meta-path p , and connecting instances of the PROMO node M with the node instance n .

$$\text{mapath}(n, p, M, IT) = \{q \mid \text{apath}(n, m, IT) = q \wedge m \leftrightarrow_{\text{Node}} M \wedge \text{match}(q, p)\}.$$

The function **ancp** determines if there exists a path in the instance tree IT connecting the node instance n with one of its ancestors, that matches with the meta-path p .

$$\text{ancp}(n, p, IT) = \begin{cases} true & \text{if } \exists m \in IT. (\text{apath}(m, n) = P \wedge \text{match}(q, p)) \\ false & \text{otherwise} \end{cases}$$

We define three functions to determine if a constraint c can be instantiated on node instance variables of a given instance tree IT . The function **inst_{nec}** determines if a node or cardinality constraint c related to a node N can be instantiated. Intuitively, c can be instantiated if there exists an instance n of N and, for each couple $\langle M, p \rangle$ (where M is a node and p a meta-path) such that there is a meta-variable $\langle V, M, p \rangle$ occurring in c , there exists an instance of M connected with n through a path matching with p .

$$\text{inst}_{\text{nec}}(c, N, IT) = \begin{cases} true & \text{if } \exists n \in IT. \left(n \leftrightarrow_{\text{Node}} N \wedge \forall \langle M, p \rangle \in \text{nodesPaths}(c) \right. \\ & \quad \left. \exists m \in IT. (m \leftrightarrow_{\text{Node}} M \wedge \text{apath}(m, n) = q \wedge \right. \\ & \quad \left. \wedge \text{match}(q, p) = true) \right) \\ false & \text{otherwise} \end{cases}$$

The function **inst_{nmc}** determines if a node model constraint c can be instantiated. Intuitively, a node model constraint c , that is not an *allDifferent* constraint nor an aggregate

constraint, can be instantiated if for each couple $\langle M, p \rangle$ (where M is a node and p a meta-path) such that there is a meta-variable $\langle V, M, p \rangle$ occurring in c , there exists an instance of N reached by a path ending with a sequence of labels matching with p . If c is an *allDifferent* or an aggregate constraint, we require only the existence of at least a couple node N , meta-path p for which there exists an instance of N reached by a path matching with p .

$$\text{inst}_{\text{nmc}}(c, IT) = \begin{cases} \text{true} & \text{if } c \not\equiv \text{allDifferent}(V) \wedge c \not\equiv \text{aggConstraint}(f, v, op, n) \wedge \\ & \wedge \forall \langle N, p \rangle \in \text{nodesPaths}(c) \exists n \in IT. (n \leftrightarrow_{\text{Node}} N \wedge \\ & \wedge \text{ancp}(n, p, IT) = \text{true}) \\ \text{true} & \text{if } (c \equiv \text{allDifferent}(V) \vee c \equiv \text{aggConstraint}(f, v, op, n)) \wedge \\ & \wedge \exists \langle N, p \rangle \in \text{nodesPaths}(c) \exists n \in IT. (n \leftrightarrow_{\text{Node}} N \wedge \\ & \wedge \text{ancp}(n, p, IT) = \text{true}) \\ \text{false} & \text{otherwise} \end{cases}$$

The function inst_{cmc} determines if a cardinality model constraint c can be instantiated. Intuitively, c can be instantiated if there exists an instance of the node whose exiting edge cardinalities are involved in c .

$$\text{inst}_{\text{cmc}}(c, IT) = \begin{cases} \text{true} & \text{if } \exists n \in IT. (n \leftrightarrow_{\text{Node}} N \wedge N = \text{node}(c)) \\ \text{false} & \text{otherwise} \end{cases}$$

$\text{node}(c)$ is the node whose exiting edge cardinalities are involved in c .

Let $M_{Prod} = \langle G, \mathcal{NMC} \cup \mathcal{CMC} \rangle$ be a PROMO model, and let IT be an instance tree, the function μ_{Prod} is defined as follows.

$$\mu_{Prod}(c, IT) = \left\{ \begin{array}{ll} \{c[V_1/U_1, \dots, V_h/U_h] \mid & \text{if } c \in \mathcal{C}_N \text{ for } N \in G \wedge \\ [V_1, \dots, V_h] = \text{vars}(c), & \wedge \text{inst}_{nec}(c, N, IT) = \text{true} \\ [U_1, \dots, U_h] \in L_{\text{node}}(c, N, IT)\} & \\ \\ \{c[V_1/U_1, \dots, V_h/U_h, C/IC] \mid & \text{if } c \in \mathcal{CC} \text{ for } e \in G \wedge \\ [V_1, \dots, V_h] = \text{vars}(c), & \wedge e = \langle l, N, M, C, \mathcal{CC} \rangle \wedge \\ [U_1, \dots, U_h, IC] \in L_{\text{edge}}(c, e, N, IT)\} & \wedge \text{inst}_{nec}(c, N, IT) = \text{true} \\ \\ \delta(c, IT) & \text{if } c \in \mathcal{NMC} \wedge \\ & \wedge \text{inst}_{nmc}(c, IT) = \text{true} \\ \\ \{c[C_1/IC_1, \dots, C_h/IC_h] \mid & \text{if } c \in \mathcal{CMC} \wedge \\ [C_1, \dots, C_h] = \text{cards}(c), & \wedge \text{inst}_{cmc}(c, IT) = \text{true} \\ [IC_1, \dots, IC_h] \in L_{\text{cmc}}(c, IT)\} & \end{array} \right.$$

where:

$$L_{\text{node}}(c, N, IT) = \left\{ [U_1, \dots, U_h] \mid \bigwedge_{i=1}^h (\text{nth}(\text{vars}(c), i) = V_i \wedge U_i \leftrightarrow_{\text{var}} V_i \wedge \right. \\ \left. \wedge (n_{U_i} = n \vee n_{U_i} = m_j)) \wedge \langle n, m_1, \dots, m_k \rangle \in \text{nituples}(N, c, IT) \right\}$$

computes the tuples of node instance variables on which to instantiate a node constraint c belonging to a node N ,

$$\text{nituples}(N, c, IT) = \left\{ \langle n, m, \dots, m_k \rangle \mid n \in IT \wedge n \leftrightarrow_{\text{Node}} N \wedge \right. \\ \wedge \forall j \in \{1, \dots, k\} m_j \in IT \wedge \\ \wedge m_j \leftrightarrow_{\text{Node}} M_j \wedge \text{apath}(n, m_j, IT) = q \wedge \\ \wedge \forall \langle M_l, p \rangle \in \text{nodesPaths}(c) \text{ } q \text{ shortest} \\ \left. \text{path in } \text{mapath}(n, p, M_j, IT) \right\}$$

computes the tuples of node instances to consider for the instantiation of a node constraint c belonging to the node N (i.e., tuples of the form $\langle n, m_1, \dots, m_k \rangle$ where n is an instance

of N , and the other elements of the tuples are ancestors of n connected to it through paths matching with meta-paths appearing in c),

$$L_{\text{edge}}(c, e, N, IT) = \left\{ [U_1, \dots, U_h, IC_n^e] \mid \bigwedge_{i=1}^h (nth(\text{vars}(c), i) = V_i \wedge U_i \leftrightarrow_{\text{Var}} V_i \wedge \right. \\ \left. \wedge (n_{U_i} = n \vee n_{U_i} = m_j)) \wedge \langle n, m_1, \dots, m_k \rangle \in \text{nituples}(N, c, IT) \right\}$$

computes the tuples of node instance variables on which to instantiate a cardinality constraint c belonging to an edge e with parent node N . The function δ instantiate a node model constraint c with respect to an instance tree IT , it is defined as follows.

$$\delta(c, IT) = \begin{cases} allDifferent(U) & \text{if } c \equiv allDifferent(V) \wedge \\ & \wedge \text{instvars}(V, IT) = U \\ \\ aggConstraint(f, U, op, n) & \text{if } c \equiv aggConstraint(f, v, op, n) \wedge \\ & \wedge \text{instvars}(V, IT) = U \\ \\ \{c[V_1/U_1, \dots, V_h/U_h] \mid & \text{otherwise} \\ [V_1, \dots, V_h] = \text{vars}(c), \\ [U_1, \dots, U_h] \in L_{\text{nmc}}(c, IT)\} \end{cases}$$

where:

$$\text{instvars}([V_1, \dots, V_h], IT) = \circ_{i=1}^h \text{instvar}(V_i, IT), \\ \text{instvar}(V, IT) = \left[U \mid U \leftrightarrow_{\text{Var}} V \wedge V = \langle v, N_v, p \rangle \wedge \text{ancp}(n_U, p, IT) \right],$$

are functions for computing the list of instance variables on which to instantiate an *allDifferent* or *aggConstraint* constraint. Finally, L_{nmc} and L_{cmc} are functions computing the tuples of node instance variables and cardinality variables on which to instantiate a node model constraint or a cardinality model constraint, respectively.

$$L_{\text{nmc}}(c, IT) = \left\{ [U_1, \dots, U_h] \mid \bigwedge_{i=1}^h (nth(\text{vars}(c), i) = V_i \wedge U_i \leftrightarrow_{\text{Var}} V_i \wedge \right. \\ \left. \wedge V_i = \langle v_i, N_{v_i}, p \rangle \wedge \text{ancp}(n_{U_i}, p, IT)) \right\},$$

$$L_{\text{cmc}}(c, IT) = \left\{ [(IC_1)_n^{e_1}, \dots, (IC_h)_n^{e_h}] \mid \bigwedge_{i=1}^h (nth(\text{cards}(c), i) = C_i \wedge \right. \\ \left. \wedge e_i = \text{edgce}(C_i)) \wedge N = \text{node}(c) \wedge n \leftrightarrow_{Node} N \right\}.$$

5.4.2 MART Instances

Given the definition of MART model presented in Section 5.3.3, an executable process can be represented with a MART *instance*. To define the notion of MART instance we need to introduce the notion of MART *candidate instance*. Let M_{Proc} be a MART model, a MART candidate instance consists of the following components.

- A set \mathcal{I} of *activity instances*, i.e., couples of the form $\langle A, i \rangle$ where A is a (multiple instance, composite, multiple instance composite, atomic) activity such that $exec_A = 1$ (A belongs to M_{Proc} or to the MART model of a composite activity B such that $exec_B = 1$), $i \in \mathbb{N}$ is a unique id for instances of A . In the following, we will denote with a an instance of the activity A .
- A set \mathcal{F} of flags $exec_A$, one for each activity A (in M_{Proc} or in MART models of composite activities) such that $exec_A \neq 1$.
- A set \mathcal{A} of *assignments* for all model variables and activity parameters (i.e., time instant variables, duration variables, execution flags, quantity resource variables, instance number variables), that is, expressions of the form $P = value$ where P is a model variable or an activity parameter, and $value \in \mathbb{Z} \vee value \in D(P)$. Assignments for instance number variables and execution flags are automatically computed from the set of activity instances.

We represent a MART candidate instance with a triple $\langle \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$.

In the following, we will use the term *partial candidate MART instance* to indicate a triple $\langle \mathcal{I}, \mathcal{F}, \tilde{\mathcal{A}} \rangle$ where $\tilde{\mathcal{A}}$ is a set of assignments for a subset of model variables and activity parameters, that is, to refer to a candidate instance under definition.

A MART instance is a candidate instance such that the assignments in \mathcal{A} satisfy all the temporal, resource, and duration constraints, appropriately instantiated on activity instances.

Intuitively, the (constraint) instantiation mechanism work as follows. Let A be an activity and a_1, \dots, a_k be its instances. If r is the resource constraint $\langle A, R, q, TE \rangle$, then it is instantiated on each instance of A . Namely, for each $i = 1, \dots, k$, we obtain a constraint $\langle a_i, R, q_i, TE \rangle$, where q_i is a fresh variable having the same domain as q . Let c be an activity duration constraint for A , for each $i = 1, \dots, k$ we obtain a constraint by substituting, in c , d_A with d_{a_i} and each quantity variable q with the corresponding variable q_i . Finally, let B be an activity and let b_1, \dots, b_h be its instances. If c is a temporal constraint involving A and B , then we obtain a constraint on activity instances for each ordered tuple $\langle i, j \rangle$, with $i = 1, \dots, k$, $j = 1, \dots, h$, substituting in c each occurrence of A with a_i , and of B with b_j . This mechanism can be easily extended to temporal constraints involving more than two activities.

Formally, we define a function μ_{Proc} that computes a set of constraints instantiated on activity instances given as input the following sets: the set of activity instances \mathcal{I} ; the set $\mathcal{RDC} = \mathcal{R} \cup \mathcal{D} \cup \mathcal{C}$, where \mathcal{R} , \mathcal{D} and \mathcal{C} are the set of resource constraints, duration constraints, and temporal constraints in M_{Proc} and in its composite activities MART models, respectively; the set \mathcal{I} of implicit constraints on instance number variables. To define μ_{Proc}

we preliminary need to introduce some basic notions. $nth(L, i)$ is the function on lists defined in Section 5.4.1. If c is a temporal constraint $\mathbf{acts}(c)$ is the list of activities involved in c . Given an activity instance a and an activity A , we say that $a \leftrightarrow_{Act} A$ if and only if a is an instance of A (correspondence between activity instances and activities). We will denote with $\mathbf{pInsts}(a)$ the set of instances of activities in the process associated to the composite activity instance a . The function μ_{Proc} is defined as follows:

$$\mu_{Proc}(\mathcal{I}, \mathcal{RCP}, \mathcal{I}) = \bigcup_{a \in \mathcal{I}} \alpha(a) \cup \bigcup_{c \in \mathcal{RCP}} \gamma(c, \mathcal{I}) \cup \bigcup_{c \in \mathcal{I}} \beta(c, \mathcal{I}).$$

The function α generates the set of default constraints on duration, start time, and finishing time for an activity instance a .

$$\alpha(a) = \begin{cases} t^{\text{Comp}}(a) & \text{if } a \text{ is a composite activity instance} \\ t(a) & \text{otherwise} \end{cases}$$

where:

$$t^{\text{Comp}}(a) = \{t_a^{\text{start}} = \min_{b \in \mathbf{pInsts}(a)} t_b^{\text{start}}, t_a^{\text{end}} = \max_{b \in \mathbf{pInsts}(a)} t_b^{\text{end}}, \\ t_a^{\text{end}} \geq t_a^{\text{start}}, d_a = t_a^{\text{end}} - t_a^{\text{start}}, exec_A = 1\},$$

$$t(a) = \{t_a^{\text{start}} \geq 0, t_a^{\text{end}} \geq t_a^{\text{start}}, d_a = t_a^{\text{end}} - t_a^{\text{start}}, exec_A = 1\}.$$

The function γ instantiate a constraint c on activity instances in \mathcal{J} .

$$\gamma(c, \mathcal{J}) = \left\{ \begin{array}{ll} \{ \langle a, R, q_a^R, TE, T, C \rangle \mid a \in \mathcal{J} \wedge & \text{if } c \in \mathcal{R} \wedge \\ \quad \wedge a \leftrightarrow_{Act} A \wedge q_a^R = q_A^R \} & \wedge ((c = \langle A, R, q_A^R, TE \rangle \wedge \\ & T = noCond \wedge C = true) \vee \\ & \vee (c = (C \rightarrow \langle A, R, q_A^R, TE \rangle) \wedge \\ & \wedge T = iffCond) \vee \\ & \vee (c = (C \leftrightarrow \langle A, R, q_A^R, TE \rangle) \wedge \\ & \wedge T = iffCond)) \\ c & \text{if } c \in \mathcal{R} \wedge c = initialLevel(R, iv) \\ \\ \{ c[d_A/d_a, q_A^R/q_a^R] \mid & \text{if } c \in \mathcal{D} \\ \quad \wedge a \in \mathcal{J} \wedge a \leftrightarrow_{Act} A \} & \\ \\ \{ c[A_1/a_1, \dots, A_k/a_k] \mid & \text{if } c \in \mathcal{C} \\ \quad \wedge [A_i, \dots, A_k] = \mathbf{acts}(c) \wedge \\ \quad \wedge [a_1, \dots, a_k] \in L_{act}(c, \mathcal{J}) \} & \end{array} \right.$$

Notice that instantiated resource constraints have two additional parameters: the parameter $T \in \{iffCond, iffCond, noCond\}$ determines if the constraint if-conditional, iff-conditional, or if it does not have a condition; the parameter C is the condition for the constraint ($C \equiv true$ if the resource constraint does not have a condition). The function $L_{act}(c, \mathcal{J})$ generates all the k -tuple of activity instances that are instances of activities involved in a constraint c :

$$L_{act}(c, \mathcal{J}) = \left\{ [a_1, \dots, a_k] \mid \bigwedge_{j=1}^k (nth(\mathbf{acts}(c), j) = A_j \wedge a_j \leftrightarrow_{Act} A_j) \right\}.$$

The function β instantiate an implicit constraint on an instance number variable on activity instances in \mathcal{J} :

$$\beta(c, \mathcal{J}) = \left(inst_A \geq \frac{\sum_{\langle i_C, c \rangle \in I(Cons(R), \mathcal{J})} (i_C \cdot c) - v - \sum_{\langle i_P, p \rangle \in I(Prod(R), \mathcal{J})} (i_P \cdot p)}{k} \right),$$

where,

$$I(Set, \mathcal{J}) = \{ \langle i_A, c \rangle \mid \langle I_A, c \rangle \in Set \wedge i_A = |\{a \mid a \in \mathcal{J} \wedge a \leftrightarrow_{Act} A\}| \}.$$

5.4.3 PRODPROC Instances

Given a PRODPROC model

$$M_{\text{ProdProc}} = \langle M_{\text{Prod}}, M_{\text{Proc}}, \mathcal{C}_{\text{Couple}} \rangle,$$

a PRODPROC *candidate instance* consists of:

- a PROMO candidate instance I_{Prod} of the model M_{Prod} ;
- a MART candidate instance I_{Proc} of the model M_{Proc} .

A PROMO candidate instance is a PROMO *instance* if the assignments for variables in the PROMO candidate instance and in the MART candidate instance satisfy all the appropriately instantiated constraints in M_{Prod} and in M_{Proc} (MART product related constraints are translated into the corresponding resource constraints and then instantiated), respectively, and the set of instantiated coupling constraints obtained from $\mathcal{C}_{\text{Couple}}$ (for the sake of simplicity we assume that this set contains also the constraints different from resource constraints obtained from MART product related constraints).

To define the instantiation mechanism for coupling constraints, we preliminary need to introduce some basic notions. Given a coupling constraint c , $\text{prodvars}(c)$ is the list of node variables and cardinality variables occurring in c . The function $\text{inst}_{\text{Couple}}(c, I_{M_{\text{Prod}}}, IT)$ determines if a coupling constraint can be instantiated and is defined as follows:

$$\text{inst}_{\text{Couple}}(c, I_{M_{\text{Prod}}}, IT) = \begin{cases} \text{true} & \text{if } \forall \langle N, p \rangle \in \text{nodesPaths}(c) \\ & \exists n \in IT. (n \leftrightarrow_{\text{Node}} N \wedge \text{ancp}(n, p, IT) = \text{true}) \wedge \\ & \wedge \forall \langle \text{label}, N, M, \text{Card} \rangle \in \text{prodvars}(c) \\ & \exists n \in IT. n \leftrightarrow_{\text{Node}} N \\ \text{false} & \text{otherwise} \end{cases}$$

The set of instantiated coupling constraints is computed using a function μ_{Couple} in the following way.

$$\bigcup_{c \in \mathcal{C}_{\text{Couple}} \wedge \text{inst}_{\text{Couple}}(c, I_{M_{\text{Prod}}}, IT)} \mu_{\text{Couple}}(c, I_{M_{\text{Prod}}}, M_{\text{Proc}}),$$

where:

$$\begin{aligned} \mu_{\text{Couple}}(c, I_{M_{\text{Prod}}}) = \{ & c[V_1/U_1, \dots, V_h/U_h] \mid [V_1, \dots, V_n] = \text{prodvars}(c) \wedge \\ & \wedge [U_1, \dots, U_h] \in L_{\text{Couple}}(c, I_{M_{\text{Prod}}}) \} \end{aligned}$$

instantiates a coupling constraints on each tuple of node instance variables computed by the function L_{Coupl} ,

$$\begin{aligned}
 L_{\text{Coupl}}(c, I_{M_{\text{Prod}}}) = & \left\{ [U_1, \dots, U_h] \wedge_{i=1}^h (\text{nth}(\text{prodvars}(c), i) = V_i \wedge \right. \\
 & \wedge ((V_i = \langle v_i, N_{V_i}, p \rangle \wedge U_i \leftrightarrow_{Var} V_i \wedge \text{ancp}(n_{U_i}, p, IT)) \vee \\
 & \vee (V_i = \langle label, N, M, Card \rangle \wedge e_i = \text{edge}(V_i) \wedge \\
 & \left. \wedge n \leftrightarrow_{Node} N \wedge U_i = IC_n^{e_i})) \right\}.
 \end{aligned}$$

Notice that the instantiations of a coupling constraint c differ only by node instance variables and cardinality instance variables. They all share the same process and instance number variables. Since all the instantiation of a coupling constraint have to hold, care must be taken when defining a coupling constraint. This in order to avoid possible inconsistencies during the configuration of a model.

Chapter 6

Case Study: Building Construction

In this chapter we show how the PRODPROC framework presented in Chapter 5 can be used to model a product and its production process. Section 6.1 presents a description of the product we consider, i.e., a prefabricated component building, while its production process is described in Section 6.2. Section 6.3 and Section 6.4 present the PROMo model of the building and the MART model of its production process, respectively. In Section 6.5 the coupling of the product model with the process model is defined.

6.1 Prefabricated Component Building

In order to show the capabilities of the PRODPROC framework, we consider as configurable product a rectangular base prefabricated component multi-story building. The building components are the following: story, roof, heating service, ventilation service, sanitary service, electrical/lighting service, suspended ceiling, floor, partition wall system. We consider two building types:

Warehouse: it is a single story building, it has no mandatory service except for the electrical/lighting service, it has no partition wall system and no suspended ceiling, it may have a basement.

Office building: it may have a basement and up to three story, all services are mandatory except for ventilation, suspended ceiling, floor, and partition wall system are mandatory for each story.

6.2 Building Construction

In this section we give a general description of the building construction process [Som10]. This process can be split in four main phases: preparation and development of the building site (Section 6.2.1); building shell and building envelope works (Section 6.2.2); building services equipment (Section 6.2.3); finishing works (Section 6.2.3).

6.2.1 Preparation and Development of the Building Site

Prior to commencing actual works, the construction project initially requires the creation of an infrastructure. In the following, we give a brief description of the tasks involved in this phase of the building construction process. The precedence relations between these tasks are depicted in Figure 6.1. The black box in the figure represents the second main phase of the construction process.

Securing of terrain The first task must always be securing the site with a suitable construction site fence (mesh wire, planks etc.) with stationary entry and exit.

Preparation of terrain Once the terrain has been secured, demolition and removal of existing buildings, trees, refuse etc. commences.

Line relaying, rerouting of main lines If there are public or private supply or disposal lines on the site, required rerouting - or often also shutdown - of these lines is to be coordinated with the parties involved and rerouting to be undertaken. It is recommendable to immediately also lay future main supply and disposal lines, all the way to the ducts close to the building, in order to prevent having to dig up the earth outside the building pit later on.

Supply and disposal lines It is recommended for large construction sites to create main connections straight away for construction site water and electricity supply and to make available sufficient space for storage areas and living options.

Creation of construction site The creation of the construction site consist of two activities.

- *Construction roads*: the building of construction roads immediately follows cable laying.
- *Construction site facilities*: this includes all facilities that the contractor requires for the provision of services, under adherence to the relevant regulations, for instance: transport equipment (cranes, concrete pumps, construction elevators); mixing equipment with access routes or transfer containers for ready-mixed concrete; storage areas for casing, reinforcements, prefabricated components, bricks and other materials; barracks for construction management, accommodation, canteen and sanitary facilities; construction site electricity transformers, water hydrants.

Subsoil improvement If the soil report points to measures for being required improving the subsoil, this must be undertaken either at the same time or immediately after the construction roads.

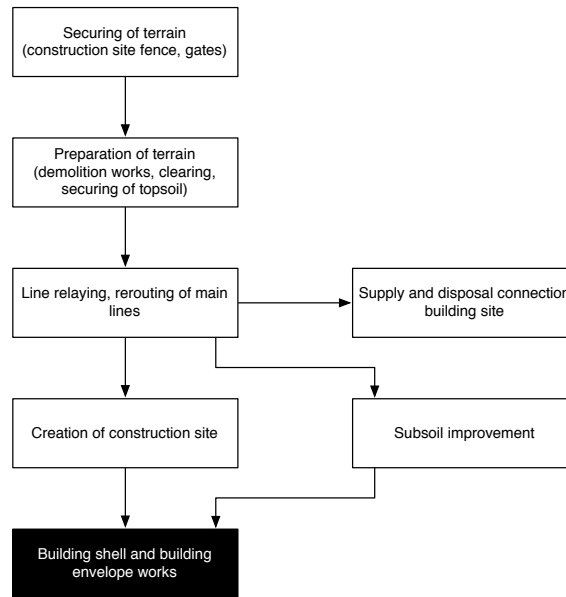


Figure 6.1: Preparation and development of the building site.

6.2.2 Building Shell and Building Envelope Works

The first parcel in construction is shell construction with space enclosure, meaning creation of the construction structure with a weatherproof envelope. This phase of the building construction process consists of seven tasks. The precedence relations between these tasks are depicted in Figure 6.2. The two black boxes in the figure represent the first and the third main phases of the construction process.

Excavation works and pit lining First, the topsoil needs to be removed and deposited on the areas provisioned for this, where it is stored. Then, actual excavation works commence according to the excavation plan, all the way to rough grading. Afterward, smaller machines are used for fine grading and for excavation works for foundations and supply lines. If the pit cannot be scarped because it is either too deep or there is not enough space (downtown), the pit sides need to be secured through cladding. There are various options for this: Berlin type pit lining with back anchoring (for regular loads and buildings with working area); Bored pile wall with back anchoring (for extreme loads, securing of neighboring buildings); Diaphragm wall (for buildings where the lining is to serve at the same time as supporting wall or at least as protective wall for the insulation); Sheet piling with the same effect as bored pile wall but rammed (very expensive). Sheet piling is rarely used for construction pits in structural engineering.

Foundation works Once the building pit base has been completed on a sufficiently large area (ca. 400 m²) foundation work and drainage pipe work can commence. Foundations

and drainage pipes are almost always mutually dependent. Often, parallel to the foundations, concreted floor ducts, elevator underpasses (hydraulic pillars), wells etc. need to be installed. After completion of these components, the resulting hollow areas are filled (for instance with sieve refuse). Afterward, a granular sub-base is applied and, on this, a reinforced base plate installed.

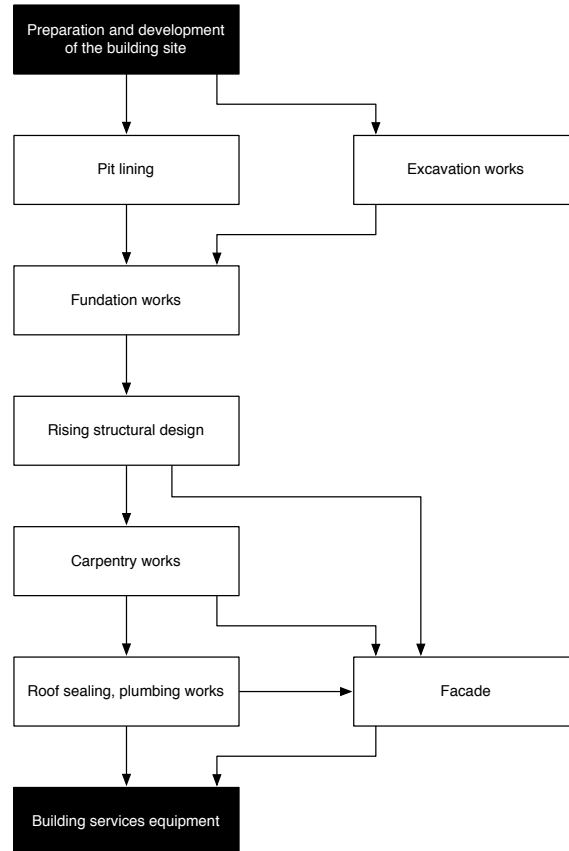


Figure 6.2: Building shell and building envelope works.

Rising structural design After the completion of foundation works the prefabricated component construction can start.

Carpentry works Carpentry works extend to: creation of substructure, creation of roof boarding and articles, wall constructions, ceiling and roof supports and trusses.

Roof sealing, plumbing works Due to their close connection, these two trades need to be executed jointly. Roof sealing should come immediately after the carpentry works in order to avoid excess soaking of the wood structure.

Facade The facade seals off the vertical parts of the building envelope with enclosed (opaque) and lit (transparent) surfaces. Non-supportive facades positioned in front of struts and ceilings (also known as a curtain wall), enclose the entire building in its vertical spatial enclosures and handles all building physics functions. As a rule, they are designed as a metal facade with integrated windows and solar protection devices. Assembly is undertaken following completion of shell construction, since there is great danger of damage. Non-supportive facades positioned between struts and ceilings are most often used when a reinforced concrete skeleton structure is “filled in” with brickwork. However, simple design adjusted facade elements of wood or metal are also installed in this position. In these cases, the frontal sides of the ceilings and struts need to be separately insulated. Facade installation can be undertaken shortly after shell construction. There is reduced danger of damage.

6.2.3 Building Services Equipment and Finishing Works

The third phase of the building construction process consists in the installation of services. We consider the following services: ventilation service, heating service, sanitary service, and electric service. Figure 6.3 presents the activities that are involved in this phase of the process, and their precedence relations. The two black boxes in the figure represent the second and the fourth main phases of the construction process.

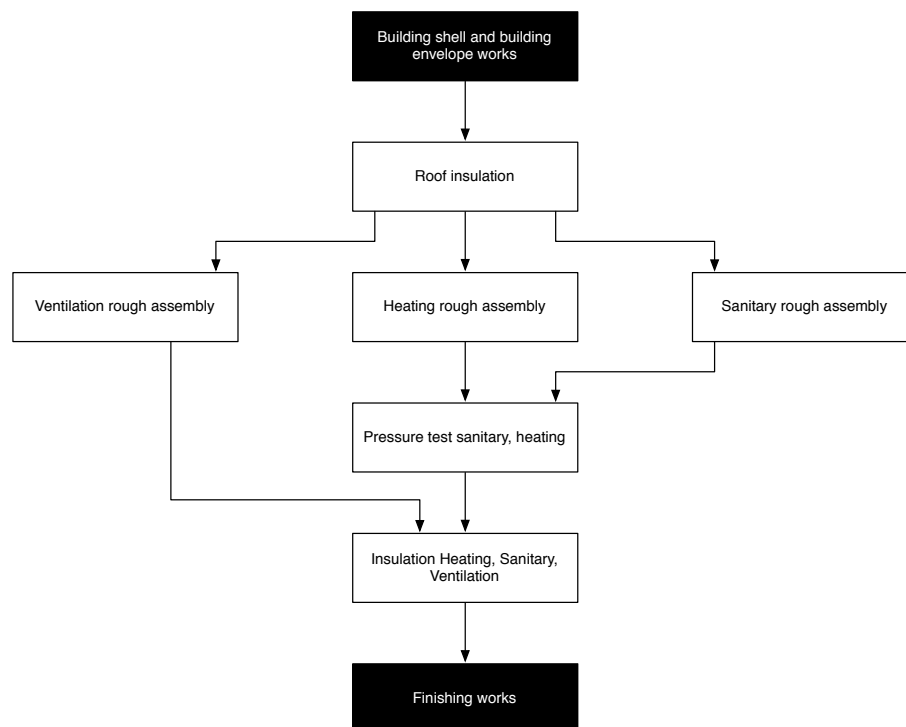


Figure 6.3: Building services equipment.

The finishing works represent the last phase of the building construction process. We

consider the following finishing works: floor covering, partition wall system construction and suspended ceiling construction. This phase includes also the assembly of the electric service. Figure 6.4 shows the process for accomplishing these three works. The black box in the figure represents the third main phase of the construction process.

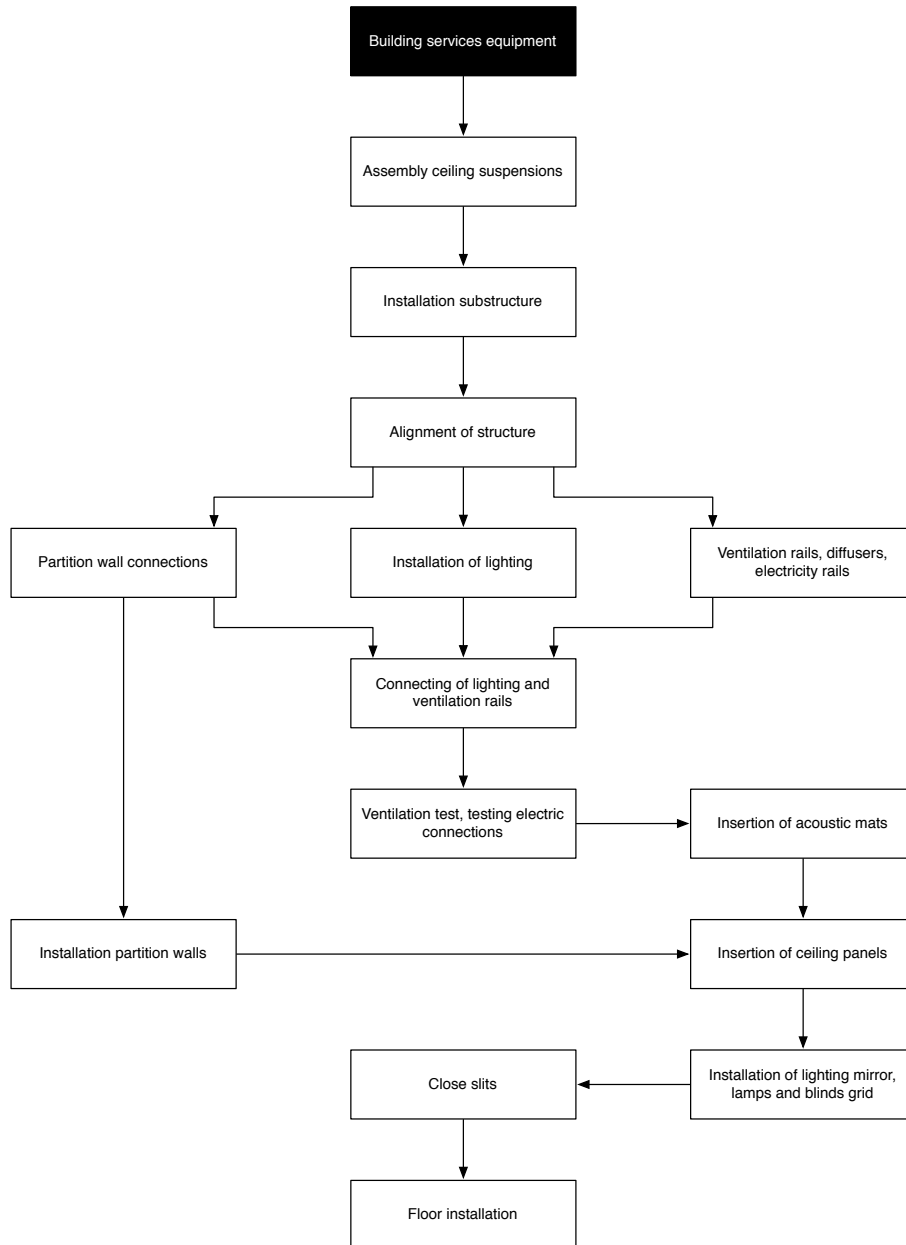


Figure 6.4: Finishing works.

6.3 Building PROMo Model

In this section we present a PROMo model for the product introduced in Section 6.1. We will use the following sets for defining node and cardinality variables.

<i>buildings</i>	=	{Warehouse, Office building},
<i>storyNumber</i>	=	[1,3],
<i>optional</i>	=	{0,1},
<i>storyHeight</i>	=	[3,15],
<i>widthLength</i>	=	[7,90],
<i>ventilationType</i>	=	{VAV, VAV induction, CAV, Multizone},
<i>heatingType</i>	=	{Hydronic convectors, Floor heating},
<i>voltage</i>	=	{Medium, Normal},
<i>floorCeiling</i>	=	{Floor, Ceiling},
<i>flatSloping</i>	=	{Flat, Sloping},
<i>roofAngle</i>	=	[0,45],
<i>floors</i>	=	{Screed covered, Hollow floor},
<i>partitionType</i>	=	{Heavy, Light_1, Light_2},
<i>partitionConf</i>	=	{PC1, PC2, PC3},
<i>ceilings</i>	=	{CT1, CT2}.

6.3.1 Building Product Model Graph

Figure 6.5 shows a graphical representation of the product model graph of the model. We represent nodes as boxes containing node names only, variables and node constraints sets are reported in the following sections. We do not show ellipses for cardinality constraints to keep the figure as tidy as possible, cardinality constraint sets are reported in the following sections too. Regarding edges, the ones having constant cardinality are labeled with an integer, while the ones having a variable representing the cardinality are labeled with the variable domain.

Building PROMo Nodes

Building: it is the root node of the product model graph, it is defined as follows.

$$\langle Building, \mathcal{V}_{Building}, \mathcal{C}_{Building} \rangle,$$

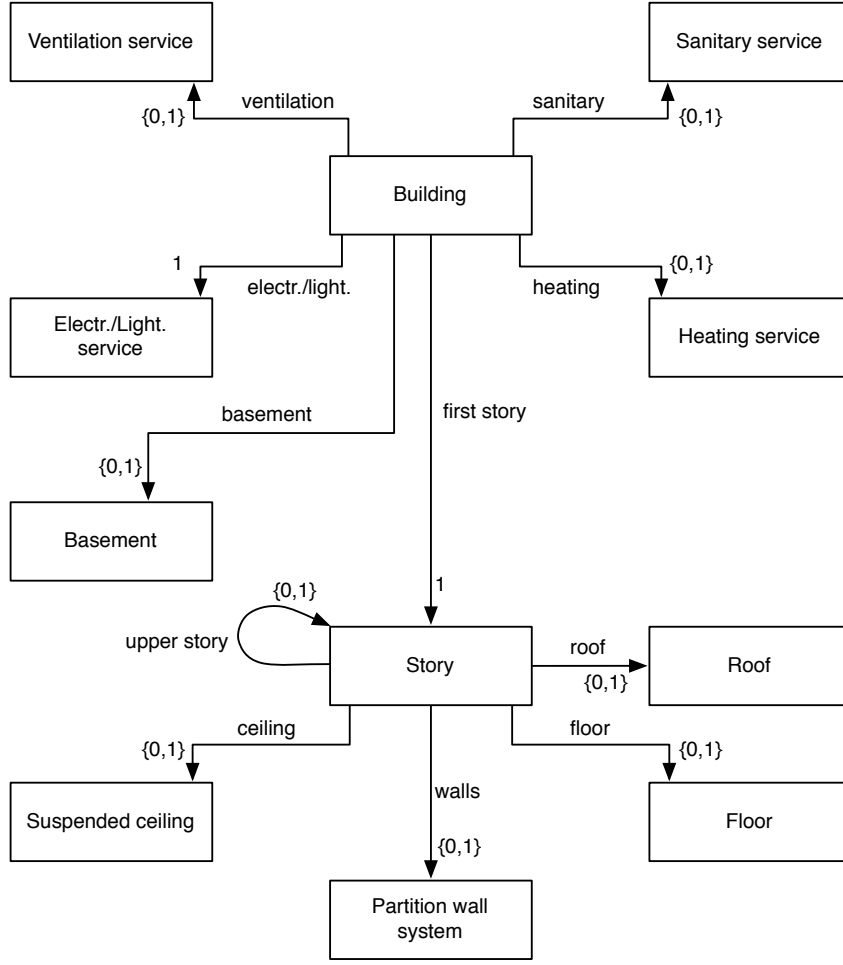


Figure 6.5: Building product model graph.

where:

$$\mathcal{V}_{Building} = \{ \langle BuildingType, buildings \rangle, \\ \langle StoryNum, storyNumber \rangle, \\ \langle Width, widthLength \rangle, \\ \langle Length, widthLength \rangle \},$$

$$\mathcal{C}_{Building} = \{ BuildingType = Warehouse \Rightarrow StoryNum = 1 \}.$$

Ventilation service: it is the node representing the ventilation service of the building, it is defined as follows.

$$\langle Ventilation\ service, \mathcal{V}_{Ventilation\ service}, \mathcal{C}_{Ventilation\ service} \rangle,$$

where:

$$\begin{aligned}
 \mathcal{V}_{Ventilation\ service} &= \{\langle VentType, ventilationType \rangle\}, \\
 \mathcal{C}_{Ventilation\ service} &= \{\langle BuildingType, Building, [ventilation] \rangle = \\
 &= Warehouse \Rightarrow VentType \neq VAV\ induction, \\
 &\langle BuildingType, Building, [ventilation] \rangle = \\
 &= Office\ building \Rightarrow VentType \neq CAV\}.
 \end{aligned}$$

Sanitary service: it is the node representing the sanitary service of the building, it is defined as follows.

$$\langle Sanitary\ service, \mathcal{V}_{Sanitary\ service}, \mathcal{C}_{Sanitary\ service} \rangle,$$

where:

$$\begin{aligned}
 \mathcal{V}_{Sanitary\ service} &= \{\langle Gas, optional \rangle, \\
 &\langle Water, optional \rangle, \\
 &\langle SanInstall, optional \rangle\}, \\
 \mathcal{C}_{Sanitary\ service} &= \{\langle BuildingType, Building, [sanitary] \rangle = \\
 &= Office\ building \Rightarrow Water = 1 \wedge SanInstall = 1, \\
 &SanitaryInstall = 1 \Rightarrow Water = 1\}.
 \end{aligned}$$

Heating service: it is the node representing the heating service of the building, it is defined as follows.

$$\langle Heating\ service, \mathcal{V}_{Heating\ service}, \mathcal{C}_{Heating\ service} \rangle,$$

where:

$$\begin{aligned}
 \mathcal{V}_{Heating\ service} &= \{\langle HeatType, heatingType \rangle\}, \\
 \mathcal{C}_{Heating\ service} &= \{\langle BuildingType, Building, [heating] \rangle = \\
 &= Warehouse \Rightarrow HeatType \neq Floor\ heating\}.
 \end{aligned}$$

Electr./Light. service: it is the node representing the electrical/lighting service of the building, it is defined as follows.

$$\langle Electr./Light.\ service, \mathcal{V}_{Electr./Light.\ service}, \mathcal{C}_{Electr./Light.\ service} \rangle,$$

where:

$$\begin{aligned}
 \mathcal{V}_{Electr./Light.\ service} &= \{\langle Voltage, voltage \rangle, \\
 &\langle ElectrInstall, floorCeiling \rangle\},
 \end{aligned}$$

$$\begin{aligned} \mathcal{C}_{Electr./Light. service} &= \{\langle BuildingType, Building, [electr./light.] \rangle = \\ &= \text{Office building} \Rightarrow Voltage \neq \text{Medium}\}. \end{aligned}$$

Basement: it is the node representing the basement of the building, it is defined as follows.

$$\langle Basement, \mathcal{V}_{Basement}, \mathcal{C}_{Basement} \rangle$$

where

$$\begin{aligned} \mathcal{V}_{Basement} &= \{Height = \langle Height, storyHeight \rangle\}, \\ \mathcal{C}_{Basement} &= \{Height = 3 \vee Height = 6\}. \end{aligned}$$

Story: it is the node representing a story of the building, it is defined as follows.

$$\langle Story, \mathcal{V}_{Story}, \mathcal{C}_{Story} \rangle,$$

where:

$$\begin{aligned} \mathcal{V}_{Story} &= \{\langle FloorNum, storyNumber \rangle, \langle Height, storyHeight \rangle\}, \\ \mathcal{C}_{Story} &= \{FloorNum = \langle FloorNum, Story, [upper story] \rangle + 1, \\ &FloorNum \leq \langle StoryNum, Building, [first story, \star] \rangle, \\ &\langle Type, Building, [first story, \star] \rangle = \text{Office building} \Rightarrow \\ &\Rightarrow Height \geq 4 \wedge Height \leq 5\}. \end{aligned}$$

Roof: it is the node representing the roof of the building, it is defined as follows.

$$\langle Roof, \mathcal{V}_{Roof}, \mathcal{C}_{Roof} \rangle,$$

where:

$$\begin{aligned} \mathcal{V}_{Roof} &= \{\langle RoofType, flatSloping \rangle, \langle Angle, roofAngle \rangle\}, \\ \mathcal{C}_{Roof} &= \{RoofType = \text{Flat} \Rightarrow Angle = 0, \\ &RoofType = \text{Sloping} \Rightarrow Angle > 0\}. \end{aligned}$$

Suspended ceiling: it is the node representing the suspended ceiling of a story of the building, it is defined as follows.

$$\langle Suspended ceiling, \mathcal{V}_{Suspended ceiling}, \emptyset \rangle,$$

where

$$\mathcal{V}_{Suspended ceiling} = \{\langle CeilingType, ceilings \rangle\}.$$

Partition wall system: it is the node representing the partition wall system of a story of the building, it is defined as follows.

$$\langle \textit{Partition wall system}, \mathcal{V}_{\textit{Partition wall system}}, \emptyset \rangle,$$

where

$$\begin{aligned} \mathcal{V}_{\textit{Partition wall system}} = \{ & \langle \textit{PartType}, \textit{partitionType} \rangle, \\ & \langle \textit{Conf}, \textit{partitionConf} \rangle \}. \end{aligned}$$

Floor: it is the node representing the floor of a story of the building, it is defined as follows.

$$\langle \textit{Floor}, \mathcal{V}_{\textit{Floor}}, \mathcal{C}_{\textit{Floor}} \rangle,$$

where:

$$\begin{aligned} \mathcal{V}_{\textit{Floor}} = \{ & \langle \textit{FloorType}, \textit{floors} \rangle, \\ & \langle \textit{SoundAbsorption}, \textit{optional} \rangle \}, \\ \mathcal{C}_{\textit{Floor}} = \{ & \textit{FloorType} = \textit{Hollow floor} \Rightarrow \textit{SoundAbsorption} = 0 \}. \end{aligned}$$

Building PROMo Edges

ventilation: it is the edge representing the *has-part* relation over the building and the ventilation service, it is defined as follows.

$$\langle \textit{ventilation}, \textit{Building}, \textit{Ventilation service}, \langle \textit{Card}, \textit{optional} \rangle, \emptyset \rangle.$$

sanitary: it is the edge representing the *has-part* relation over the building and the sanitary service, it is defined as follows.

$$\langle \textit{sanitary}, \textit{Building}, \textit{Sanitary service}, \langle \textit{Card}, \textit{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\mathcal{CC} = \{ \textit{BuildingType} = \textit{Office building} \Rightarrow \textit{Card} = 1 \}.$$

heating: it is the edge representing the *has-part* relation over the building and the heating service, it is defined as follows.

$$\langle \textit{heating}, \textit{Building}, \textit{Heating service}, \langle \textit{Card}, \textit{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\mathcal{CC} = \{ \textit{BuildingType} = \textit{Office building} \Rightarrow \textit{Card} = 1 \}.$$

electr./light.: it is the edge representing the *has-part* relation over the building and the electrical/lighting service, it is defined as

$$\langle \textit{electr./light.}, \textit{Building}, \textit{Electr./Light. service}, 1, \emptyset \rangle.$$

basement: it is the edge representing the *has-part* relation over the building and its basement, it is defined as follows.

$$\langle \text{basement}, \text{Building}, \text{Basement}, \langle \text{Card}, \text{optional} \rangle, \emptyset \rangle.$$

first story: it is the edge representing the *has-part* relation over the building and its first story, it is defined as

$$\langle \text{first story}, \text{Building}, \text{Story}, 1, \emptyset \rangle.$$

upper story: it is the edge representing the *has-part* relation over two story of the building, it is defined as follows.

$$\langle \text{upper story}, \text{Story}, \text{Story}, \langle \text{Card}, \text{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\begin{aligned} \mathcal{CC} = \{ & \text{FloorNum} = \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 0, \\ & \text{FloorNum} < \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 1 \}. \end{aligned}$$

roof: it is the edge representing the *has-part* relation over a story and the building roof, it is defined as follows.

$$\langle \text{roof}, \text{Story}, \text{Roof}, \langle \text{Card}, \text{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\begin{aligned} \mathcal{CC} = \{ & \text{FloorNum} = \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 1, \\ & \text{FloorNum} < \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 0 \}. \end{aligned}$$

ceiling: it is the edge representing the *has-part* relation over a story and its ceiling, it is defined as follows.

$$\langle \text{ceiling}, \text{Story}, \text{Suspended ceiling}, \langle \text{Card}, \text{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\begin{aligned} \mathcal{CC} = \{ & \langle \text{BuildingType}, \text{Building}, [\text{first story}] \rangle = \text{Warehouse} \Rightarrow \text{Card} = 0, \\ & \langle \text{BuildingType}, \text{Building}, [\text{first story}, \star] \rangle \neq \text{Warehouse} \Rightarrow \text{Card} = 1 \}. \end{aligned}$$

walls: it is the edge representing the *has-part* relation over a story and its partition wall system, it is defined as follows.

$$\langle \text{walls}, \text{Story}, \text{Partition wall system}, \langle \text{Card}, \text{optional} \rangle, \mathcal{CC} \rangle,$$

where:

$$\begin{aligned} \mathcal{CC} = \{ & \langle \text{BuildingType}, \text{Building}, [\text{first story}] \rangle = \text{Warehouse} \Rightarrow \text{Card} = 0, \\ & \langle \text{BuildingType}, \text{Building}, [\text{first story}, \star] \rangle \neq \text{Warehouse} \Rightarrow \text{Card} = 1 \}. \end{aligned}$$

floor: it is the edge representing the *has-part* relation over a story and its floor, it is defined as follows.

$$\langle floor, Story, Floor\ ceiling, \langle Card, optional \rangle, CC \rangle,$$

where:

$$CC = \{ \langle BuildingType, Building, [first\ story, \star] \rangle = Office\ building \Rightarrow Card = 1 \}.$$

6.3.2 Model Constraints

The model constraint of the model are the following.

$$\langle FloorNum, Story, [first\ story] \rangle = 1,$$

$$\begin{aligned} \langle ElectrInstall, Electr./Light., [electr./light.] \rangle &= Floor \Rightarrow \\ \Rightarrow \langle FloorType, Floor, [floor] \rangle &= Hollow\ floor, \end{aligned}$$

$$\langle upper\ story, Story, Story, Card \rangle \neq \langle roof, Story, , Roof, Card \rangle.$$

6.4 Building Construction MART Model

In this section we present a MART model for the process described in Section 6.2. It consists of four composite activities (one of them is a multiple instance composite activity), one for each of the main phases of the building construction process. Figure 6.6 shows a graphical representation of the temporal constraint network of the model.

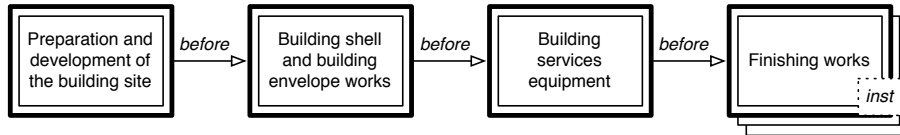


Figure 6.6: Temporal constraint network for the building construction process.

The set of model variables for the (main) process is defined as $\mathcal{V}_{Main} = \mathcal{V}_{Resources} \cup \mathcal{V}_{Proc}$

where:

$$\begin{aligned}
 \mathcal{V}_{\text{Resources}} &= \{ \langle \text{GeneralWorkers}, [0,10] \rangle, \\
 &\quad \langle \text{Excavators}, [0,3] \rangle, \\
 &\quad \langle \text{Trucks}, [0,3] \rangle, \\
 &\quad \langle \text{Cranes}, [0,2] \rangle, \\
 &\quad \langle \text{VentilationInstallers}, [0,2] \rangle \}, \\
 \mathcal{V}_{\text{Proc}} &= \{ \langle \text{BuildingArea}, [49,8100] \rangle, \\
 &\quad \langle \text{Basement}, [0,1] \rangle, \\
 &\quad \langle \text{StoryNum}, [1,3] \rangle \}.
 \end{aligned}$$

We consider the following initial level constraints for resource variables in $\mathcal{V}_{\text{Resources}}$.

$$\begin{aligned}
 &\text{initialValue}(\text{GeneralWorkers}, 10), \\
 &\text{initialValue}(\text{Excavators}, 3), \\
 &\text{initialValue}(\text{Trucks}, 3), \\
 &\text{initialValue}(\text{Cranes}, 2), \\
 &\text{initialValue}(\text{VentilationInstallers}, 2).
 \end{aligned}$$

For the multiple instance activity “Finishing works” we have:

$$\langle \text{inst}_{\text{Finishing works}}, [1,3] \rangle.$$

In the following sections we define the models of the four composite activities shown in Figure 6.6.

6.4.1 Preparation and development of the building site MART Model

Figure 6.7 shows a graphical representation of the temporal constraint network of the model of the composite activity “Preparation and development of the building site”.

Model variables The set of model variables is defined as

$$\mathcal{V}_{\text{Main}} \cup \mathcal{V}_{\text{R}} \cup \mathcal{V}_{\text{P}},$$

where:

$$\begin{aligned}
 \mathcal{V}_{\text{R}} &= \{ \langle \text{Steamrollers}, [0,2] \rangle \}, \\
 \mathcal{V}_{\text{P}} &= \{ \langle \text{SubImp}, [0,1] \rangle \}.
 \end{aligned}$$

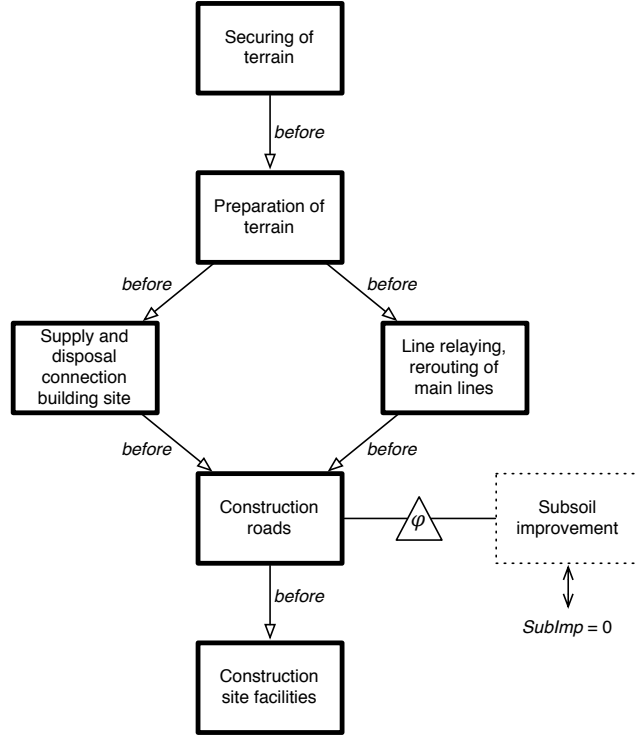


Figure 6.7: Temporal constraint network for the composite activity “Preparation and development of the building site”. φ is the temporal constraint “Construction roads” *equals* “Subsoil improvement” *or* “Construction roads” *before* “Subsoil improvement”

Resource constraints We consider the following resource constraints for activities in Figure 6.7.

$\langle \text{Securing of terrain}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Preparation of terrain}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Preparation of terrain}, \text{Excavators}, \langle q_E, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Preparation of terrain}, \text{Trucks}, \langle q_T, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Supply and disposal connection building site}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Supply and disposal connection building site}, \text{Excavators}, \langle q_E, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Line relaying, rerouting of main lines}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Line relaying, rerouting of main lines}, Excavators, \langle q_E, [-3, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Line relaying, rerouting of main lines}, Trucks, \langle q_T, [-3, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction roads}, GeneralWorkers, \langle q_{GW}, [-10, -4] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction roads}, Excavators, \langle q_E, [-2, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction roads}, Trucks, \langle q_T, [-2, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction roads}, Steamrollers, \langle q_S, [-2, -1] \rangle, FromStartToEnd \rangle,$

$initialLevel(Steamrollers, 2),$

$\langle \text{Subsoil improvement}, GeneralWorkers, \langle q_{GW}, [-10, -4] \rangle, FromStartToEnd \rangle,$

$\langle \text{Subsoil improvement}, Excavators, \langle q_E, [-2, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Subsoil improvement}, Trucks, \langle q_T, [-2, -1] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction site facilities}, GeneralWorkers, \langle q_{GW}, [-10, -4] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction site facilities}, Excavators, \langle q_E, [-1, 0] \rangle, FromStartToEnd \rangle,$

$\langle \text{Construction site facilities}, Trucks, \langle q_T, [-1, 0] \rangle, FromStartToEnd \rangle.$

Duration constraints We consider the following duration constraints for activities in Figure 6.7.

$$d_{\text{Securing of terrain}} = \frac{BuildingArea}{2 \cdot |q_{GW}|},$$

$$d_{\text{Preparation of terrain}} = \frac{BuildingArea}{|q_{GW}| + 2 \cdot |q_E| + 2 \cdot |q_T|},$$

$$d_{\text{Supply and disposal connection building site}} = \frac{BuildingArea}{2 \cdot |q_{GW}| + 4 \cdot |q_E|},$$

$$d_{\text{Line relaying, rerouting of main lines}} = \frac{BuildingArea}{2 \cdot |q_{GW}| + 2 \cdot |q_E| + 2 \cdot |q_T|},$$

$$d_{\text{Construction roads}} = \frac{BuildingArea}{|q_{GW}| + 2 \cdot |q_E| + 2 \cdot |q_T| + 3 \cdot |q_S|},$$

$$d_{\text{Subsoil improvement}} = \frac{\text{BuildingArea}}{|q_{GW}| + 3 \cdot |q_E| + 3 \cdot |q_T|},$$

$$d_{\text{Construction site facilities}} = \frac{\text{BuildingArea}}{2 \cdot |q_{GW}| + 3 \cdot |q_E| + 3 \cdot |q_T|}.$$

6.4.2 Building shell and building envelope works MART Model

Figure 6.8 shows a graphical representation of the temporal constraint network of the model of the composite activity “Building shell and building envelope works”.

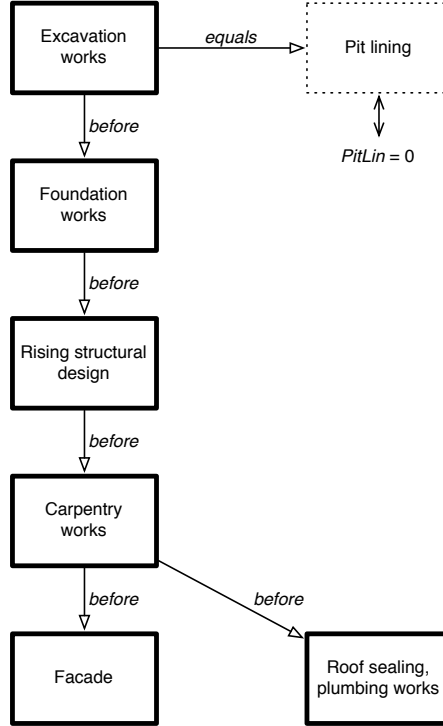


Figure 6.8: Temporal constraint network for the composite activity “Building shell and building envelope works”.

Model variables The set of model variables is defined as

$$\mathcal{V}_{\text{Main}} \cup \mathcal{V}_{\text{R}} \cup \mathcal{V}_{\text{P}},$$

where:

$$\mathcal{V}_{\text{R}} = \{\langle \text{Mixers}, [0, 2] \rangle\},$$

$$\mathcal{V}_{\text{P}} = \{\langle \text{PitLin}, [0, 1] \rangle\}.$$

Resource constraints We consider the following resource constraints for activities in Figure 6.8.

$\langle \text{Excavation works}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Excavation works}, \text{Excavators}, \langle q_E, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Excavation works}, \text{Trucks}, \langle q_T, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Pit lining}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Pit lining}, \text{Excavators}, \langle q_E, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Pit lining}, \text{Trucks}, \langle q_T, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Foundation works}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Foundation works}, \text{Mixers}, \langle q_M, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Rising structural design}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -6] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Rising structural design}, \text{Mixers}, \langle q_M, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\text{initialValue}(\text{Mixers}, 2),$

$\langle \text{Rising structural design}, \text{Trucks}, \langle q_T, [-3, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Rising structural design}, \text{Cranes}, \langle q_C, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Carpentry works}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Facade}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -5] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Facade}, \text{Mixers}, \langle q_M, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Facade}, \text{Trucks}, \langle q_T, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Facade}, \text{Cranes}, \langle q_C, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Roof sealing plumbing works}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -5] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Roof sealing plumbing works}, \text{Trucks}, \langle q_T, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Roof sealing plumbing works}, \text{Cranes}, \langle q_C, [-2, -1] \rangle, \text{FromStartToEnd} \rangle.$

Duration constraints We consider the following duration constraints for activities in Figure 6.8.

$$d_{\text{Excavation works}} = \text{Basement} \cdot \frac{\text{BuildingArea}}{|q_{GW}| + 2 \cdot |q_E| + 2 \cdot |q_T|} + (1 - \text{Basement}) \cdot \frac{\text{BuildingArea}}{2 \cdot |q_{GW}| + 3 \cdot |q_E| + 2 \cdot |q_T|},$$

$$d_{\text{Foundation works}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}| + 2 \cdot |q_M|},$$

$$d_{\text{Rising structural design}} = \frac{\text{BuildingArea} \cdot \text{StoryNum}}{|q_{GW}| + 3 \cdot |q_M| + 2 \cdot |q_T| + 3 \cdot |q_C|},$$

$$d_{\text{Carpentry works}} = \frac{\text{BuildingArea} \cdot \text{StoryNum}}{|q_{GW}|},$$

$$d_{\text{Facade}} = \frac{\text{BuildingArea} \cdot \text{StoryNum}}{|q_{GW}| + 2 \cdot |q_T| + |q_C|},$$

$$d_{\text{Roof sealing plumbing works}} = \frac{\text{BuildingArea} \cdot \text{StoryNum}}{|q_{GW}| + |q_T| + |q_C|}.$$

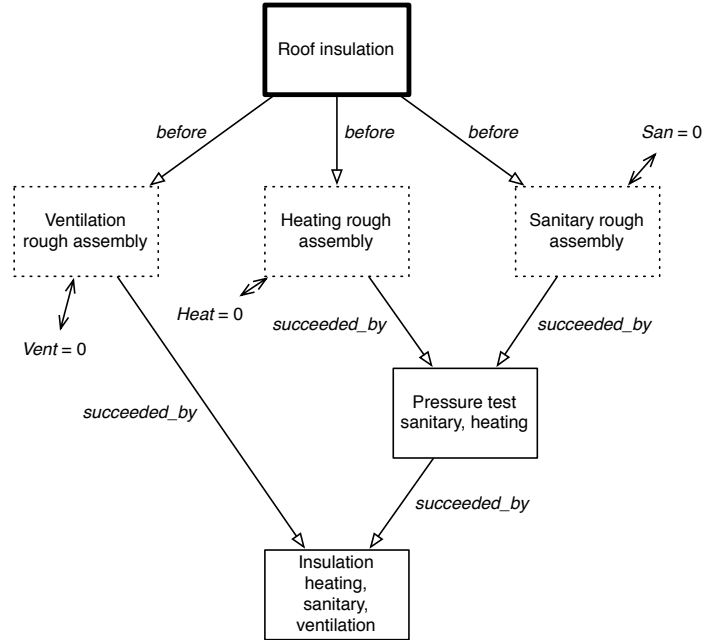


Figure 6.9: Temporal constraint network for the composite activity "Building services equipment".

6.4.3 Building services equipment MART Model

Figure 6.9 shows a graphical representation of the temporal constraint network of the model of the composite activity “Building services equipment”.

Model variables The set of model variables is defined as

$$\mathcal{V}_{\text{Main}} \cup \mathcal{V}_{\text{R}} \cup \mathcal{V}_{\text{P}},$$

where:

$$\begin{aligned}\mathcal{V}_{\text{R}} &= \{\langle \text{Plumbers}, [0, 4] \rangle\}, \\ \mathcal{V}_{\text{P}} &= \{\langle \text{San}, [0, 1] \rangle, \\ &\quad \langle \text{Heat}, [0, 1] \rangle, \\ &\quad \langle \text{Vent}, [0, 1] \rangle\}.\end{aligned}$$

Resource constraints We consider the following resource constraints for activities in Figure 6.9.

$$\langle \text{Roof insulation}, \text{GeneralWorkers}, \langle q_{\text{GW}}, [-10, -4] \rangle, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Roof insulation}, \text{Cranes}, \langle q_{\text{C}}, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Roof insulation}, \text{Trucks}, \langle q_{\text{T}}, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Ventilation rough assembly}, \text{VentilationInstallers}, -2, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Heating rough assembly}, \text{Plumbers}, \langle q_{\text{P}}, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$$

$$\text{initialValue}(\text{Plumbers}, 4),$$

$$\langle \text{Sanitary rough assembly}, \text{Plumbers}, \langle q_{\text{P}}, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Pressure test sanitary heating}, \text{Plumbers}, \langle q_{\text{P}}, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Insulation heating, sanitary, ventilation}, \text{VentilationInstallers}, -2, \text{FromStartToEnd} \rangle,$$

$$\langle \text{Insulation heating, sanitary, ventilation}, \text{Plumbers}, \langle q_{\text{P}}, [-4, -2] \rangle, \text{FromStartToEnd} \rangle.$$

Figure 6.10 shows how the graphical representation for resource constraints can be used to depict the above constraints.

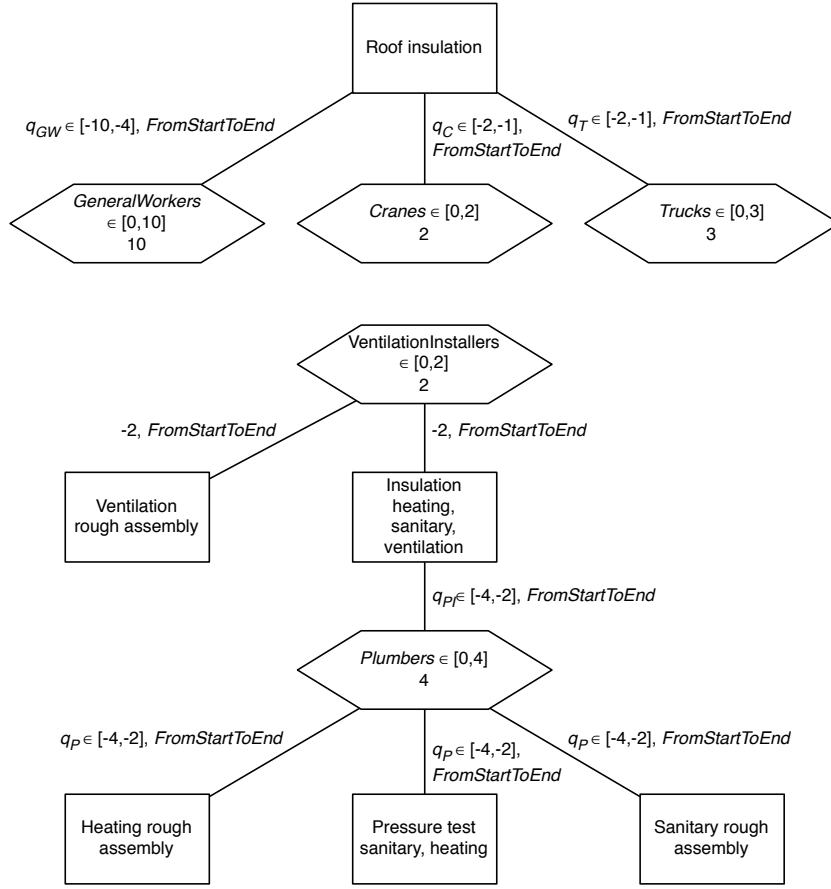


Figure 6.10: Resource constraints for the activity “Building services equipment”.

Duration constraints We consider the following duration constraints for activities in Figure 6.9.

$$d_{\text{Roof insulation}} = \frac{\text{BuildingArea}}{2 \cdot |q_{GW}| + 2 \cdot |q_T| + 3 \cdot |q_C|},$$

$$d_{\text{Ventilation rough assembly}} = \text{StoryNum} \cdot 18,$$

$$d_{\text{Heating rough assembly}} = \frac{\text{StoryNum} \cdot 27}{|q_P|},$$

$$d_{\text{Sanitary rough assembly}} = \frac{\text{StoryNum} \cdot 27}{|q_P|},$$

$$d_{\text{Pressure test sanitary heating}} = \frac{\text{StoryNum} \cdot 27}{|q_P|},$$

$$d_{\text{Insulation heating, sanitary, ventilation}} = \frac{\text{StoryNum} \cdot 12}{|qP|}.$$

6.4.4 Finishing works MART Model

Figure 6.11 shows a graphical representation of the temporal constraint network of the model of the composite activity “Finishing works”.

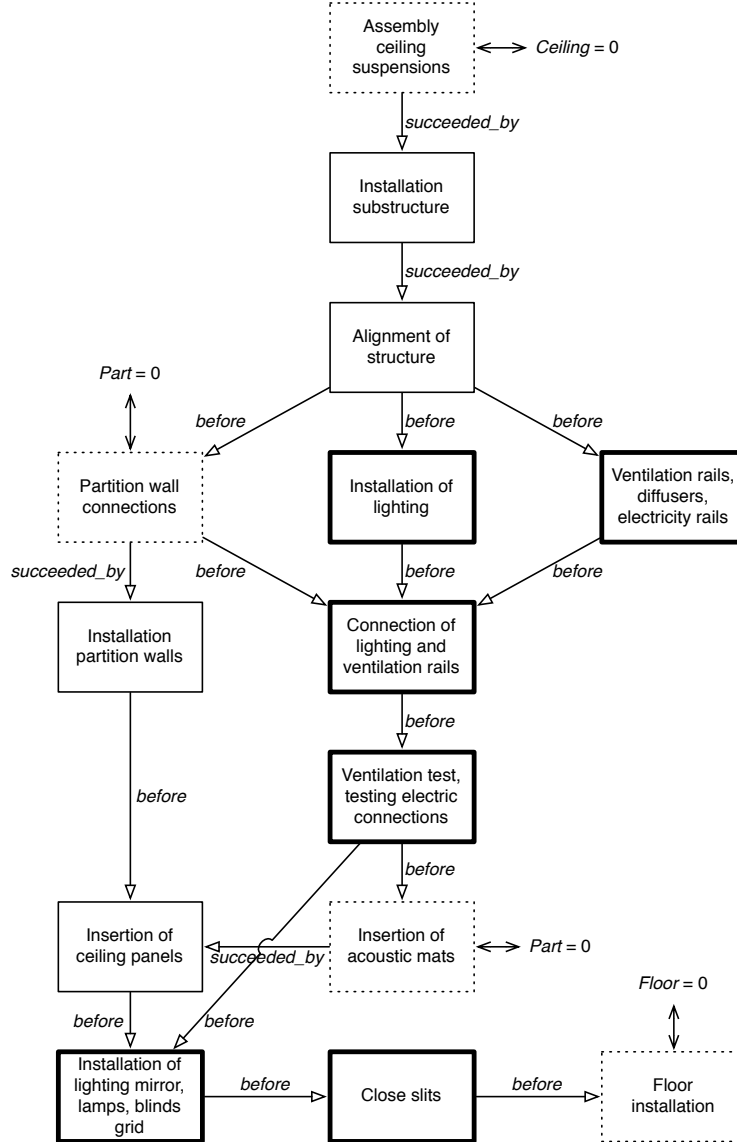


Figure 6.11: Temporal constraint network for the multiple instance composite activity “Finishing works”.

Model variables The set of model variables is defined as

$$\mathcal{V}_{\text{Main}} \cup \mathcal{V}_{\text{R}} \cup \mathcal{V}_{\text{P}},$$

where:

$$\begin{aligned}\mathcal{V}_{\text{R}} &= \{\langle \text{Electricians}, [0,4] \rangle\}, \\ \mathcal{V}_{\text{P}} &= \{\langle \text{Ceiling}, [0,1] \rangle, \\ &\quad \langle \text{Part}, [0,1] \rangle, \\ &\quad \langle \text{Floor}, [0,1] \rangle\}.\end{aligned}$$

Resource constraints We consider the following resource constraints for activities in Figure 6.11.

$\langle \text{Assembly ceiling suspensions}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -3] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Installation substructure}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -3] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Alignment of structure}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Partition wall connections}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Installation of lighting}, \text{Electricians}, \langle q_E, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Ventilation rails, diffusers, electricity rails}, \text{Electricians}, \langle q_E, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Ventilation rails, diffusers, electricity rails}, \text{VentilationInstallers}, -2, \text{FromStartToEnd} \rangle,$

$\langle \text{Installation partition walls}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Connection of lighting, and ventilation rails}, \text{Electricians}, \langle q_E, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Connection of lighting, and ventilation rails}, \text{VentilationInstallers}, -2, \text{FromStartToEnd} \rangle,$

$\langle \text{Ventilation test, testing electric connections}, \text{Electricians}, \langle q_E, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Ventilation test, testing electric connections}, \text{VentilationInstallers}, \langle q_V, [-2, -1] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Insertion of acoustic mats}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Insertion of ceiling panels}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Installation of, lighting mirror, lamps, blind grid}, \text{Electricians}, \langle q_E, [-4, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Close slits}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\langle \text{Floor installation}, \text{GeneralWorkers}, \langle q_{GW}, [-10, -2] \rangle, \text{FromStartToEnd} \rangle,$

$\text{initialValue}(\text{Electricians}, 4).$

Duration constraints We consider the following duration constraints for activities in Figure 6.11.

$$d_{\text{Assembly ceiling suspensions}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Installation substructure}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Alignment of structure}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Partition wall connections}} = \frac{\text{BuildingArea}}{|q_{GW}|},$$

$$d_{\text{Installation of lighting}} = \frac{\text{BuildingArea}}{4 \cdot |q_E|},$$

$$d_{\text{Ventilation rails, diffusers, electricity rails}} = \frac{\text{BuildingArea}}{4 \cdot |q_E|},$$

$$d_{\text{Installation partition walls}} = \frac{\text{BuildingArea}}{|q_{GW}|},$$

$$d_{\text{Connection of lighting, and ventilation rails}} = \frac{\text{BuildingArea}}{4 \cdot |q_E|},$$

$$d_{\text{Ventilation test, testing electric connections}} = \frac{\text{BuildingArea}}{4 \cdot |q_E|},$$

$$d_{\text{Insertion of acoustic mats}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Insertion of ceiling panels}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Installation of, lighting mirror, lamps, blind grid}} = \frac{\text{BuildingArea}}{4 \cdot |q_E|},$$

$$d_{\text{Close slits}} = \frac{\text{BuildingArea}}{4 \cdot |q_{GW}|},$$

$$d_{\text{Floor installation}} = \frac{\text{BuildingArea}}{2 \cdot |q_{GW}|}.$$

6.5 Coupling of the Models

We define the coupling of the product model presented in Section 6.3 with the process model presented in Section 6.4, through a set $\mathcal{C}_{\text{Coup}}$ of coupling constraints.

$$\begin{aligned} \mathcal{C}_{\text{Coup}} = \{ & \langle \text{sanitary}, \text{Building}, \text{Sanitary service}, \text{Card} \rangle = \text{San}, \\ & \langle \text{heating}, \text{Building}, \text{Heating service}, \text{Card} \rangle = \text{Heat}, \\ & \langle \text{ventilation}, \text{Building}, \text{Ventilation service}, \text{Card} \rangle = \text{Vent}, \\ & \langle \text{ceiling}, \text{Story}, \text{Suspended ceiling}, \text{Card} \rangle = \text{Ceiling}, \\ & \langle \text{walls}, \text{Story}, \text{Partition wall system}, \text{Card} \rangle = \text{Part}, \\ & \langle \text{floor}, \text{Story}, \text{Floor}, \text{Card} \rangle = \text{Floor}, \\ & \langle \text{StoryNum}, \text{Building}, [] \rangle = \text{inst}_{\text{Finishing works}}, \\ & \langle \text{Width}, \text{Building}, [] \rangle \cdot \langle \text{Length}, \text{Building}, [] \rangle = \text{BuildingArea} \\ & \langle \text{basement}, \text{Building}, \text{Card} \rangle = \text{Basement} \\ & \langle \text{StoryNum}, \text{Building}, [] \rangle = \text{StoryNum} \}. \end{aligned}$$

Chapter 7

CSP Encoding of the Product/Process Configuration Problem

In this chapter we explain how a PRODPROC based configuration system can support a user through the configuration of a PRODPROC model, encoding a configuration problem into a Constraint Satisfaction Problem (CSP) and using Constraint Logic Programming (CLP) [JM94] for checking the validity of user's choices with respect to a model, and reacting to user's inputs by propagating their effects.

7.1 A CSP-based Configuration System

A possible general structure of a configuration process supported by a CSP-based system is pictorially described in Figure 7.1. First the user initializes the system (1) selecting the model to be configured. After such an initialization phase the user can start making her/his choices using the system interface (2). The interface communicates to the system engine (i.e., the piece of software that maintains a representation of the product/process under configuration, and checks the validity and consistency of user's choices) each data variation specified by the user (3) and the system engine updates the current partial configuration accordingly. Whenever an update of the (partial) configuration takes place, the user, through the system interface, can activate the engine inference process (4). The engine instantiates model constraints on (partial) configuration variables (cf. Section 5.4) and encodes the product/process configuration problem in a CSP. Then it uses a finite domain solver to propagate the logical effects of user's choices (5). Once the inference process ends (6), the engine returns to the interface the results of its computation (7). In its turns, the system interface communicates to the user the consequences of her/his choices on the (partial) configuration (8).

In the following sections, we give an exhaustive description of the CSP encoding we proposed in [CF11]. In particular, we show how it is possible to encode in a CSP a product configuration problem (Section 7.2), a process configuration problem (Section 7.3), and a

product and process configuration problem (Section 7.4). Moreover, we explain how CLP can be used for checking the validity of user's choices with respect to a model. We will use the SWI Prolog syntax for finite domain constraints to present the CSP encoding. The SWI Prolog syntax for finite domain constraints is close to both the syntax of PRODPROC constraints, and the one provided by similar CLP systems.

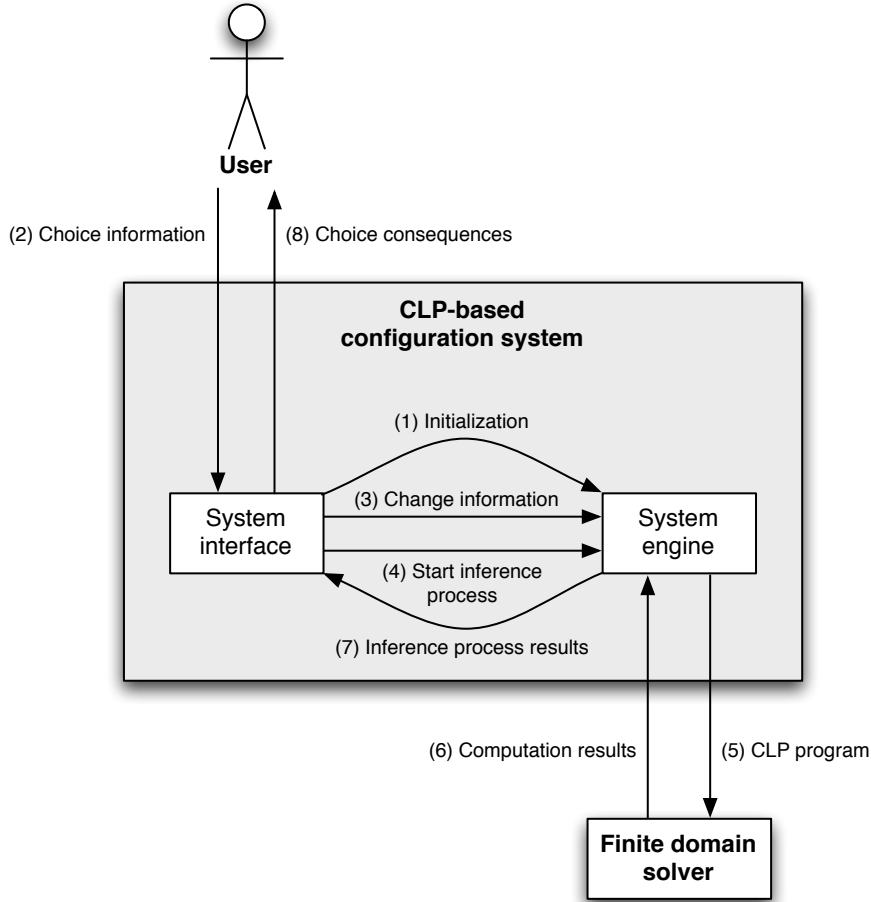


Figure 7.1: Configuration process supported by a CLP-based system.

7.2 Encoding of the Product Configuration Problem

Given a PROMO model and a candidate PROMO instance (or a partial candidate instance), we can generate a CSP to check if it is a PROMO instance (or to compute the consequences of user's choices on the partial candidate instance). Basically, we define a triple $CSP_{\text{Prod}} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where \mathcal{V} is a tuple of finite domain (FD) variables, \mathcal{D} is a tuple of domain constraints for variables in \mathcal{V} , and \mathcal{C} is a tuple of constraints on variables in \mathcal{V} . We

compute the three tuples using three functions, respectively, $\tau_{\text{Prod}}^{\mathcal{V}}$, $\tau_{\text{Prod}}^{\mathcal{D}}$ and $\tau_{\text{Prod}}^{\mathcal{C}}$. Such functions maps variables, terms, and constraints into the corresponding entities of a CSP formulated in Constraint Logic Programming.

In Table 7.1, we show the way in which basic arithmetic operators are denoted in SWI Prolog. Sets defining variable domains are represented using the union operator \setminus , e.g., the set $\{1,2,5\}$ is represented as $1 \setminus 2 \setminus 5$; intervals $[a, b]$ are compactly represented as $a..b$.

Relational symbols:		Boolean constructors:	
Abstract syntax	SWI Concrete Syntax	Abstract syntax	SWI Concrete Syntax
$=$	$\# =$	\neg	$\# \setminus$
\neq	$\# \setminus =$	\vee	$\# \setminus /$
$< \quad (>)$	$\# < \quad (\# >)$	\wedge	$\# / \setminus$
$\leq \quad (\geq)$	$\# = < \quad (\# > =)$	\Rightarrow	$\# == >$

Table 7.1: SWI concrete syntax for finite domains operators.

Let M_{Prod} be a PROMO model, and let $I_{M_{\text{Prod}}} = \langle IT, \mathcal{A} \rangle$ be a (partial) candidate PROMO instance of M_{Prod} . Variables of the CSP, i.e., the tuple \mathcal{V} of CSP_{Prod} , correspond to variables of nodes instances belonging to the (partial) candidate instance. They are identified by the concatenation of three elements, namely, the variable V of the instance with id i of a node N :

$$\begin{aligned} \tau_{\text{Prod}}^{\mathcal{V}}(M_{\text{Prod}}, I_{M_{\text{Prod}}}) &= \{ \text{N_i_V} \mid \langle N, i, \mathcal{V}_N \rangle \in \mathcal{N} \wedge V \in \mathcal{V}_N \} \cup \\ &\cup \{ \text{IC_e_N_i} \mid \langle N, i, \mathcal{V}_N \rangle \in \mathcal{N} \wedge e = \langle \text{label}, N, M, \text{Card}, \text{CC} \rangle \in M_{\text{Prod}} \}. \end{aligned}$$

The Prolog variables N_i_V , IC_e_N_i are used as FD variables. The domain of N_i_V is the domain $D(V)$ of the variable V . The domain of IC_e_N_i is the domain $D(\text{Card})$, of the variable Card . Due to the lack of a tool for string handling in the finite domain solver of SWI Prolog, we define a bijection $B : \text{strings} \rightarrow \mathbb{Z}$ that associates an integer with each string belonging to the domain of a variable of sort **string**. The domain constraints for variables are encoded as follows.

$$\begin{aligned} \tau_{\text{Prod}}^{\mathcal{D}}(M_{\text{Prod}}, I_{M_{\text{Prod}}}) &= \{ \text{ndomain}(N, i, V) \mid \langle N, i, \mathcal{V}_N \rangle \in \mathcal{N} \wedge V \in \mathcal{V}_N \} \cup \\ &\cup \{ \text{cdomain}(N, i, e, \text{Card}) \mid \langle N, i, \mathcal{V}_N \rangle \in \mathcal{N}^I \wedge e = \langle \text{label}, N, M, \text{Card}, \text{CC} \rangle \in M_{\text{Prod}} \}, \end{aligned}$$

where:

- $\text{ndomain}(N, i, \langle V, \{i_1, \dots, i_n\} \rangle) = \text{N_i_V in } i_1 \setminus \dots \setminus i_n$,
if i_1, \dots, i_n are integers;
- $\text{ndomain}(N, i, \langle V, [\text{min}, \text{max}] \rangle) = \text{N_i_V in min .. max}$;

- $\text{ndomain}(N, i, \langle V, \{s_1, \dots, s_n\} \rangle) = \text{N_i_V in } B(s_1) \setminus \dots \setminus B(s_n)$, if s_1, \dots, s_n are ASCII strings;
- $\text{cdomain}(N, i, e, \langle \text{Card}, \{i_1, \dots, i_n\} \rangle) = \text{IC_e_N_i in } i_1 \setminus \dots \setminus i_n$, if i_1, \dots, i_n are integers;
- $\text{cdomain}(N, i, e, \langle \text{Card}, [\text{min}, \text{max}] \rangle) = \text{IC_e_N_i in min} \dots \text{max}$.

To give some examples of variable and domain encoding, let us consider the the bicycle product model and instance tree introduced in Chapter 5. The variables *FrameType* of node *Frame*, and *SpokeNumber* of node *Wheel*, are encoded as follows.

```

Frame_1_FrameType in 1 \ / 2,
Wheel_1_SpokeNumber in 18..24,
Wheel_2_SpokeNumber in 18..24.
    
```

The values 1 and 2 of the domain of *Frame_1_FrameType* are the encoding of the string “Racing bike” and “City bike” respectively. We have two variables *SpokeNumber* since in the instance tree presented in Chapter 5 there are two nodes *Wheel*.

To define the function τ_{Prod}^C we need to define a transformation function τ that maps (variables, terms and) constraints into the corresponding SWI Prolog entities. The result of the application of the translation function τ to (terms and) constraints is inductively defined as follows.

- If $t = V$ and V is a variable of $\langle N, i, \mathcal{V}_N \rangle$, then $\tau(t) = \text{N_i_V}$.
- If $t = \text{IC}_n^e$ and $n = \langle N, i, \mathcal{V}_N \rangle$, then $\tau(t) = \text{IC_e_N_i}$.
- If t is a constant of sort *int*, then $\tau(t) = t$.
- If t is a constant of sort *string*, then $\tau(t) = B(t)$.
- If t_1, t_2 are terms of sort *int* and $op \in \{+, -, *, /, \text{mod}\}$, then we have $\tau(t_1 \text{ op } t_2) = \tau(t_1) \text{ op } \tau(t_2)$ (there is obviously a slight abuse of notation here, as the arithmetical function symbol $+$ denotes both the sum on integers and the sum on finite domains; the same holds for the other operators). Moreover, we have $\tau(|t_1|) = \text{abs}(\tau(t_1))$.
- If $t_1 \text{ op } t_2$ is a primitive constraint, then $\tau(t_1 \text{ op } t_2) = \tau(t_1) \text{ FDop } \tau(t_2)$, where *FDop* is the finite domain version of the binary operator *op*, that is, $\# =$ for $=$, $\# <$ for $<$, and so on, as in Table 7.1.
- If C_1 is a constraint, then $\tau(\neg C_1) = \# \setminus \tau(C_1)$.
- If C_1 and C_2 are constraints and *Boolop* is the binary Boolean operator \wedge (resp., $\vee, \Rightarrow, \Leftrightarrow$), then $\tau(C_1 \text{ Boolop } C_2) = \tau(C_1) \text{ FDBoolop } \tau(C_2)$, where *FDBoolop* is the finite domain binary Boolean operator $\# \wedge$ (resp., $\# \vee, \# \Rightarrow, \# \Leftrightarrow$).

- A *valid_tuples* constraint instantiated on variables V_1 of node instance $(n_1)_{j_1}$, V_2 of node instance $(n_2)_{j_2}$, ..., V_k of node instance $(n_k)_{j_k}$ ($(n_1)_{j_1}, \dots, (n_k)_{j_k}$ need not to be pairwise distinct), which defines the set of admissible tuples

$$\{\langle t_{1,1}, \dots, t_{1,k} \rangle, \dots, \langle t_{h,1}, \dots, t_{h,k} \rangle\},$$

is translated into the following SWI Prolog DNF constraint:

```
tuples_in([[N1_j1_V1, ..., Nk_jk_Vk]],
          [[t1,1, ..., t1,k], ..., [th,1, ..., th,k]]),
```

where `tuples_in/2` is the built-in combinatorial global constraint of SWI Prolog and j_i , for $i = 1, \dots, k$, is the id of an instance of node N_i .

- An *allDifferent* constraint instantiated on variables V_1 of node instance $(n_1)_{j_1}$, V_2 of node instance $(n_2)_{j_2}$, ..., V_k of node instance $(n_k)_{j_k}$ ($(n_1)_{j_1}, \dots, (n_k)_{j_k}$ need not to be pairwise distinct), is translated into the following SWI Prolog constraint:

```
all_different([N1_j1_V1, ..., Nk_jk_Vk]),
```

where `all_different/2` is the built-in global constraint of SWI Prolog and j_i , for $i = 1, \dots, k$, is the id of an instance of node N_i .

- An *aggConstraint*(f, L, op, n) constraint instantiated on variables V_1 of node instance $(n_1)_{j_1}$, V_2 of node instance $(n_2)_{j_2}$, ..., V_k of node instance $(n_k)_{j_k}$ (node instances $(n_1)_{j_1}, \dots, (n_k)_{j_k}$ need not to be pairwise distinct), is translated into a SWI Prolog constraint as follows:

- if $f = \text{sum}$ the constraint is translated into

```
sum([N1_j1_V1, ..., Nk_jk_Vk], FDop, n),
```

where `sum/3` is the built-in global constraint of SWI Prolog, and j_i , for $i = 1, \dots, k$, is the id of an instance of node N_i .

- if $f = \text{avg}$ the constraint is translated into

```
(N1_j1_V1 + ... + Nk_jk_Vk / varNum) FDop n,
```

where `FDop` is the finite domain version of the binary operator op , and `varNum` is the number of variables involved in the constraint.

The constraints of CSP_{Prod} are obtained applying τ to the assignments on instance variables, and to the elements of the compatibility relation pool, appropriately instantiated with variables of node instances in the (partial) candidate instance through the mechanism described in Section 5.4.1.

Let us consider again the the bicycle product model and instance tree shown in Chapter 5. The instantiated constraints 5.3 and 5.4 presented in Chapter 5 are encoded by τ as follows.

```
Wheel_1_WheelType #= Frame_1_FrameType,
```

```
Wheel_2_WheelType #= Frame_1_FrameType.
```

Applying τ to the instantiated constraint 5.5, also presented in Chapter 5, we get the following reified constraint.

`Frame_1_FrameType #= 1 #==> Wheel_2_SpokeNumber > 20.`

The union of the set of instantiated constraints and the set of assignments on instance variables, i.e., \mathcal{A} , called *product instance constraint pool* ($\mathcal{ICP}_{\text{Prod}}$), can be defined in terms of μ_{Prod} as follows:

$$\mathcal{ICP} = \{\mu_{\text{Prod}}(c, IT) \mid c \in \mathcal{CRP}\} \cup \mathcal{A},$$

where \mathcal{CRP} is the compatibility relation pool of M_{Prod} . The function $\tau_{\text{Prod}}^{\mathcal{C}}$ is defined as follows:

$$\tau_{\text{Prod}}^{\mathcal{C}}(\mathcal{ICP}_{\text{Prod}}) = \{\tau(c) \mid c \in \mathcal{ICP}_{\text{Prod}}\}.$$

The CSP associated to a (partial) PROMO instance is then $\text{CSP}_{\text{Prod}} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where:

$$\begin{aligned} \mathcal{V} &= \tau_{\text{Prod}}^{\mathcal{V}}(M_{\text{Prod}}, I_{M_{\text{Prod}}}), \\ \mathcal{D} &= \tau_{\text{Prod}}^{\mathcal{D}}(M_{\text{Prod}}, I_{M_{\text{Prod}}}), \\ \mathcal{C} &= \tau_{\text{Prod}}^{\mathcal{C}}(\mathcal{ICP}_{\text{Prod}}). \end{aligned}$$

7.3 Encoding of the Process Configuration Problem

Given a MART model and a candidate MART instance (or a partial candidate instance), we can generate a CSP to check if it is a MART instance (or to compute the consequences of user's choices on the partial candidate instance). Basically, we define a triple $\text{CSP}_{\text{Proc}} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where \mathcal{V} is a tuple of FD variables, \mathcal{D} is a tuple of domain constraints for variables in \mathcal{V} , and \mathcal{C} is a tuple of constraints on variables in \mathcal{V} . We compute the three tuples using three functions, respectively, $\tau_{\text{Proc}}^{\mathcal{V}}$, $\tau_{\text{Proc}}^{\mathcal{D}}$ and $\tau_{\text{Proc}}^{\mathcal{C}}$. Such functions maps variables, terms, and constraints into the corresponding entities of a CSP formulated in Constraint Logic Programming. As for the encoding of a product configuration problem, constraints on finite domains are formulated according to the concrete syntax of SWI Prolog.

Let M_{Proc} be a MART model, and $I_{M_{\text{Proc}}} = \langle \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$ be a (partial) candidate MART instance. Variables of the CSP correspond to model variables and parameters of activity instances in the (partial) candidate MART instance. Let \mathcal{V}_{R} be the set of resource variables in the MART model and in MART models of its composite activities. Let \mathcal{V}_{P} be the set of process variables in the MART model and in MART models of its composite activities. The function $\tau_{\text{Proc}}^{\mathcal{V}}$ is defined as follows.

$$\begin{aligned} \tau_{\text{Proc}}^{\mathcal{V}}(M_{\text{Proc}}, I_{M_{\text{Proc}}}) &= \{\text{var}(V) \mid V \in \mathcal{V}_{\text{R}} \cup \mathcal{V}_{\text{P}}\} \cup \\ &\cup \{\text{actvars}(A, i) \mid \langle A, i \rangle \in \mathcal{I}\} \cup \{\text{actvars}(A) \mid \text{exec}_A \in \mathcal{F}\} \cup \\ &\cup \{\text{Inst_A} \mid A \text{ multiple instance activity in } M_{\text{Proc}}\} \cup \\ &\cup \{\text{Q_R_A_i} \mid \langle A, i \rangle \in \mathcal{I} \wedge \langle A, R, q, TE \rangle \in M_{\text{Proc}} \wedge q \notin \mathbb{Z}\}, \end{aligned}$$

where:

$$\text{var}(V) = \begin{cases} \text{v_p} & \text{if } V \in \mathcal{V}_P \\ \text{v_r} & \text{if } V \in \mathcal{V}_R \end{cases}$$

$$\text{actvars}(A, i) = \{\text{T_start_A_i}, \text{T_end_A_i}, \text{D_A_i}, \text{Exec_A}\},$$

$$\text{actvars}(A) = \{\text{T_start_A}, \text{T_end_A}, \text{Exec_A}\}.$$

Domain constraints for variables in \mathcal{V} are generated by the function $\tau_{\text{Proc}}^{\mathcal{D}}$, that is defined in the following way.

$$\begin{aligned} \tau_{\text{Proc}}^{\mathcal{D}}(M_{\text{Proc}}, I_{M_{\text{Proc}}}) = & \{\text{vdomain}(V) \mid V \in \mathcal{V}_R \cup \mathcal{V}_P\} \cup \\ & \cup \bigcup_{\langle A, i \rangle \in \mathcal{I}} \text{adomains}(A, i) \cup \\ & \cup \{\text{edomain}(A) \mid \text{exec}_A \in \mathcal{F}\} \cup \\ & \cup \{\text{idomain}(A) \mid A \text{ multiple instance activity in } M_{\text{Proc}}\} \\ & \cup \{\text{qdomain}(q, R, A, i) \mid \langle A, i \rangle \in \mathcal{A}^I \wedge \langle A, R, q, TE \rangle \in M_{\text{Proc}} \wedge q \notin \mathbb{Z}\}, \end{aligned}$$

where:

$$\begin{aligned} \text{vdomain}(V) = & \begin{cases} \text{v_p in domain}(D(V)) & \text{if } V \in \mathcal{V}_P \\ \text{v_r in domain}(D(V)) & \text{if } V \in \mathcal{V}_R \end{cases} \\ \text{adomains}(A, i) = & \{\text{Exec_A in } [0..1], \text{D_A_i in domain}(D(d_A))\}, \\ \text{edomain}(A) = & \text{Exec_A in } [0..1], \\ \text{idomain}(A) = & \text{Exec_A in domain}(D(\text{inst}_A)), \\ \text{qdomain}(q, R, A, i) = & \text{Q_R_A_i in domain}(D(q)). \end{aligned}$$

We have that:

- $\text{domain}(\{i_1, \dots, i_n\}) = i_1 \setminus / \dots \setminus / i_n$, if i_1, \dots, i_n are integers;
- $\text{domain}([\min, \max]) = \min \dots \max$;
- $\text{domain}(\{s_1, \dots, s_n\}) = B(s_1) \setminus / \dots \setminus / B(s_n)$, if s_1, \dots, s_n are ASCII strings.

To define the function $\tau_{\text{Proc}}^{\mathcal{C}}$ we need to add some cases to the definition of the transformation function τ introduced in Section 7.2. Let $a = \langle A, i \rangle$ be an activity instance, we have that:

- $\tau(t_a^{\text{start}}) = \text{T_start_A_i}$;
- $\tau(t_a^{\text{end}}) = \text{T_end_A_i}$;
- $\tau(d_a) = \text{D_A_i}$;

- $\tau(exec_A) = Exec_A$;
- $\tau(q_a^R) = Q_R_A_i$;
- $\tau(t_A^{start}) = T_start_A$;
- $\tau(t_A^{end}) = T_end_A$;
- $\tau(inst_A) = Inst_A$;
- $\tau(t_a^{start} = \min_{b \in pInsts(a)} t_b^{start}) = \text{minimum}(T_start_A_i, Ts)$, where Ts is the list

$$[T_start_B_j \mid \langle B, j \rangle \in pInsts(A)].$$

$\text{minimum}(X, L)$ is a constraint that is true if the FD variable X is the minimum of the list of FD variables L .

- $\tau(t_a^{end} = \max_{b \in pInsts(a)} t_b^{end}) = \text{maximum}(T_end_A_i, Ts)$, where Ts is the list

$$[T_end_B_j \mid \langle B, j \rangle \in pInsts(a)].$$

$\text{maximum}(X, L)$ is a constraint that is true if the FD variable X is the maximum of the list of FD variables L .

Given an instantiated temporal constraint c , let c^P be the corresponding propositional formula. Let $iacts(c) = \{A \mid A \text{ or } \langle A, i \rangle \text{ appears in } c\}$. We have that:

$$\tau(c) = \begin{cases} \tau(\text{pre}(c)) \# ==> \tau(c^P) & \text{if } c \notin Other \\ \tau(c^P) & \text{otherwise} \end{cases},$$

where

$$\text{prec}(c) = \bigwedge_{A \in iacts(c)} exec_A = 1,$$

$$Other = \{A \text{ must_be_executed}, A \text{ is_absent},$$

$$A \text{ not_co_existent_with } B, A \text{ succeeded_by } B\}.$$

For the encoding of resource constraints, we define a function $\tau_{Cumulative}$ that, given a set of instantiated resource constraints \mathcal{IR} and the set \mathcal{V}_R , generates a **cumulatives** global constraint encoding the scheduling problem implicitly defined in the MART (partial) candidate instance. Such a global constraint does not exist in either SWI Prolog or other CLP systems. However, it can be implemented using, for example, the algorithms described in [Lab03].

$$\tau_{Cumulative}(\mathcal{IR}, \mathcal{V}_R) = \text{cumulatives}(Tasks, Machines),$$

where $Tasks$ is the list

$$\left[\text{task}(T_start_A_i, D_A_i, T_end_A_i, Q, V_r, TE, T, \tau(C)) \mid \right. \\ \left. a = \langle A, i \rangle, \langle a, V, q_a, TE, T, C \rangle \in \mathcal{IR} \wedge ((q_a \in \mathbb{Z} \Rightarrow Q = q_a) \vee Q = \tau(q_a)) \right],$$

and *Machines* is the list

$$\left[\text{machine}(\mathbf{v_r}, D(V), n) \mid V \in \mathcal{V}_R \wedge \right. \\ \left. \wedge ((\text{initialLevel}(V, iv) \in \mathcal{IR} \Rightarrow n = iv) \vee n = \min(D(V))) \right].$$

The union of the set of instantiated constraints and the set of assignments on instance variables, i.e., \mathcal{A} , called *process instance constraint pool* ($\mathcal{ICP}_{\text{Proc}}$), can be defined in terms of μ_{Proc} as follows:

$$\mathcal{ICP}_{\text{Proc}} = \mu_{\text{Proc}}(\mathcal{J}, \mathcal{RDC}, \mathcal{I}) \cup \mathcal{A},$$

where \mathcal{RDC} is the set of resources constraints, activity duration constraints, and temporal constraints in M_{Proc} , and \mathcal{I} is the set of implicit constraints on instance number variables. The function $\tau_{\text{Proc}}^{\mathcal{C}}$ is defined as follows:

$$\tau_{\text{Prod}}^{\mathcal{C}}(\mathcal{ICP}_{\text{Proc}}) = \{\tau(c) \mid c \in \mathcal{ICP}_{\text{Proc}} \setminus \mathcal{IR}\} \cup \{\tau_{\text{Cumulative}}(\mathcal{IR}, \mathcal{V}_R)\},$$

where $\mathcal{IR} \subset \mathcal{ICP}_{\text{Proc}}$ is the set of instantiated resource constraints. The CSP associated to a (partial) MART instance is then $\mathcal{CSP}_{\text{Proc}} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where:

$$\begin{aligned} \mathcal{V} &= \tau_{\text{Proc}}^{\mathcal{V}}(M_{\text{Proc}}, I_{M_{\text{Proc}}}), \\ \mathcal{D} &= \tau_{\text{Proc}}^{\mathcal{D}}(M_{\text{Proc}}, I_{M_{\text{Proc}}}), \\ \mathcal{C} &= \tau_{\text{Proc}}^{\mathcal{C}}(\mathcal{ICP}_{\text{Proc}}). \end{aligned}$$

To show some examples of constraint of $\mathcal{CSP}_{\text{Proc}}$, let us consider the bicycle production process introduced in Chapter 5. In particular, let us consider the constraint network depicted in Figure 5.12. For the composite activity “Construction of bicycle components” we will have the following constraints in $\mathcal{CSP}_{\text{Proc}}$ (in the constraints we abbreviate “Construction of bicycle components” to “CBC”, “Frame construction” to “F”, “Partial assembly” to “PA”, “Handlebar construction” to “H”, “Wheels construction” to “W”, and “Construction of other components” to “CC”).

```
minimum(T_start_CBC_1, [T_start_F_1, T_start_PA_1, T_start_H_1,
                        T_start_W_1, T_start_CC_1]),
maximum(T_end_CBC_1, [T_end_F_1, T_end_PA_1, T_end_H_1,
                      T_end_W_1, T_end_CC_1]),
T_start_CBC_1 #>= 0, T_end_CBC_1 #>= T_start_CBC_1,
D_CBC_1 #= T_end_CBC_1 - T_start_CBC_1,
Exec_CBC #= 1.
```

They are obtained applying τ to the constraint generated by the function t^{Comp} on “Construction of bicycle components”. The constraint *before* between “Frame construction”

and “Partial assembly” is encoded as the following reified constraint

$$T_start_F_1 \#< T_start_PA_1 \#/\backslash T_end_F_1 \#< T_start_PA.$$

Finally, the duration constraints for the activity “Bicycle assembly” (cf. Section 5.3.2) is encoded as the following constraint (we abbreviate “Bicycle assembly” to “BA”)

$$D_BA_1 \# = (3 * V_Components) / abs(Q_Worker_BA_1).$$

7.4 Encoding of the Product/Process Configuration Problem

Given a PRODPROC model and a candidate PRODPROC instance (or a partial candidate instance) we can generate a CSP to check if it is a PRODPROC instance (or to compute consequence’s of user choices on the partial instance). Basically we define a triple $CSP_{ProdProc} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where \mathcal{V} is a tuple of FD variables, \mathcal{D} is a tuple of domain constraints for variables in \mathcal{V} , and \mathcal{C} is a tuple of constraints on variables in \mathcal{V} . Let $M_{ProdProc} = \langle M_{Prod}, M_{Proc}, C_{Couple} \rangle$ be a PRODPROC model, and let $I_{M_{ProdProc}} = \langle I_{M_{Prod}}, I_{M_{Proc}} \rangle$ be a (partial) candidate instance of $M_{ProdProc}$. Moreover, let $CSP_{Proc} = \langle \mathcal{V}_{Proc}, \mathcal{D}_{Proc}, \mathcal{C}_{Proc} \rangle$ and $CSP_{Prod} = \langle \mathcal{V}_{Prod}, \mathcal{D}_{Prod}, \mathcal{C}_{Prod} \rangle$. Finally, let \mathcal{IC}_{Couple} be the set of instantiated coupling constraints obtained from C_{Couple} . We have that $CSP_{ProdProc} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where:

$$\mathcal{V} = \mathcal{V}_{Prod} \cup \mathcal{V}_{Proc},$$

$$\mathcal{D} = \mathcal{D}_{Prod} \cup \mathcal{D}_{Proc},$$

$$\mathcal{C} = \mathcal{C}_{Prod} \cup \mathcal{C}_{Proc} \cup \{\tau(c) \mid c \in \mathcal{IC}_{Couple}\}.$$

Given $CSP_{ProdProc}$ it is easy to write a SWI Prolog program of the following form.

```
:- use_module(library(clpfd)).

csp_prodProc(Vars) :-
    DOMAINS,
    CONSTRAINTS.
```

In it **Vars** is the list of variables in \mathcal{V} , **DOMAINS** is the conjunction of domain constraints in \mathcal{D} , and **CONSTRAINTS** is the conjunction of constraints in \mathcal{C} .

Given a program with the above-described characteristics, the finite domain solver of SWI Prolog can be used to reduce the domains associated with variables, preserving satisfiability, or to detect the inconsistency of the encoded CSP (due to user’s assignments that violate constraints or to inconsistencies of the original PRODPROC model). Moreover, it can be used to determine that further node instances and/or activity instances are needed, or that there are too many nodes in the instance tree.

The CSP encoding presented in this chapter can also be used within an algorithm for automatic configuration. Such an algorithm, given a PRODPROC model, may obtain a PRODPROC instance performing a search in a tree where each node corresponds to a partial candidate instance, and exploiting the CSP encoding to prune branches of this tree.

Chapter 8

GCSP Encoding of PRODPROC Models

In this chapter, exploiting a simple example, we show how it is possible to encode a PRODPROC model into a Generative Constraint Satisfaction Problem (GCSP). The presented mapping shows that GCSP are well-suited for modeling product and process configuration problem. Also, the mapping suggests a possible way to give a constraint-based semantics to PRODPROC.

8.1 Generative Constraint Satisfaction Problems

Generative Constraint Satisfaction Problems (GCSPs) [SFH98] are an extension of Constraint Satisfaction Problems (CSPs) that has been proposed to overcome the limitation of the latter in supporting product configuration modeling. In fact, traditional CSPs demonstrate not completely adequate in modeling configuration problems whenever the number of components to be configured is unknown a priori. Generative constraint satisfaction represents an AI technique suitable for mastering this challenge. Using such a technique one models and solves a problem by exploiting a constraint network which is extended during the configuration process. Generative constraints hold for all components of a given type and are used as generators for extending the configuration including the constraint network.

In recalling the notion of Generative Constraint Satisfaction Problem (GCSP) (cf. Section 8.1.1) we briefly describe some of its applications (cf. Section 8.1.2). Further details can be found in the mentioned literature and in the references therein.

8.1.1 Generative CSPs and Configuration Problems

We state here the definition of GCSP we will use in the sequel. Such as in [SFH98, SH93, MTS09], this definition is tailored to the context of configuration domains. For a more general definition see [ZJSF08].

Definition 6 (GCSP). Let \mathcal{T} be the set of available components types, and let \mathcal{A} be the set of port and attribute names. A GCSP is a quadruple $\langle \mathcal{X}, \Gamma, \mathcal{C}, \mathcal{P} \rangle$, where: \mathcal{X} is a set of placeholder variables; Γ is a set of generic constraints over \mathcal{X} ; \mathcal{C} is an infinite set of constraint variables representing component instances; $\mathcal{P} = \{C.a \mid C \in \mathcal{C}, a \in \mathcal{A}\}$ is an infinite set of constraint variables representing the components attributes and ports. Each component variable $C \in \mathcal{C}$ may be assigned a value $t \in \mathcal{T}$ ($C \sqsubseteq t$), whereas attribute/port variables $P \in \mathcal{P}$ may choose their value from \mathcal{C} (if P represents a port connecting to another component), or may take on a numeric or a string value (if P represents an attribute).

Each variable $C \in \mathcal{C}$ may be either *active* or *inactive*. C is active if it corresponds to a component that is part of the solution of a configuration problem, otherwise it is inactive. Similarly, for variables in \mathcal{P} .

A *generic constraint* in Γ is a logical implication of the form $\Phi \Rightarrow \Psi$ over variables in \mathcal{X} . Φ represents the precondition of the constraint and Ψ specifies the variables and constraints that are introduced into the CSP if Φ holds for some instantiation of \mathcal{X} in \mathcal{C} . A generic constraint is satisfied if it holds for all the possible substitutions of the placeholder variables with active component variables.

A configuration problem consists of a GCSP (i.e., a knowledge base comprising component types and the generic constraints governing them), a set of initial variables (i.e., specific components that are required to be part of a solution), and a set of additional constraint instances (for expressing problem requirements).

Definition 7 (Configuration Problem). Let $\mathcal{M} = \langle \mathcal{X}, \Gamma, \mathcal{C}, \mathcal{P} \rangle$ be a GCSP. A configuration problem in \mathcal{M} is a tuple $\langle V^I, \Gamma^I, \mathcal{M} \rangle$, where $V^I \subseteq \mathcal{C} \cup \mathcal{P}$ is the set of initial variables, and Γ^I is the set of constraint instances on V^I .

The variables in V^I define a partial constraint network, consisting of constraints in Γ^I and the appropriate instances of generic constraints in Γ .

Definition 8 (Solution of a Configuration Problem). A solution S of a configuration problem $\langle V^I, \Gamma^I, \mathcal{M} \rangle$ is an assignment to a set of active variables V ($V \subseteq \mathcal{C} \cup \mathcal{P}$, $V^I \subseteq V$), such that all (generic) constraints are satisfied.

8.1.2 Applications of Generative CSPs

GCSPs have been first applied in the field of product configuration. In fact, GCSPs have been proposed as an extension of CSPs to overcome the limitations of the latter for modeling problem where the structure or the size of the solution is unknown a priori. In the past years, approaches based on GCSP for service composition and automated software deployment have also been proposed. We just mention few of them. In [FFH⁺98] GCSPs have been exploited to define the program semantics of ConTalk. ConTalk is the modeling language of the Cocos knowledge base configuration tool. It allows one to model a product in terms of components organized in an inheritance hierarchy. Attributes, ports, and constraints define the properties and behavior of a component. Attributes model configurable characteristics of components. Ports establish connections between two components. Constraints express compatibility knowledge between components. The ConTalk semantics is

defined by mapping a program to a Generative CSP, since constraint statements in Con-Talk are generic, i.e., they hold for all components of a given type and are used to generate new components during the configuration process. In [ZJSF08] it is presented a framework for supporting configuration tasks where knowledge is distributed over a set of agents. The proposed framework is build on the definition of GCSP, and is called Distributive GCSP. In [MTS09] the GCSP framework is extended to realize a consistency-based service composition approach, that is rich enough to capture a diverse set of service-composition principles. The service composition problem is posed as a configuration task where a set of service components and their interconnections are sought in order to satisfy a given goal. A constraint model for automated software deployment for embedded automotive system is described in [NPW08]. The authors exploit the GCSP framework to define a model capable of handling resource-, quality-, cost-, and timing constraints, and to compute optimal solution to the automated software deployment problem.

8.2 Encoding of PRODPROC Models as Generative CSPs

In this section, we show how it is possible to encode a PRODPROC model into a Generative CSP. In particular, by exploiting the bicycle example introduced in Chapter 5, we illustrate how the different modeling features of our framework can be translated into GCSP elements.

Let us reconsider the product model graph in Figure 5.3 and the following sets of variables for nodes of such model.

$$\begin{aligned}
 \mathcal{V}_{Frame} &= \{ \langle FrameType, \{Racing\ bike, City\ bike\} \rangle, \\
 &\quad \langle FrameMaterial, \{Steel, Aluminum, Carbon\} \rangle \}, \\
 \mathcal{V}_{Wheel} &= \{ \langle WheelType, \{Racing\ bike, City\ bike\} \rangle, \\
 &\quad \langle SpokeNumber, [18, 28] \rangle \}, \\
 \mathcal{V}_{Gears} &= \{ \langle GearType, \{Standard, Racing, Special\} \rangle \}, \\
 \mathcal{V}_{Junction} &= \{ \langle JunctionMaterial, \{Steel, Aluminum, Carbon\} \rangle \}.
 \end{aligned}$$

The following sets of constraints define the compatibility relations between variables' values:

$$\begin{aligned}
 \mathcal{C}_{Wheel} &= \{ \langle FrameType, Frame, [front\ wheel] \rangle = WheelType, \\
 &\quad \langle FrameType, Frame, [rear\ wheel] \rangle = WheelType, \\
 &\quad WheelType = Racing\ bike \Rightarrow SpokeNumber \leq 24, \\
 &\quad \langle FrameType, Frame, [rear\ wheel] \rangle = Racing\ bike \Rightarrow \\
 &\quad \Rightarrow SpokeNumber > 20 \},
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{C}_{Junction} &= \{ \langle \text{FrameMaterial}, \text{Frame}, [\text{rear bag}, \text{junction}] \rangle = \\
 &\quad = \text{JunctionMaterial} \}, \\
 \mathcal{CC}_{rear\ gears} &= \{ \text{FrameType} = \text{Racing bike} \Rightarrow \text{CG} = 1 \}, \\
 \mathcal{CC}_{rear\ bag} &= \{ \text{FrameType} = \text{Racing bike} \Rightarrow \text{CB} = 0 \}, \\
 \mathcal{C}_{Model} &= \{ \langle \text{GearType}, \text{Gears}, [\text{rear gears}] \rangle = \text{Special} \Rightarrow \\
 &\quad \Rightarrow \langle \text{SpokeNumber}, \text{Wheel}, [\text{rear wheel}] \rangle = 26 \}.
 \end{aligned}$$

Figure 8.1 shows the constraint network we will consider for the bicycle production process.

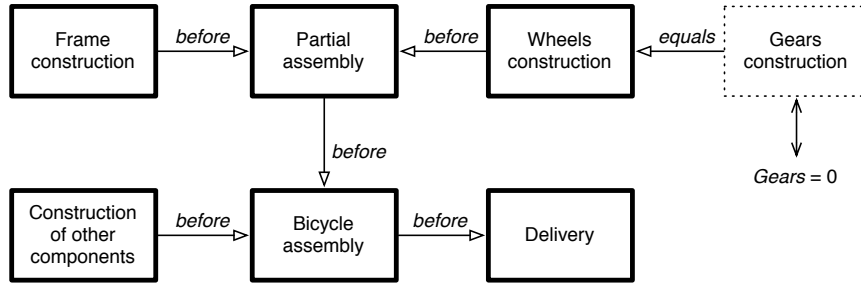


Figure 8.1: Constraint network for the bicycle production process.

The product/process description we presented in Chapter 5 can be encoded as a Generative CSP as follows. We define a set of types \mathcal{T} containing an element for each node and each activity, plus an element *Null* to represent the absence of a component, and an element *Process* that will have associated the process variables. In our example:

$$\begin{aligned}
 \mathcal{T} = \{ &\text{Frame}, \text{Wheel}, \text{Gears}, \text{Bag}, \text{Junction}, \text{Null}, \text{FrameConstruction}, \\
 &\text{PartialAssembly}, \text{WheelsConstruction}, \text{GearsConstruction}, \\
 &\text{Assembly}, \text{Delivery}, \text{OtherComponentsConstruction}, \text{Process} \}.
 \end{aligned}$$

The set \mathcal{A} of attribute and port names contains an element for each node variable name, for each process variable, and the elements *Start*, *End* and *Duration*, denoting start times, end times and durations of activities. Moreover, for each edge *edge name* whose maximum number of instances is n , \mathcal{A} contains the names $\text{EdgeName}_1, \dots, \text{EdgeName}_n$ (if $n = 1$ we write *EdgeName* instead of EdgeName_1). In our example:

$$\begin{aligned}
 \mathcal{A} = \{ &\text{FrameType}, \text{FrameMaterial}, \text{WheelType}, \text{SpokeNumber}, \text{GearType}, \\
 &\text{BagMaterial}, \text{JunctionMaterial}, \text{FrontWheel}, \text{RearWheel}, \text{RearGears} \\
 &\text{RearBag}_1, \text{RearBag}_2, \text{Junction}, \text{Start}, \text{End}, \text{Duration}, \text{Gears} \}.
 \end{aligned}$$

The set Γ of generic constraints is defined as

$$\Gamma = \Gamma_{\text{Structure}} \cup \Gamma_{\text{Domains}} \cup \Gamma_{\text{Nodes}} \cup \Gamma_{\text{Edges}} \cup \Gamma_{\text{Model}} \cup \Gamma_{\text{Process}},$$

where each of these sets, with respect to our working example, is as follows.

The constraints in $\Gamma_{Structure}$ determine the admissible values for component ports. I.e., they represent the *has-part* relations between components:

$$\begin{aligned}
 \Gamma_{Structure} = \{ & X \sqsubseteq Frame \wedge X.RearWheel = Y \wedge X.FrontWheel = Z \Rightarrow \\
 & \Rightarrow Y \sqsubseteq Wheel \wedge Z \sqsubseteq Wheel \wedge Y \neq Z \\
 & X \sqsubseteq Frame \wedge X.RearGears = Y \Rightarrow Y \sqsubseteq Gears \vee Y \sqsubseteq Null, \\
 & Y \sqsubseteq Gears \wedge X \sqsubseteq Frame \Rightarrow X.RearGears = Y, \\
 & X \sqsubseteq Frame \wedge X.RearBag_1 = B \wedge X.RearBag_2 = C \Rightarrow \\
 & \Rightarrow ((C \sqsubseteq Null \wedge B \sqsubseteq Null) \vee (B \sqsubseteq Bag \wedge C \sqsubseteq Null) \vee \\
 & \vee (B \sqsubseteq Null \wedge C \sqsubseteq Bag) \vee (B \sqsubseteq Bag \wedge C \sqsubseteq Bag)) \wedge \\
 & \wedge C \neq B, \\
 & B \sqsubseteq Bag \wedge X \sqsubseteq Frame \Rightarrow \\
 & \Rightarrow X.RearBag_1 = B \vee X.RearBag_2 = B, \\
 & B \sqsubseteq Bag \wedge B.Junction = J \Rightarrow J \sqsubseteq Junction, \\
 & J \sqsubseteq Junction \wedge B \sqsubseteq Bag \Rightarrow B.Junction = J \}.
 \end{aligned}$$

$\Gamma_{Domains}$ contains those constraints determining the admissible values for attributes of components. This defines the domains of configurable characteristic. For example, for the domains of variables of the node *Wheel*, $\Gamma_{Domains}$ includes these constraints:

$$\begin{aligned}
 X \sqsubseteq Wheel & \Rightarrow X.WheelType = T \wedge (T = \text{Racing bike} \vee T = \text{City bike}), \\
 X \sqsubseteq Wheel & \Rightarrow X.SpokeNumber = N \wedge N \geq 18 \wedge N \leq 28.
 \end{aligned}$$

Generic constraints encoding node and edges constraints belongs to the sets Γ_{Nodes} and Γ_{Edges} , respectively. In our example:

$$\begin{aligned}
 \Gamma_{Nodes} = \{ & X \sqsubseteq Frame \wedge X.FrontWheel = Y \wedge Y \sqsubseteq Wheel \wedge \\
 & \wedge X.FrameType = T \wedge Y.WheelType = T' \Rightarrow T = T', \\
 & X \sqsubseteq Frame \wedge X.RearWheel = Y \wedge Y \sqsubseteq Wheel \wedge \\
 & \wedge X.FrameType = T \wedge Y.WheelType = T' \Rightarrow T = T', \\
 & X \sqsubseteq Wheel \wedge X.WheelType = \text{Racing bike} \Rightarrow \\
 & \Rightarrow X.SpokeNumber \leq 24,
 \end{aligned}$$

$$\begin{aligned}
 & X \sqsubseteq \text{Frame} \wedge X.\text{RearWheel} = Y \wedge Y \sqsubseteq \text{Wheel} \wedge \\
 & \quad \wedge X.\text{FrameType} = \text{Racing bike} \Rightarrow Y.\text{SpokeNumber} > 20 \\
 & X \sqsubseteq \text{Frame} \wedge (X.\text{RearBag}_1 = Y \vee X.\text{RearBag}_2 = Y) \wedge \\
 & \quad \wedge Y \sqsubseteq \text{Bag} \wedge Y.\text{Junction} = Z \wedge Z \sqsubseteq \text{Junction} \wedge \\
 & \quad \wedge X.\text{FrameMaterial} = M \wedge \\
 & \quad \wedge J.\text{JunctionMaterial} = M' \Rightarrow M = M'\}, \\
 \Gamma_{\text{Edges}} = & \{X \sqsubseteq \text{Frame} \wedge X.\text{FrameType} = \text{Racing bike} \wedge \\
 & \quad \wedge X.\text{RearGears} = Y \Rightarrow Y \sqsubseteq \text{Gears}, \\
 & X \sqsubseteq \text{Frame} \wedge X.\text{FrameType} = \text{Racing bike} \wedge \\
 & \quad \wedge X.\text{RearBag}_1 = B \wedge X.\text{RearBag}_2 = C \Rightarrow \\
 & \quad \Rightarrow B \sqsubseteq \text{Null} \wedge C \sqsubseteq \text{Null} \wedge B \neq C\}.
 \end{aligned}$$

Meta-paths occurring in meta-variables of node and edge constraint are easily encoded as generic constraints by using ports. For example, the meta-path $[\text{rear bag}, \text{junction}]$ can be encoded as

$$\begin{aligned}
 & X \sqsubseteq \text{Frame} \wedge (X.\text{RearBag}_1 = Y \vee X.\text{RearBag}_2 = Y) \wedge \\
 & \quad \wedge Y \sqsubseteq \text{Bag} \wedge Y.\text{Junction} = Z \wedge Z \sqsubseteq \text{Junction}.
 \end{aligned}$$

As regards the wildcards ' $_$ ' and ' \star ' occurring in meta-paths, we act as follows. Since each ' $_$ ' matches any arbitrary edge between two nodes. This can be encoded as a disjunction of expression on all the ports corresponding to edges between these two nodes. Let us consider, for instance, the following constraint for the node *Wheel*:

$$\langle \text{FrameType}, \text{Frame}, [_] \rangle = \text{WheelType}.$$

It can be encoded by this disjunction of expression on ports:

$$\begin{aligned}
 & X \sqsubseteq \text{Frame} \wedge (X.\text{FrontWheel} = Y \vee X.\text{RearWheel} = Y) \wedge Y \sqsubseteq \text{Wheel} \wedge \\
 & \quad \wedge X.\text{FrameType} = T \wedge Y.\text{WheelType} = T' \Rightarrow T = T'.
 \end{aligned}$$

On the other hand, each ' \star ' stands for any arbitrary path between two nodes. It can be encoded in a generic constraint as a disjunction of conjunction of expression on ports. Each conjunction represents a possible path between the two nodes. Notice that this encoding can be exploited only if there are no cycles between the two nodes (because, clearly, a cycle corresponds to an infinite number of possible paths). Hence, in the presence of cycles, we have to postpone the encoding of ' \star ' at the instance level, when generic constraints are instantiated on the concrete variables.

Γ_{Model} contains encodings of model constraints. For the bicycle example we have:

$$\begin{aligned} \Gamma_{Model} = \{ & X \sqsubseteq \text{Frame} \wedge X.\text{RearGears} = Y \wedge Y \sqsubseteq \text{Gears} \wedge \\ & \wedge X.\text{RearWheel} = Z \wedge Z \sqsubseteq \text{Wheel} \wedge \\ & \wedge Y.\text{GearType} = \text{Special} \Rightarrow W.\text{SpokeNumber} = 26 \}. \end{aligned}$$

Recall that meta-paths in model constraint meta-variables have a different interpretation than the ones appearing in node and edges constraints. Given a meta-variable $\langle \text{VarName}, \text{NodeName}, \text{MetaPath} \rangle$, we have to consider all the possible path matching MetaPath and connecting a node with NodeName in the product model graph. MetaPath is encoded as a disjunction of conjunction of expressions on ports, where each conjunction represents one of the possible paths reaching NodeName and matching MetaPath . Considerations made earlier for ‘_’ and ‘★’ also apply in this case.

Temporal constraints between activities, as well as other process related constraints such as resource constraints, duration constraints, etc., are encoded as generic constraints belonging to $\Gamma_{Process}$. For our example, the following generic constraints for temporal constraints between (atomic) activities are members of the set $\Gamma_{Process}$.

$$\begin{aligned} Y \sqsubseteq \text{Gears} &\Rightarrow Z \sqsubseteq \text{GearsConstruction}, \\ Z \sqsubseteq \text{GearsConstruction} &\Rightarrow Y \sqsubseteq \text{Gears}, \\ X \sqsubseteq \text{GearsConstruction} \wedge Y \sqsubseteq \text{WheelsConstruction} &\Rightarrow \\ X.\text{Start} = Y.\text{Start} \wedge X.\text{End} = Y.\text{End} \wedge X.\text{Start} &\leq X.\text{End}, \\ X \sqsubseteq \text{FrameConstruction} \wedge Y \sqsubseteq \text{PartialAssembly} &\Rightarrow \\ X.\text{Start} \leq X.\text{End} \wedge X.\text{End} < Y.\text{Start} \wedge Y.\text{Start} &\leq Y.\text{End}. \end{aligned}$$

The first two constraints encode the *is_absent* constraint on activity “Gear construction”. The third one encodes the *equals* constraint between “Gear construction” and “Wheels construction”. The last one encodes the *before* constraint between “Frame construction” and “Partial assembly”.

Composite and multiple instance activities can also be encoded in a GCSP. A given composite activity is simply unfolded in the model before producing the encoding. To deal with multiple instance activities we add to the set \mathcal{A} a name InstActName for each multiple instance activity ActName . By means of expressions like $X \sqsubseteq \text{Process} \wedge X.\text{InstActName} = n$ generic constraints are introduced to control the number of instance of ActName . Depending on the number of instances we may have:

$$\begin{aligned} X \sqsubseteq \text{Process} \wedge X.\text{InstActName} = 1 &\Rightarrow Y \sqsubseteq \text{ActName}, \text{ or} \\ X \sqsubseteq \text{Process} \wedge X.\text{InstActName} = 2 &\Rightarrow \\ \Rightarrow Y \sqsubseteq \text{ActName} \wedge Z \sqsubseteq \text{ActName} \wedge Y \neq Z, \text{ or} \\ \dots \end{aligned}$$

A better encoding can be achieved by extending the framework of GCSP with the notion of *global generic constraint*. In particular, we consider a global constraint of the form **exactly_k**(T, k), where $T \in \mathcal{T}$, and $k \in \mathbb{N}$. Such a constraint is satisfied if and only if $|\{X \mid X \in \mathcal{X} \wedge X \sqsubseteq T\}| = k$ holds, i.e., if there are exactly k component instances of type T . Constraints similar to the global generic constraint **exactly_k** appeared, for instance, in [SH93]. A generic constraint **exactly_k**(T, k) may be instantiated on active variables in \mathcal{C} using reified constraints and the global constraint **sum** as follows (where C_i and B_i for $i \in \{1, \dots, n\}$ are the active variables in \mathcal{C} and auxiliary variables introduced by the instantiation mechanism, respectively):

$$(B_1 \Leftrightarrow C_1 \sqsubseteq T) \wedge (B_2 \Leftrightarrow C_2 \sqsubseteq T) \wedge \dots \wedge (B_n \Leftrightarrow C_n \sqsubseteq T) \wedge \\ \wedge \text{sum}([B_1, \dots, B_n], =, k).$$

We may deal with a composite multiple instance activity C that can have at most n instances, adding to the GCSP the encoding of the model associated to C (i.e., types, names and generic constraints). By means of the global constraints **exactly_k** we can control the number of instances of C . By defining constraints of the form

$$X \sqsubseteq C \wedge C.a = Y \wedge C.b = Z \Rightarrow Y \sqsubseteq A \wedge Z \sqsubseteq B,$$

where A and B are activities belonging to the model associated to C , and a and b are ports modeling this relations, we can control the instantiation of activities that belongs to the model associated to C .

Notice that the generic global constraint **exactly_k** can be also used to control the number of instances of a node, thus avoiding the need to introduce a generic constraint for each possible value of a cardinality variable.

As mentioned in Chapter 5, PRODPROC allows one to model resource production/consumption, caused by process activities, by means of resource constraints of the form $\langle A, R, q, TE \rangle$. To encode a constraint of this form we add to \mathcal{T} the element R and to \mathcal{A} the attribute names *Min*, *Max*, and *iv*. This to represent, respectively, the minimum, the maximum, and the initial capacity value for the resource. Moreover, we add to \mathcal{A} an attribute name for each quantity variable q appearing in a resource constraint, as well as the attribute name *Before_start_A* to represent the quantity of a resource available before the execution of an activity that produces/consumes it. As an example, let us consider the resource constraint $\langle A, R, q, FromStartToEnd \rangle$, this generic constraints has to hold:

$$X \sqsubseteq R \wedge Y \sqsubseteq A \Rightarrow X.Before_start_A + Y.q \geq X.Min \wedge \\ \wedge X.Before_start_A + Y.q \leq X.Max \wedge \\ \wedge X.Before_start_A - Y.q \geq X.Min \wedge \\ \wedge X.Before_start_A - Y.q \leq X.Max.$$

The value of a variable *Before_start_A* depends on which activities that produce/consume R start their executions before A begins and on their time extents. For example,

let B an activity having associated the constraint $\langle B, R, -5, AfterStart \rangle$ and suppose the temporal constraint B before A has to hold. To compute the value of $Before_start_A$ (and $Before_start_B$) we exploit this generic constraint:

$$X \sqsubseteq R \Rightarrow X.Before_start_A = X.iv - 5 \wedge X.Before_start_B = Z.iv.$$

On the other hand, if there are no temporal constraint imposed between A and B , we adopt the following generic constraints.

$$X \sqsubseteq A \wedge Y \sqsubseteq B \wedge Z \sqsubseteq R \wedge X.Start > Y.Start \Rightarrow$$

$$Z.Before_start_A = Z.iv - 5 \wedge Z.Before_start_B = Z.iv,$$

$$X \sqsubseteq A \wedge Y \sqsubseteq B \wedge Z \sqsubseteq R \wedge X.Start \leq Y.Start \Rightarrow$$

$$Z.Before_start_A = Z.iv \wedge Z.Before_start_B = Z.iv.$$

Further constraints are needed to impose that the amount of R remains between the defined minimum and maximum capacity when the executions of A and B overlaps in some way. For instance, in the case of a temporal constraint A equals B , we have:

$$X \sqsubseteq A \wedge Y \sqsubseteq B \wedge Z \sqsubseteq R \wedge X.Start = Y.Start \wedge X.End = Y.End \Rightarrow$$

$$\Rightarrow Z.Before_start_A + X.q - 5 \geq Z.Min \wedge$$

$$\wedge Z.Before_start_A + X.q - 5 \leq Z.Max \wedge$$

$$\wedge Z.Before_start_A - X.q - 5 \geq Z.Min \wedge$$

$$\wedge Z.Before_start_A - X.q - 5 \leq Z.Max.$$

In general, when two different activities A and B produce/consume the same resource and are not ruled by any temporal relations, we will have (at most) one generic constraint for each of the atomic temporal relations mentioned in Chapter 5 (different from *before* and *after*) that may hold between A and B . Notice that, different temporal relations might give rise to the same generic constraint, in presence of certain time extents.

Clearly, if the activities producing/consuming a resource are more than two, and there are not temporal constraints imposed between them, then it is necessary to take into account all the possible temporal ordering of their executions. To deal with if- and iff-conditional resource constraint is sufficient to add the encoding of the condition to the left side of the generic constraints (iff-conditional constraint require also additional generic constraints in which the condition appears on the right side).

An alternative viable possibility, that would generate more succinct encodings of resource constraints, consists in considering, at the GCSP level, a generic **cumulatives** constraint, i.e., a constraint of the form

$$\text{generic_cumulatives}(Tasks, Machines),$$

where: *Tasks* is a list of predicates

$$\mathbf{task}(A, R, X.Start, X.Duration, X.End, X.q, Y, TE),$$

one for each resource constraint $\langle A, R, q, TE \rangle$, while *Machines* contains a predicate

$$\mathbf{machine}(R, X.Min, X.Max, X.iv),$$

for each resource R involved in a resource constraint. The instantiation of such a constraint on the active variables in \mathcal{C} and \mathcal{P} leads to a constraint $\mathbf{cumulatives}(Tasks^I, Machines^I)$, where $Tasks^I$ and $Machines^I$ are lists containing instantiation of **task** and **machine** predicates, respectively. A predicate $\mathbf{task}(A, R, X.Start, X.Duration, X.End, X.q, Y, TE)$ is instantiated to a predicate $\mathbf{task}(C_i.Start, C_i.Duration, C_i.End, C_i.q, C_j, TE)$ for each active variables $C_i \in \mathcal{C}$ such that $C_i \sqsubseteq A$ holds ($C_j \in \mathcal{C}$ is an active variable such that $C_j \sqsubseteq R$ holds). A predicate $\mathbf{machine}(R, X.Min, X.Max, X.iv)$ is instantiated to a predicate $\mathbf{machine}(C_j.Min, C_j.Max, C_j.iv)$ for each active variable $C_j \in \mathcal{C}$ such that $C_j \sqsubseteq R$. To deal with if- and iff- conditional resource constraint is sufficient to extend the generic **cumulatives** constraint in order to take them into account.

The encoding of duration constraints and constraints involving product and process elements is straightforward. For example, the following duration constraint for the activity “Frame construction” of the bicycle production process (where *FrameMultiplier* is a process variable, and q_W a resource quantity variable)

$$d = \frac{2 \cdot \text{FrameMultiplier}}{|q_W|},$$

can be encoded as the generic constraint

$$\begin{aligned} X \sqsubseteq \text{FrameConstruction} \wedge Y \sqsubseteq \text{Process} &\Rightarrow \\ \Rightarrow X.Duration &= \frac{2 \cdot Y.FrameMultiplier}{|X.q_W|}. \end{aligned}$$

Similarly, a constraint involving process variables and product variables like the following one

$$\langle \text{FrameType}, \text{Frame}, [] \rangle = \text{Racing bike} \Rightarrow \text{FrameMultiplier} = 4,$$

can be encoded as the generic constraint

$$\begin{aligned} X \sqsubseteq \text{Frame} \wedge Y \sqsubseteq \text{Process} \wedge X.FrameType &= \text{Racing bike} \Rightarrow \\ \Rightarrow Y.FrameMultiplier &= 4. \end{aligned}$$

Concerning the global constraints, such as **alldifferent**, that may be used as node model constraints, we can express them into a GCSP by introducing the notion of generic **alldifferent** constraint. As the intuition suggests, this is a constraint of the form

$$\mathbf{generic_alldifferent}([\langle T_1, A_1 \rangle, \dots, \langle T_n, a_n \rangle]),$$

where $\forall i \in \{1, \dots, n\} T_i \in \mathcal{T}, a_i \in \mathcal{A}$. The instantiation of such a constraint on active variables in \mathcal{C} and \mathcal{P} leads to a global constraint **alldifferent**(L), where $L = \circ_{i=1}^n [C.a_i \mid C \in \mathcal{C} \wedge C \sqsubseteq T_i]$ (here $\circ_{i=1}^n L_i$ denotes the concatenation of the lists L_1, \dots, L_n).

Clearly, if a **generic_alldifferent** constraint appears in a generic constraint together with other constraint on placeholder variables of type in the set $\{T_1, \dots, T_n\}$, then each C in L must satisfy those constraints too. Global constraints similar to **generic_alldifferent** can be also defined to encode *aggConstraints*.

To conclude our working example, observe that given a GCSP \mathcal{M} encoding the bicycle model as explained above, a bicycle configuration problem can be represented as the tuple $\langle V^I, \Gamma^I, \mathcal{M} \rangle$, where V^I contains the variables related to product component that are mandatory (e.g., the frame) and of interest (e.g., a bag) as well as those related to activities that must be executed for sure (e.g., “Frame construction”).

8.3 Custom Product Manufacturing and Generative CSPs

In this chapter we illustrated how all the features offered by PRODPROC to specify product and production process configuration problem can be encoded into a Generative Constraint Satisfaction Problem.

The mapping we defined leads to two different considerations on the role of Generative CSPs in the field of product and process configuration. First, it suggests that Generative CSPs can be exploited to give a constraint-based semantics to the PRODPROC framework. Second, the mapping shows that Generative CSPs are well-suited for modeling not only product configuration or service composition problems, but also mixed product and process configuration problems. Consequently, the Generative CSP framework can be the core of a full-fledged configuration system covering the whole customizable product cycle.

Chapter 9

Language Comparisons

In this chapter, we compare the PRODPROC framework with some of the most important existing product and process modeling tools, to put in evidence its strength and limitations. We start comparing PROMo with existing product modeling languages in Section 9.1. Then, in Section 9.2, we compare MART with existing process modeling languages. Finally, in Section 9.3, we point out the innovativeness of PRODPROC. Section 9.4 concludes the chapter with a brief summary of the comparison of PRODPROC with selected approaches to product and process modeling.

9.1 PROMo Language

Existing product configuration tools can be partitioned in three main classes, namely, ASP-based systems, BDD-based systems, and CSP-based systems. In this section, we compare the PROMo language with the modeling languages of tools belonging to these classes. Moreover, we discuss the commonalities and differences between PROMo, the configuration ontology introduced by Soininen et al. in [STMS98], and the UML/OCL based modeling approach presented by Fefernig in [Fel07].

9.1.1 ASP-based Systems

ASP-based systems provide a number of features that are specifically tailored to the modeling of software product families. On the one hand, this makes these systems appealing for a relevant set of application domains. On the other hand, it results in a lack of generality, which is probably the major drawback of this class of systems. In particular, they support neither DNF constraints nor global constraints, and they encounter some problems in the management of arithmetic constraints related to the grounding stage. Let us consider the case of Kumbang Configurator [MAMS05], which is probably the most significant system in this class. Kumbang Configurator provides a domain ontology to model variability in software product families, that unifies feature-based and architecture-based approaches. In addition, it allows the user to model the interrelations between these two views. A precise characterization of the main features of Kumbang Configurator has been given in

UML [AMS07]. As for its limitations, Kumbang Configurator lacks various natural constructs for product modeling. As an example, in Kumbang Configurator a domain of the form $D(V) = [\min, \max]$ can be defined only by explicitly enumerating its elements.

9.1.2 BDD-based Systems

BDD-based systems, like Configit Product Modeler, CLab, CLab#, and iCoDE, trade the complexity of the construction of the BDD, that basically provides an encoding of all possible configurations, for the simplicity and efficiency of the configuration process.

Despite their various appealing features, BDD-based systems suffer from some significant limitations. First, even though some work has been done on the introduction of modules, they basically support flat models only. Moreover, they find it difficult to cope with global constraints. As an example, `alldifferent` constraints lead to exponential space complexity for BDD construction. As a matter of fact, some attempts at combining BDD with CSP to tackle `alldifferent` constraints have been recently done [NBJT09]; however, they are confined to the case of flat models. We are not aware of any BDD system that deals with global constraints in a sufficiently general and satisfactory way.

The most significant BDD-based system is Configit Product Modeler [Con09]. Compared to such a system, PROMO makes it possible to define a larger set of compatibility relations between product component characteristics. As an example, PROMO allows one to compare a characteristic with an expressions, while Configit Product Modeler supports comparisons with constants only. For instance, given two product component characteristics X and Y , PROMO allows one to define the constraint $X = 2 \cdot Y$. This is not possible with Configit Product Modeler (this limitation does not affect CLab [Jen04], CLab#, and iCoDE). Another limitation of Configit Product Modeler concerns the management of multiple instances of a product component. The current version of the tool does not allow one to define product components whose number of instances in a given configuration can vary depending on some constraints. In previous versions of Configit Product Modeler this was possible by means of the so-called modules. However, such a capability had a severe limitation: it was not possible to define a constraint involving characteristics associated with different modules in the product model. As a consequence, the consistency of a configuration with respect to such a constraint could only be ensured by the runtime application that was using the BDD to support the user in the search for a valid configuration. On the contrary, constraints over arbitrary sets of characteristics can be easily implemented in PROMO, thus fully preserving the declarative nature of the modeling process.

In [vdMA04] and [vdMWA06] the authors propose a modeling language for a BDD-based system, that allows one to model products built of an arbitrary number of components. Product components are modeled through the so-called modules. A module is an entity with variables, domains, and constraints of its own. It can be instantiated by another module importing it, with which it then shares the variables it exports, and it can instantiate other modules importing them, with which it then shares the variables they export. In this way, the module instances form a tree at run-time. In programming language terms, a module can to some extent be thought of as a class, from which objects are instantiated. Also, the usual hierarchy building is possible. A car can be a module,

with submodules like door, roof, and engine, and each of those can in turn have submodules. Hence, this import/instantiate mechanism allows one to define *has-part/is-part-of* relations, and leads to the definition of a dependency graph (that may contain cycles) for the modules. The main difference between this product modeling language and PROMo is represented by *has-part/is-part-of* relations with non-fixed cardinality. The modeling language described in [vdMA04, vdMWA06] allows one to define only *has-part/is-part-of* relations having a fixed cardinality, while in PROMo it is possible to define a *has-part/is-part-of* relation having a variable with a finite domain representing its cardinality, and to define constraints involving cardinality variables too. Moreover, it does not support global constraints.

9.1.3 CSP-based Systems

Unlike ASP-based and BDD-based systems, CSP-based ones allow the user to define non-flat models and to deal with global constraints. Unfortunately, the modeling expressiveness of CSP-based systems has a cost, i.e., backtrack-free configuration algorithms for CSP-based systems are often inefficient, while non backtrack-free ones need to explicitly deal with dead ends. Moreover, most CSP-based systems do not offer high-level modeling languages (product models must be specified at the CSP level). Some well-known CSP-based configuration systems, such as ILOG Configurator [Jun03], which features various interesting modeling facilities, and Lava [FFH⁺98], which is based on Generative-CSP, seem to be no longer supported.

A recent CSP-based product configuration system is Morphos Configuration Engine (MCE) [CRD⁺10]. As all CSP-based systems, it makes it possible to define non-flat models. Its modeling support turns out to be simple enough and easily extensible (addition of global constraints, definition of specific optimization strategies, etc.). Its configuration algorithm is not backtrack-free, but it exploits back-jumping capabilities, to cope with dead ends, and branch-and-prune capabilities, to improve domain reduction. PROMo can be viewed as an extension of MCE modeling language. In particular, it extends MCE modeling language with the following features: (1) *cardinality variables*, i.e., *is-part-of* relations can have non-fixed cardinalities; (2) *product model graph*, i.e., nodes and relations can define a graph, not only a tree; (3) *cardinality constraints* and *cardinality model constraints*, i.e., constraints can involve cardinalities of relations; (4) *meta-paths*, i.e., a mechanism to refer to particular node instance variables in constraints.

9.1.4 Ontologies and UML-based Approaches

The definition of a common representation language to support knowledge interchange between and integration of knowledge based configuration systems is an important issue. In [STMS98] one approach to collect relevant concepts for modeling configuration knowledge bases is presented. The defined ontology is based on Ontolingua [Gru92] and represents a synthesis of resource-based, function-based, connection-based and structure-based configuration approaches. The goal of [STMS98] was to present an ontology including

major modeling concepts needed for the design of configuration models. The PROMO approach cover only a subset of these concepts, i.e., PROMO currently does not allow one to define taxonomies, functions and contexts. However, PROMO defines a rich constraint language, while it remains unclear to what extent the language used in [STMS98] supports the formulation of configuration-domain specific constraints.

Another approach that aims at obtaining standardized configuration knowledge representations is discussed in details in [Fel07]. The use of OCL [WK03] and the UML [RJB98] as standard representation languages for building platform independent and platform specific configuration model is demonstrated in [Fel07]. These models are specified conforming to the model development process defined by the model-driven architecture (MDA) [Fra03] which is an industrial standard framework for model development and interchange. Such a standardized representation of configuration knowledge is a foundation for the effective integration of configuration technologies into software environment dealing with the management of complex products and services. PRODPROC can be viewed as the source code representation of a configuration system with respect to the MDA abstraction levels presented in [Fel07]. PRODPROC product modeling elements can be mapped to UML/OCL in order to obtain platform specific (PSM) and platform independent (PIM) models. The mapping to OCL of meta-paths containing ‘ \star ’ wildcards and of model constraints requires some attention. For example, the latter do not have an explicit context as OCL constraint must have. Since PRODPROC (currently) does not support the definition of taxonomies of product components, there will not be generalization hierarchies in PMSs and PIMs corresponding to PRODPROC models.

9.1.5 Feature Diagrams

Feature Diagrams (FDs) [KCH⁺90] are a widely used technique in software product line configuration.

A *feature* is a system property that is relevant to some stakeholder and is used to capture some commonalities or discriminate among systems in a family. A feature may denote any functional or non-functional characteristics at the requirements, architectural, component, platform or any other level. Feature are organized in *feature diagrams*. A feature diagram is a tree of features with the root representing a concept (e.g., a software system). *Feature models* are feature diagrams plus additional information such as feature descriptions, binding times, priorities, stakeholders, etc.

There are three kinds of feature, *root feature*, *grouped feature*, and *solitary feature*. A grouped feature is a feature occurring in a *feature group*. A solitary feature is a feature which is, by definition, not grouped in a feature group. Any feature can be given maximum one attribute by associating it with a type (e.g., integer). A feature needs only at most one attribute since a collection of attributes can be easily modeled as a number of sub-features, where each is associated with a desired type. A feature can be mandatory or optional, features belonging to a feature group may be in an *and* (all sub-features must be selected), *alternative* (only one sub-feature can be selected) or *or* (one or more sub-features can be selected) relation.

A number of extensions for FDs have been proposed in the past years. Examples

are: feature diagrams based on directed acyclic graphs [KSJ⁺98]; cardinality-based feature models (i.e., diagrams with UML-like multiplicities of the form $[n, m]$ with n being the lower bound and m the upper bound, used to limit the number of sub-features that can be part of a product whenever the parent is selected) [CHE05]; feature models with extra functional features (i.e., constraints between attributes) [BTRC05]. Staged configuration [CHE04] is a process that allows the incremental configuration of feature models. It can be achieved by performing a step-wise specialization of the feature model.

Despite the existing extensions, feature diagrams lacks different features present in PROMo, e.g, the hyper-graph structure of the model, and cardinality (model) constraints. The structure of feature models is particularly well suited for representing requirements-level variability, while the more rich structure of the ProMo approach and AI configuration approaches may be more appropriate at the level of component configuration.

9.1.6 SysML Structural Constructs

SysML [Obj10] is a general-purpose modeling language for systems engineering applications. It can be viewed as a customized variant of UML. It is intended to unify the diverse modeling languages currently used by systems engineers. SysML supports the specification, analysis, design, verification, and validation of a broad range of complex systems. These may include hardware, software, information, processes, personnel, and facilities.

SysML modeling constructs can be partitioned into two classes: static structural constructs, and dynamic behavioral constructs. Structural constructs are used in SysML structure diagrams, including the package diagram, block definition diagram, internal block diagram, and parametric design. We focus here on the most relevant structural constructs for a comparison with PROMo, namely, *blocks* and *constraints*.

Blocks are modular units of system description. Each block defines a collection of features to describe a system or other element of interest. These may include both structural and behavioral features, such as properties and operations, to represent the state of the system and behavior the system may exhibit. Blocks provide a general-purpose capability to model systems as trees of modular components. SysML blocks can be used throughout all the phases of system specification and design, and can be applied to many different kind of systems.

The *block definition diagram* in SysML defines features of blocks and relationships between blocks such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties and operations, and relationships such as system hierarchy or a system classification tree. The internal block diagram in SysML captures the internal structure of a block in terms of *properties* and *connectors* between properties. A block can include properties to specify its values, parts and references to other blocks. A property can represent a role or usage in the context of its enclosing block. A property has a type that supplies its definition. A part belonging to a block, for example, may be typed by another block. The part defines a local usage of its defining block within the specific context to which the part belongs. For example, a block that represents the definition of the wheel can be used in different ways. The front wheel and the rear wheel can represent different usages of the same wheel definition. SysML also allows each usage

to define context-specific values and contains associated with the individual usage.

Constraint blocks provide a mechanism for integrating engineering analysis such as performance and reliability models with other SysML models. Constraints blocks can be used to specify a network of constraints that represent mathematical expressions which constraint the physical properties of a system. A constraint block includes the constraint and the parameters of the constraint. Constraint blocks define generic form of constraints that can be used in multiple context. Such constraints can be arbitrary complex mathematical or logical expressions.

From the point of view of product modeling, SysML structure constructs, in particular blocks and constraints, may be used for defining models of configurable products. That is, SysML may be used as the modeling language for a product configurator. However, it does not allow one to model a product as a graph of components.

9.2 MART Language

In this section we compare the MART language with the modeling language of some of the most important tools for process modeling. We begin with a brief analysis of the temporal constraint language, and then proceed examining the features of the MART language.

9.2.1 Temporal Constraints

The temporal constraint language we presented in Chapter 5 combines the binary relations on intervals introduced in [All83], and atomic temporal constraints inspired by some of the constraint templates of DECLARE [PSvdA07]. It allows one to define the typical precedence relation of (imperative) process modeling languages like YAWL and BPMN with the *before* relation. The atomic temporal constraints from 2 to 13 (see Section 5.3.1) allow to precisely define the temporal relation between two activities and, to the best of our knowledge, are not available in current process modeling tools. Atomic formulas from 14 to 17 (see Section 5.3.1) have been inspired by constraint templates of the language ConDec having similar meanings, they make our temporal constraint language a bit more declarative, allowing one to easily define complex relations among activities. The atomic temporal constraints and the possibility to combine and condition them, lead to an expressive language for stating temporal constraints, that combines temporal relation modeling elements of imperative and declarative process modeling languages.

9.2.2 Modeling Processes with MART

The MART language combines features for modeling different aspects of a process, from activities and temporal relations between them, to resource production and consumption. Some of these features are present in existing imperative and/or declarative process modeling languages too, while others are a novelty.

With respect to the works of Mayer et al. on service process composition (e.g. [MTS09]), MART is more tuned towards production process modeling and configuration. However, certain aspects of service composition problems can be modeled using MART and PROMO.

In the past years different formalism have been proposed for process modeling: Petri nets and their extensions (e.g., condition event nets, colored Petri nets, etc.); event driven process chains; workflow nets; the Business Process Modeling Notation (BPMN) [WM08]; Yet Another Workflow Language (YAWL) [tHvdAAR10]; DECLARE [PSvdA07]. In the following we will compare the modeling capabilities of MART with some of the most used and newest formalisms, i.e., BPMN, YAWL and DECLARE.

BPMN

The Business Process Modeling Notation (BPMN) [WM08] has been developed under the coordination of the Object Management Group. The intent of the Business Process Modeling Notation in business process modeling is very similar to the intent of the Unified Modeling Language for object-oriented design and analysis. BPMN aims at identifying the best practices of existing approaches to process modeling, and to combine them into a new generally accepted language. The set of ancestors of BPMN include not only graph-based and Petri-net- based process modeling languages, but also UML activity diagrams and event-driven process chains. Business process models are expressed in business process diagrams, the notational elements in business process diagrams (see Figure 9.1) are divided into four basic categories, each of which consists of a set of elements. *Flow objects* are the building blocks of business processes; they include events, activities, and gateways. The occurrence of states in the real world that are relevant for business processes are represented by events, they are the glue between situations in business organizations and processes that will be enacted if these situations occur. Events in a business process can be partitioned into three types, based on their position in the business process: start events are used to trigger processes, intermediate events can delay processes, or they can occur during processes. End events signal the termination of processes. Activities represent work performed during business processes. BPMN supports a hierarchical nesting of activities, so that each activity is either an atomic activity or a sub-process; the nesting of subprocesses can have arbitrary depth. Gateways are used to represent the split and join behavior of the flow of control between activities, events, and gateways. Each gateway acts as either a join node or a split node: join nodes have at least two incoming arcs and exactly one outgoing edge, split nodes have exactly one incoming arc and at least two outgoing edges. There are notational symbols for different gateways, e.g., exclusive or, inclusive or, and, etc. BPMN supports different types of exclusive or splits, depending on data and events. For each outgoing edge of a data-based exclusive or split there is a gate with an associated condition, also known as gate condition, based on data. The gate conditions associated with a gateway are evaluated in a specific order. Once a gate condition evaluates to true, the corresponding branch is taken, and the other conditions are disregarded. There is also an option for a default flow, which is followed in case all other gate conditions evaluate to false. It is the responsibility of the modeler to make sure that at least one condition of a data-based exclusive or split evaluates to true or that a default flow is defined. An event-based exclusive or gateway enables multiple activities of type receive. If the first of these activities receives a message, the other activities are neglected. Organizational aspects are represented in business process diagrams by *swimlanes*. Swimlanes are restricted to a

two-level hierarchy: pools and lanes. Pools represent organizations that participate in the interaction of multiple business processes, each of which is enacted by one organization. Lanes represent organizational entities such as departments within a participating organization. *Artefacts* are used to show additional information about a business process that is “not directly relevant for sequence flow or message flow of the process”, as the standard mentions. Data objects, groups, and annotations are supported as artefacts. Each artefact can be associated with flow elements. Artefacts serve only information purposes, the process flow is not influenced by them. *Connecting objects* connect flow objects, swimlanes, or artefacts. Sequence flow is used to specify the ordering of flow objects, while message flow describes the flow of messages between business partners represented by pools. Association is a specific type of connecting object that is used to link artefacts to elements in business process diagrams.

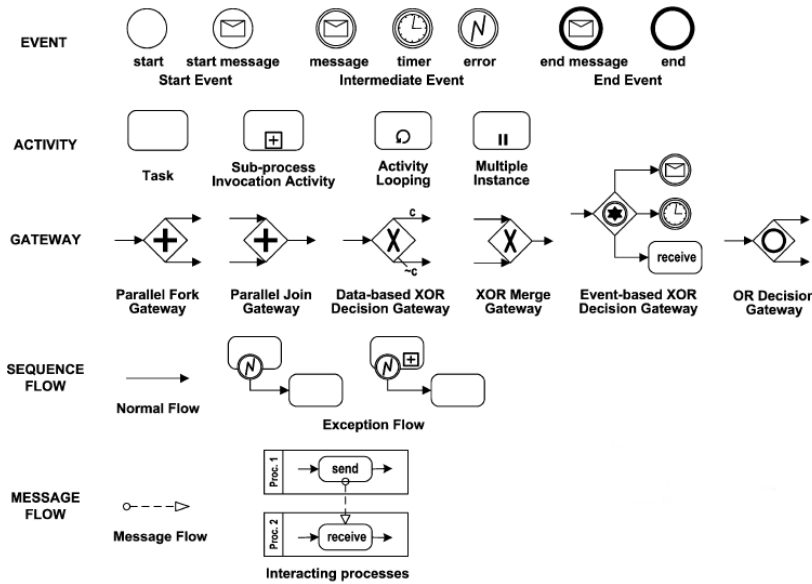


Figure 9.1: BPMN elements.

MART has in common with BPMN the notion of atomic activity, sub-process, and multiple instance activity. The effects of BPMN joins and splits on the process flow can be obtained using temporal constraints. In MART there are not the notions of BPMN events, exception flows, and message flows. However, events can be modeled as instantaneous activities, and data flowing between activities can be modeled with model variables.

YAWL

Yet Another Workflow Language (YAWL) [tHvdAAR10] is a process modeling language whose development has been motivated by the lack of a process language that directly supported all control flow patterns. YAWL is based on workflow nets, in particular, it uses extended workflow nets as building blocks for workflow specifications. Extended

workflow nets enhance traditional workflow nets with notational convenience (direct arcs between transitions are allowed), explicit split and join behavior that can be attached to transitions, nonlocal behavior, and the handling of multiple instances tasks. A YAWL *model* is composed of a set of YAWL *nets* in the form of a rooted graph structure. Each YAWL net is composed of a series of *tasks* and *conditions*. Tasks and conditions in YAWL nets play a similar role to transitions and places in Petri nets. Atomic tasks have a corresponding implementation that underpins them. *Composite tasks* refer to a unique YAWL net at a lower level in the hierarchy that describes the way in which the composite task is implemented. One YAWL net, referred to as the top level process or top level net, does not have a composite task referring to it and it forms the root of the graph. Each YAWL net has one unique input and output condition. The input and output conditions of the top level net serve to signify the start and endpoint for a process instance. Similar to Petri nets, conditions and tasks are connected in the form of a directed graph; however, there is one distinction in that YAWL allows for tasks to be directly connected to each other. In this situation, it is assumed that an implicit condition exists between them. It is possible for tasks (both atomic and composite) to be specified as having *multiple instances*. Tasks in a YAWL net can have specific join and split behaviors associated with them. The supported join and split constructs are the AND-join, OR-join, XOR-join, AND-split, OR-split, and XOR-split. YAWL supports the notion of a *cancellation region*, which encompasses a group of conditions and tasks in a YAWL net. It is linked to a specific task in the same YAWL net. At runtime, when an instance of the task to which the cancellation region is connected completes executing, all of the tasks in the associated cancellation region that are currently executing for the same case are withdrawn. The *data perspective* of YAWL encompasses the definition of a range of data elements, each with a distinct scoping. These data elements are used for managing data with a YAWL process instance and are passed between process components using query-based parameters. YAWL is able to support conditions (known as link or flow conditions) on outgoing arcs from OR-splits and XOR-splits. These conditions take the form of XPath expressions, which evaluate to a Boolean result, indicating whether the thread of control can be passed to this link or not. Figure 9.2 shows the graphical representations of YAWL elements.

MART has in common with YAWL the notion of task, multiple instance task, and composite task. YAWL join and split constructs are not present in MART, but using temporal constraints it is possible to obtain similar results. The notion of cancellation region is not present in MART, but our language could be extended to implement such a feature.

DECLARE

Languages like BPMN and YAWL define a process model as a detailed specification of a step-by-step procedure that should be followed during the execution. This approach is an *imperative* approach because it strictly specifies how the process will be executed and yields *highly-structured processes*. With imperative models all possibilities have to be entered into the model by specifying its control-flow, this is not an easy task when considering *loosely-structured processes*. As opposed to traditional imperative approaches to process

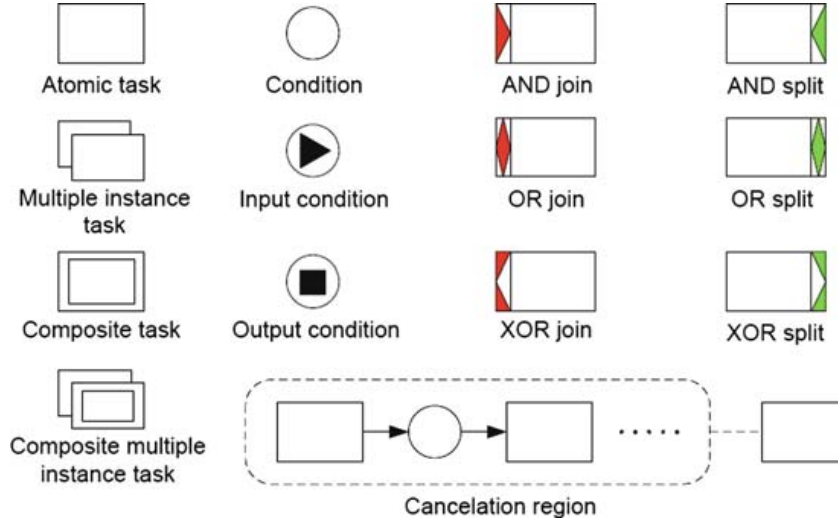


Figure 9.2: YAWL elements.

modeling, DECLARE [PSvdA07] uses a constraint-based declarative approach. DECLARE provides for multiple declarative languages (e.g., DecSerFlow, ConDec, etc.), is extendible and without any programming it can be configured to support additional constraint-based languages. Instead of explicitly defining the ordering of activities in models, DECLARE models rely on constraints to implicitly determine the possible ordering of activities (any order that does not violate constraints is allowed). Unlike most of the approaches, which offer a predefined set of constructs for defining dependencies between activities in process models (e.g., sequence, choice, parallelism, loops, etc.), DECLARE uses a customizable set of arbitrary constructs called constraint templates. Each template has (i) a unique name, (ii) a graphical representation, and (iii) a formal specification of its semantics in terms of Linear Temporal Logic (LTL) (Figure 9.3 shows some constraint templates). The enabling and execution of activities is driven by constraints: everything that does not violate the constraints is enabled for execution and all constraints must be satisfied at the end of the instance execution.

With respect to DECLARE, MART has in common the notion of activity, and the use of temporal constraints to define the control flow of a process. The set of atomic temporal constraints is not as big as the set of template constraints available in DECLARE, however it is possible to easily combine them to define complex constraints. Moreover, in MART it is possible to define multiple instance and composite activities, notions that are not presents in DECLARE.

As reported in [PSvdA07], it is possible to mix the modeling capabilities of YAWL and DECLARE in order to model processes exploiting a mixed imperative/declarative approach. Even if some temporal aspects can be modeled in YAWL via the extension proposed in [CP09], and YAWL's extended attributes could be used to specify product

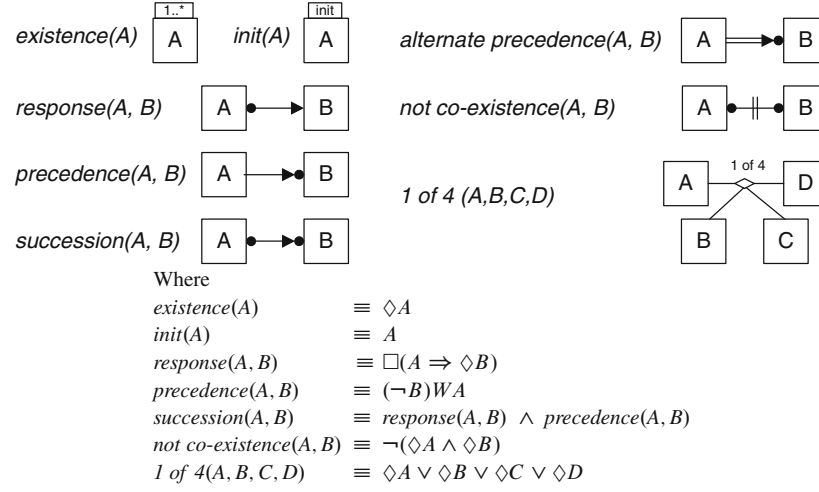


Figure 9.3: Examples of DECLARE constraint templates.

characteristics and resource constraints, the resulting framework still lacks features present in MART, such as temporal relations between activities.

Rather than extending YAWL and DECLARE, that are more geared toward the modeling of business processes, we decided to develop a new language, i.e., MART, covering the production process aspects we think are the most relevant during product configuration, namely, temporal relations between activities, duration constraints, and resource constraints, and more geared toward the coupling with PROMO models.

MART combines modeling features of imperative modeling languages like BPMN and YAWL (e.g., multiple instance activities, composite activities), with a declarative approach for control flow definition. Moreover, it presents features that, to the best of our knowledge, are not presents in other existing process modeling languages. These are: resource variables and resource constraints; activity duration constraints; product related constraints. Thanks to these features, MART is suitable for modeling production processes and, in particular, to model mixed scheduling and planning problems related to production processes. However, MART is not, and it has not been developed to be, a general business modeling language like BPMN or YAWL. Its main objective is to make it possible to (easily) model processes that are not highly structured, taking into account not only temporal relations between activities, but also activity durations and resources consumption and production. We are not aware of the existence of a process modeling language tool having all the necessary features to accomplish to this objective.

Process Configuration

A MART model does not simply represent a process ready to be executed like YAWL (or DECLARE) model does, it allows one to obtain a process instance ready to be executed through a configuration process. A user can configure a process assigning values to model variables and activity parameters (i.e., execution flags, quantity resource variables, instance

number variables). Moreover, we can allow a user to impose additional constraints, e.g. on activities duration and resource quantities.

Different works on process configuration have been proposed in the past years. Examples are: C-YAWL [LaR09], C-iECPs [LDtHM11], Provop [HBR10].

C-YAWL [LaR09] is an extension of YAWL that provides configuration opportunities for YAWL models. C-YAWL introduces the notion of input and output for YAWL tasks. By blocking such ports, the process flow through the port is inhibited and thus tasks subsequent to this port will not be executed. An input port of a task can also be hidden. This means, the execution of the task itself should be skipped, while the process flow continues afterwards. Moreover, the behavior of multiple instance tasks and tasks cancelation regions can be restricted through process configuration. MART allows one to configure processes selecting which activities will be executed and which not, and choosing values for model variables too.

C-iECP [LDtHM11] is a notation for configurable process modeling that extends the EPC notation with mechanism for representing a range of variations along multiple perspectives. C-iECPs address a major shortcoming of existing configurable process modeling notations, namely their lack of support for the resource (the notion of resources in C-iECPs is different from the one of MART, in particular they do not give rise to scheduling problems as in MART) and object perspectives. A C-iEPC model define a configurable process specifying variation points (e.g., configurable connectors, configurable functions, etc.), such a process can be configured via a questionnaire-driven approach. Specifically, dependencies between variation points are captured via domain constraints encoded in a questionnaire model. Analysts configure the process model by answering this questionnaire. In this way, it is guaranteed that the individualized process models are domain-compliant. An individualization algorithm for C-iEPC process models is provided in [LDtHM11], the syntactic correctness of the individualized iEPCs is guaranteed by this algorithm. However, the proposed approach does not prevent users from creating inconsistencies between object flow dependencies and control-flow dependencies that may lead to behavioral issues. As a result, the behavioral correctness of an iEPC is not guaranteed during configuration.

Provop [HBR10] is a framework for modeling and managing large collection of business process variants. Provop allows one to define a reference process model (i.e., a base process model) as well as the adjustments that may be applied to it in order to obtain specific process variants. Currently, Provop supports the following change patterns (i.e., parametrizable change operations): *insert* fragment, *delete* fragment, *move* fragment, and *modify* attribute. In Provop, an activity or connection node of the base process may be associated with adjustment points which correspond to its entry or exit, respectively. This, in turn, enables designers of process-specific adaptations to refer to selected process fragments. By the use of explicit adjustment points, it is possible to restrict the regions to which adaptations (e.g., insertion of a fragment) may be applied when configuring variants.

The concept of configuration in MART is different from the one of C-YAWL, C-iECPs, or Provop. We are interested in obtaining a process instance, a solution to the scheduling/planning problem described by a MART model, while the other process configuration systems we cited define variation points in process models, and aim at deriving different process model variants from a given model.

9.2.3 SysML Behavioral Constructs

As stated in Section 9.1.6, SysML modeling constructs can be partitioned into two classes: static structural constructs, and dynamic behavioral constructs. Behavioral constructs are used in SysML behavioral diagrams, including the activity diagram, sequence diagram, state machine diagram, and use case diagram. We focus here on activity modeling.

Activity modeling emphasizes the inputs, outputs, sequence, and conditions for coordinating other behaviors. It provides a flexible link to blocks owning those behaviors. An activity diagram allows to describe the flow of control and flow of inputs and outputs among actions. For example, it allows to define a model of driving and braking in a car that has an automatic braking system. In SysML activities can be both started and disabled. Creating an instance of an activity causes the activity to start executing, and vice versa. Destroying an instance of an activity terminates the corresponding execution, and vice versa. Activity as blocks can have associations between each other, including composition associations, and properties of any kind.

From the process modeling point of view, the behavioral constructs of SysML are geared toward modeling system behavior and not toward (configurable) production processes. Even if SysML has activities and constructs for control flow description, it lacks ad hoc constructs for resources and resource constraints. Moreover, duration constraints are not included in the current version of SysML.

9.3 PRODPROC Framework

The PRODPROC framework allows one to model products with the PROMO language, their production processes with the MART language, and to couple product models with process models by means of coupling constraints.

The only works on the coupling of product and process modeling and configuration we are aware of are the ones by Aldanondo et al. [AVDG08, AV08]. They propose to consider simultaneously product configuration and process planning problems as two constraint satisfaction problems; in order to propagate decision consequences between the two problems, they suggest to link the two constraint based models using coupling constraints, i.e., compatibility constraints linking variables of the configuration model with variables of the planning model.

The development of PRODPROC has been inspired by the papers of Aldanondo et al., in fact we introduced separated models for products and processes, and constraints for coupling them too. However, our modeling languages are far more complex and expressive than the one presented in [AVDG08, AV08]. For example, in [AVDG08] a product is modeled with two views: the functional view, i.e., variables representing configurable characteristics of a product, and the physical view, i.e., components constituting a product. The functional and physical views are linked with constraints that show how the components can fulfill the parameters of the functional view. At the same time, a planning problem is defined and modeled thanks to a network of tasks that and “AND”, “OR” nodes. Hence, the modeling language presented in [AVDG08] lacks various of the features of PRODPROC, although it could be extended to implement them.

9.4 Comparison Summary

In this section we provide a brief summary of the comparisons presented in this chapter. We consider a set of features that are relevant for product and production process modeling. For each selected approach included in the comparisons, we indicate whether or not it supports each feature.

Table 9.1 summarizes the comparisons of PRODPROC with selected systems and tools for product and process modeling. The features that we consider relevant for product and production process modeling are: modular structure of product models (column *Mod*), *has-part* relations with non-fixed cardinalities (column *N-f*), arithmetic constraints (column *ACs*), cardinality constraints (column *CCs*), global constraints (column *GCs*), modeling of activities and temporal relations (column *AsTs*), modeling of composite activities (CAs), modeling of multiple instance activities (MAs), resource constraints (column *RCs*), duration constraints (column *DCs*). In the table, a ✓ denotes available support for a feature, a × denotes the absence of support for a feature, and s ~ denotes limited support for a feature.

System/Tool	Mod	N-f	ACs	CCs	GCs	AsTs	CAs	MAs	RCs	DCs
Kumbang	✓	✓	~	×	×	×	×	×	×	×
Configit	✓	×	~	×	×	×	×	×	×	×
MCE	✓	✓	✓	×	✓	×	×	×	×	×
Ontologies	✓	✓	~	×	×	×	×	×	×	×
Feature Diagrams	✓	✓	~	×	×	×	×	×	×	×
BPMN	×	×	×	×	×	✓	✓	✓	~	~
YAWL	×	×	×	×	×	✓	✓	✓	~	~
DECLARE	×	×	×	×	×	✓	×	×	×	×
SysML	✓	✓	✓	✓	×	✓	✓	~	×	×
C-iEPC	×	×	×	×	×	✓	×	×	×	×
Provop	×	×	×	×	×	✓	×	×	×	×
ProdProc	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 9.1: Comparison summary.

Part III

PRODPROC Implementation

Chapter 10

Prolog Implementation of PRODPROC Models

In order to show the effectiveness of the PRODPROC framework presented in Chapter 5, and a possible use of the CSP encoding introduced in Chapter 7, we implemented a modeling system that allows a user to define a product description using the PRODPROC graphical language, to check model syntactic correctness, and to automatically generate product instances to check model validity.

In particular, we used SWI Prolog to develop a CLP-based modeling system, called PRODPROC Modeler, that exploit the close relation between configuration problems and CSPs. The choice of Prolog and CLP, instead of an imperative language and Constraint Programming, has been motivated by the advantages in terms of rapid software prototyping that the declarative nature of both Prolog and CLP gives. Figure 10.1 shows a screenshot of PRODPROC Modeler.¹

Once a PRODPROC model has been created using the tool graphical interface, the user can check its syntactic correctness and its validity. The former check aims at determining if the model respects the PRODPROC framework specifications (e.g., the presence of a root node in the product model graph). The latter check, instead, consists in the automatic generation of a PRODPROC instance, in order to prove the existence of (at least) a valid configuration for the model.

To implement both the syntax and the validity checking, we devised a representation of PRODPROC models based on Prolog predicates. The main purpose of this representation is to have a description of a PRODPROC model that consists of only the data that are relevant for syntax and validity checking, and that is independent from the graphical aspects of PRODPROC models.

In the following sections we briefly present the architecture of PRODPROC Modeler, and describe the Prolog predicates for representing PRODPROC models. A predicate descriptions consists of the predicate name printed in typewriter font, followed by the arguments

¹PRODPROC Modeler can be downloaded from http://www.dmi.unipg.it/dario.campagna/software/ProdProc_Modeler.html.

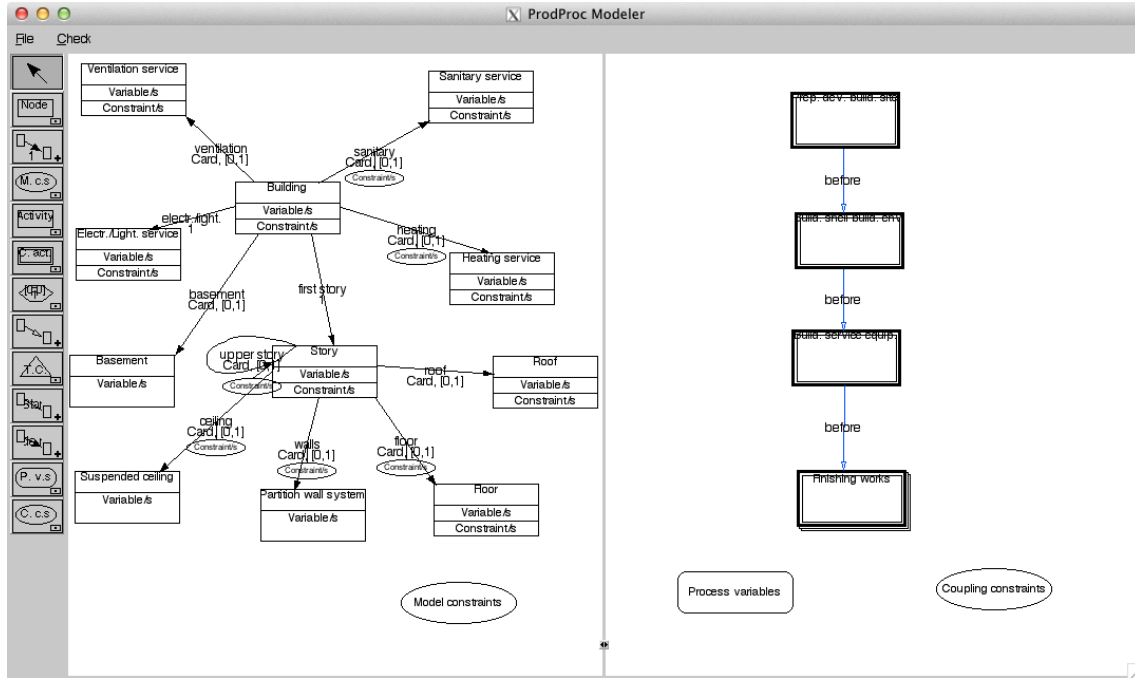


Figure 10.1: The PRODPROC modeling tool.

in *italics*. We assume all argument fully instantiated to a term that satisfies the required argument type.

10.1 Modeling System Architecture

The architecture of PRODPROC Modeler is pictorially described in Figure 10.2. The modeling tool consists of three main parts: a graphical user interface, a syntax checker, and a validity checker.

The graphical user interface has been implemented using the PCE library of SWI Prolog. It consists of 16 Prolog modules, for a total of about 8300 lines of code including comments. The interface allows a user to create and save PRODPROC models. Each model is saved into two files, one for the graphical elements drawn by a user, and one for the Prolog predicates defining its Prolog representation. Basically, the Prolog representation associates to each PRODPROC modeling feature a predicate having as arguments the constituting elements of the feature. This representation is automatically generated once a graphical model has been created. For each (graphical) element of the model the interface creates the corresponding Prolog predicate.

Once a model has been saved, the user can start the syntax and the validity checking of the model. The syntax checker has been implemented in Prolog, it consists of 1 Prolog module for a total of 794 lines of code (including comments). The validity checker has been implemented in Prolog too, it uses the CLPFD library of SWI Prolog to propagate

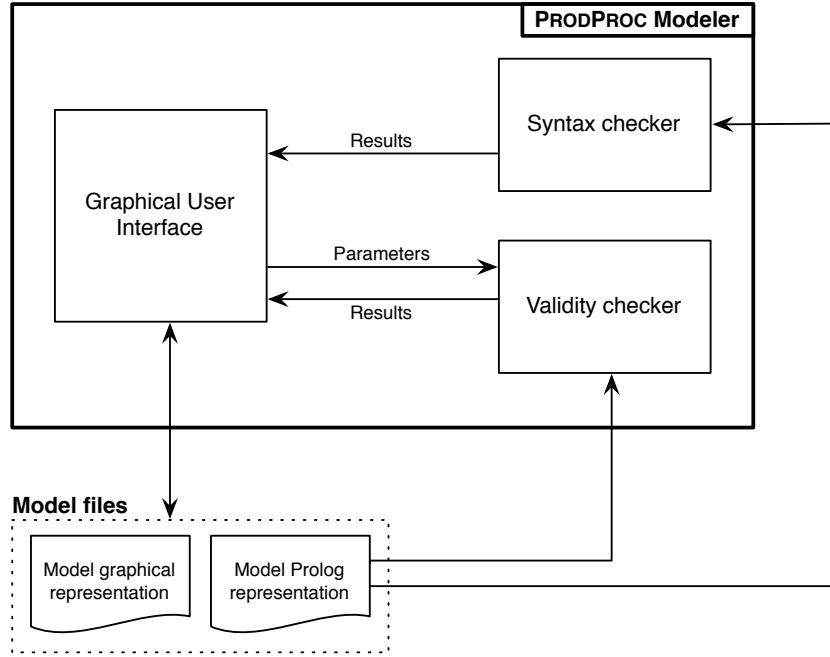


Figure 10.2: Architecture of the PRODPROC-based modeling tool.

constraints of CSPs associated to (partial) candidate instances. The implementation of the validity checker consists of 5 Prolog modules, for a total of about 3800 lines of code (including comments). The syntax checker takes as input only the Prolog representation of a model, while the validity checker requires also the user to set some parameters. Both the syntax checker and the validity checker return the results of their computation to the user interface upon termination.

10.2 PRODPROC Variable Domains as Types

In order to ease the implementation of syntax and validity checking, we consider finite domains of variables in a PRODPROC models as types with a name and a sort. In particular, we associate a predicate `type/3` to each finite domain (of integers or strings) appearing in a PRODPROC model. The predicate `type/3` for a domain D is defined as follows.

`type(TypeName, Sort, Domain)`

The argument *TypeName* is an atom representing the name of a type. *Sort* is the atom `int` when D is an integer domain, while it is the atom `string` when D is a string domain. *Domain* is a term representing the domain D . If $D = [\min, \max]$ *Domain* is the term `(min,max)`. If $D = \{i_1, \dots, i_n\}$, with i_1, \dots, i_n integers, *Domain* is a list of integers representing the set D (e.g., `[12,26]` for the set $\{12,26\}$). If $D = \{s_1, \dots, s_n\}$, with s_1, \dots, s_n ASCII strings, *Domain* is a list of atoms representing the set of strings D (e.g.,

[`'medium'`, `'normal'`] for the set {medium, normal}). For example, let us consider the following domains taken from the case study presented in Chapter 6:

`{0,1}`, `[7,90]`, `{Warehouse, Officebuilding}`.

They lead to the generation of the following Prolog predicates:

```
type(optional, int, [0,1]), type(widthLength, int, (7,90)),
type(buildingType, string, ['Warehouse', 'Office building']).
```

`type/3` predicates makes it easy to define a bijection $B : \text{strings} \rightarrow \mathbb{Z}$, necessary to handle strings in the finite domain solver of SWI Prolog, and to associate SWI Prolog CLPFD finite domains to domains in a PRODPROC model.

10.3 PRODPROC Constraints

In a PRODPROC model we may have different types of constraints, i.e., node constraints, cardinality constraints, duration constraints, etc. All these constraint types share the same syntax, and differ only in the variables they involve. For the Prolog representation, we define constraints inductively using Prolog terms. In the following, we will use the term *variable* to refer to node variables, meta variables, cardinality variables, process variables, resource quantity variables, instance number variables.

<i>IntTerm</i>	$:= n \in \mathbb{Z} \mid \text{integer variable } X \mid$ $\mid \text{plus}(IntTerm, IntTerm) \mid \text{minus}(IntTerm, IntTerm) \mid$ $\mid \text{times}(IntTerm, IntTerm) \mid \text{frac}(IntTerm, IntTerm) \mid$ $\mid \text{minus_sign}(IntTerm) \mid \text{mod}(IntTerm) \mid \text{mod}(IntTerm)$
<i>StringTerm</i>	$:= \text{Prolog atom} \mid \text{string variable } X$
<i>PrimitiveConstraint</i>	$:= \text{lt}(IntTerm, IntTerm) \mid \text{leq}(IntTerm, IntTerm) \mid$ $\mid \text{eq}(IntTerm, IntTerm) \mid \text{neq}(IntTerm, IntTerm) \mid$ $\mid \text{geq}(IntTerm, IntTerm) \mid \text{gt}(IntTerm, IntTerm) \mid$ $\mid \text{eq}(StringTerm, StringTerm) \mid$ $\mid \text{neq}(StringTerm, StringTerm) \mid$ $\mid \text{valid_tuples}(Vars, Tuples) \text{ where } Vars \text{ is a list of}$ $\text{variables and } Tuples \text{ is a list of lists of integers/strings}$
<i>Constraint</i>	$:= \text{PrimitiveConstraint} \mid \text{not}(Constraint) \mid$ $\mid \text{and}(Constraint, Constraint) \mid$ $\mid \text{or}(Constraint, Constraint) \mid$ $\mid \text{impl}(Constraint, Constraint)$

This Prolog term-based constraint representation makes it easy to check constraint syntactic correctness, and to implement the instantiation mechanisms presented in Chapter 5.

10.4 PROMo Modeling Features

In this section, we show how the modeling features of the PROMo language can be represented using Prolog predicates.

10.4.1 Product Model Graph

The Prolog representation of product model graphs is based on two Prolog predicates: the predicate `node/3` for representing PROMO nodes, and the predicate `edge/5` for representing PROMO edges.

The predicate `node/3` for representing a PROMO node $\langle N, \mathcal{V}, \mathcal{C} \rangle$ is defined as follows.

`node(NodeName, NodeVariables, NodeConstraints)`

NodeName is an atom representing the name N of the node. *NodeVariables* is a list representing the set of node variables \mathcal{V} . A node variable $\langle V, D(V) \rangle$ is represented as a term of the form $(VariableName, TypeName)$, where *VariableName* is an atom representing the node variable name V , and *TypeName* is an atom representing the name of the type associated to $D(V)$. *NodeConstraints* is a list representing the set of node constraints \mathcal{C} . Each constraint is represented using Prolog terms as explained in Section 10.3. A variable belonging to \mathcal{V} appears in a constraint as an atom representing its name. A variable belonging to a node ancestor appears in a constraint as the term $(Var, Node, MetaPath)$, where *Var* is an atom representing the variable name, *Node* is an atom representing the name of the ancestor, and *MetaPath* is a list of atoms representing edge labels, and wild-cards `_` and `*`, encoding a meta-path. For example, the Prolog representation of the node *Building* presented in Chapter 6 is the following.

```
node('Building',
    [
        ('BuildingType', buildingType), ('StoryNum', storyNum),
        ('Width', widthLength), ('Length', widthLength)
    ],
    [
        impl(eq('BuildingType', 'Warehouse'), eq('StoryNum', 1)),
        impl(gt('StoryNum', 1), eq('BuildingType', 'Office building'))
    ]).
```

The predicate `edge/5` for representing the PROMO edge $\langle label, N, M, C, CC \rangle$ has the following definition.

`edge(Label, ParentName, ChildName, Card, CardConstraints)`

Label is an atom representing the edge label *label*. *ParentName* and *ChildName* are atoms representing the name of the parent node *N* and of the child node *M*, respectively. If the cardinality *C* is an integer *n*, *Card* is the integer *n*. Otherwise, if *C* is a cardinality variable $\langle V, D(V) \rangle$, *Card* is the term $(CardName, TypeName)$, where *CardName* is an atom representing the cardinality name *V*, and *TypeName* is an atom representing the name of the type associated to *D(V)*. *CardConstraints* is a list representing the set of cardinality constraints *CC*. Each constraint is represented using Prolog terms as explained in Section 10.3. A variable belonging to *N* appears in a constraint as an atom representing its name. A variable belonging to an ancestor of *N* appears in a constraint as the term $(Var, Node, MetaPath)$, where *Var* is an atom representing the variable name, *Node* is an atom representing the name of the ancestor, and *MetaPath* is a list of atoms representing edge labels, and wild-cards `_` and `*`, encoding a meta-path. The cardinality variable of the edge appears in a constraint as the atom *CardName*. For example, the Prolog representation of the edge *upper story* presented in Chapter 6 is the following.

```
edge('upper story','Story','Story',('Card',optional),
[
    impl(eq('FloorNum',('StoryNum','Building',['first story',*])),
        eq('Card',0))
    ,
    impl(lt('FloorNum',('StoryNum','Building',['first story',*])),
        eq('Card',1))
]).
```

10.4.2 Model Constraints

PROMO model constraints are represented in Prolog using four different predicates. Meta-variables occurring in model constraints are represented in such predicates as terms of the form $(Var, Node, MetaPath)$, where *Var* is an atom representing a variable name, *Node* is an atom representing the name of the node the variable belongs to, and *MetaPath* is a list of atoms representing edge labels, and wild-cards `_` and `*`, encoding a meta-path.

Node model constraints appearing in a PROMO model are represented in Prolog using the predicate `nodeModelConstraint/1`, it is defined as

```
nodeModelConstraint(Constraint)
```

where *Constraint* is a Prolog term representing a node model constraint. It is defined as explained in Section 10.3.

allDifferent and aggregation constraints are represented in Prolog using two ad-hoc predicates. The predicate `allDifferent/1` represents a constraint *allDifferent(L)* as:

```
allDifferentValues(Variables)
```

where *Variables* is a list of terms of the form $(Var, Node, MetaPath)$, representing the list of meta-variables *L*. The predicate `aggConstraint/4` represents a constraint of the form *aggConstraint(f, L, op, n)* as

`aggConstraint(Function, Variables, RelOp, n)`

The argument *Variables* is a list of terms of the form $(Var, Node, MetaPath)$, representing the list of meta-variables L . *Function* is either the atom `sum` (when $f = sum$) or the atom `avg` (when $f = avg$). *RelOp* is an atom whose value depends on *op* as shown in Table 10.1.

<i>op</i>	<i>RelOp</i>
=	<code>eq</code>
≠	<code>neq</code>
< (≤)	<code>lt</code> (<code>leq</code>)
> (≥)	<code>gt</code> (<code>geq</code>)

Table 10.1: Correspondence between relational operators and Prolog atoms.

Cardinality model constraints appearing in a PROMO model are represented in Prolog using the predicate `cardModelConstraint/1`, it is defined as

`cardModelConstraint(Constraint)`

Constraint is a Prolog term representing a cardinality model constraint. It is defined as explained in Section 10.3. A cardinality variable appears in *Constraint* as a term of the form $(Label, ParentNode, ChildNode, Card)$, where *Label* is an atom representing the label of an edge from node N to node M , *ParentNode* and *ChildNode* are atoms representing the names of N and M , respectively, and *Card* is an atom representing the name of the cardinality variable of the edge labeled *Label*.

The followings are the Prolog representations of two model constraints of the PROD-PROC model presented in Chapter 6.

```
nodeModelConstraint(eq(('FloorNum','Story',['first story']),1)).
```

```
cardModelConstraint(neq(('upper story','Story','Story','Card'),
                        ('roof','Story','Roof','Card'))).
```

10.5 MART Modeling Features

In this section, we show how the modeling features of the MART language can be represented using Prolog predicates.

10.5.1 MART Model Variables

MART process variables are represented in Prolog using the predicate `processVariable/2`, it is defined as

`processVariable(VariableName, TypeName)`

VariableName is an atom representing a process variable name, and *TypeName* is an atom representing the name of a type.

Resource variables are represented in Prolog using the predicate `res/3`. This predicate is used to represent both the resource name and domain, and the resource initial value. The predicate `res/3` is defined as

```
res(ResourceName, TypeName, IV)
```

ResourceName is an atom representing a resource variable name. *TypeName* is an atom representing the name of a type. *IV* is an integer representing the initial value of resource *ResourceName*. For example, the process variable $\langle Vent, [0,1] \rangle$ is represented by the predicate `process_variable('Vent', optional)`, the resource *GeneralWorkers* presented in Chapter 6 is represented by the predicate `res('General Workers', zeroTen, 10)`.

10.5.2 Resource constraints

The Prolog predicate `resource_constraint/5` is used to represent resource constraint, i.e., tuple of the form $\langle A, R, q, TE \rangle$.

```
resource_constraint(ActivityName, ResourceName, Q, TimeExtent, Cond)
```

ActivityName is an atom representing the name of the activity *A*. *ResourceName* is an atom representing the resource name *R*. If $q = n$, $n \in \mathbb{Z}$, $Q = n$, otherwise $q = (V, D(V))$ and *Q* is the term $(Var, TypeName)$, where *Var* is an atom representing the variable name *V*, and *TypeName* is an atom representing the name of the type associated to *D(V)*. *TimeExtent* is an atom representing the time extent *TE* (one of: 'FromStartToEnd', 'BeforeStart', 'BeforeEnd', 'AfterStart', 'AfterEnd', 'Always'). *Cond* is a constraint on process variable names representing a condition, or the atom `true` if the resource constraint has no condition. For example, the resource constraint between the activity “Floor installation” and the resource *GeneralWorkers* presented in Chapter 6, is represented as follows.

```
resource_constraint('Floor installation', 'GeneralWorkers',
                   ('q_GW', tenTwo), 'FromStartToEnd', true).
```

10.5.3 Activities

Atomic activities, composite activities, multiple instance activities, and multiple instance composite activities, are all represented in Prolog using the predicate `activity/5`.

```
activity(ActivityName, AType, Proc, Insts, DurationConstraints)
```

ActivityName is an atom representing the name of an activity. *AType* is one of the following atoms: `atomic` for atomic activities, `comp` for composite activities, `multi` for multiple instance activities, `multicomp` for multiple instance composite activities. *Proc* is either the name of the composite (or multiple instance composite) activity the activity named *ActivityName* belongs to, or the atom `main` if the activity named *ActivityName* is belongs to the main process. When *AType* is the atom `multi` or `multicomp`, *Insts* is a term of the

form $(I, \text{TypeName})$, where I is an atom representing an instance number variable name, and TypeName is an atom representing a type name. Otherwise, Insts is the atom `nil`. $\text{DurationConstraints}$ is a list of duration constraints for the activity. Each constraint is defined as explained in Section 10.3. Atoms are used to represent activity durations, process variables names, and quantity resource variables names occurring in duration constraints. Let us consider, for example, the activities “Building shell and building envelope works”, “Finishing works”, and “Floor installation” presented in Chapter 6, the followings are the Prolog predicates representing them.

```
activity('Building shell and building envelope works',comp,main,nil,[]).
```

```
activity('Finishing works',multicomp,main,
        ('insts_finishing_works',type_2),[]).
```

```
activity('Floor installation',atomic,'Finishing works',nil,
        [ eq('d_Floor installation',
              frac('BuildingArea',times(2,abs('q_GW')))) ]).
```

10.5.4 Temporal Constraints

We define temporal constraints inductively using Prolog terms. If-conditional and iff-conditional constraints are also defined using Prolog terms. In the following, we denote with ActName an atom representing the name of an activity, and with Cond a constraint on process variable names representing a condition.

$$\begin{aligned} \text{AtomicTempConstr} \quad := \quad & \text{before}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{after}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{meets}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{met_by}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{overlaps}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{overlapped_by}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{during}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{includes}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{starts}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{started_by}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{finishes}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{finished_by}(\text{ActName}, \text{ActName}) \mid \\ & \mid \text{equals}(\text{ActName}, \text{ActName}) \mid \end{aligned}$$


```

| must_be_executed(ActName, ActName) |
| is_absent(ActName, ActName) |
| not_co_existent_with(ActName, ActName) |
| succeeded_by(ActName, ActName)
TempConstr      := AtomicTempConstr |
| and(TempConstr, TempConstr) |
| or(TempConstr, TempConstr)
CondTempConstr  := cond(if(Cond), TempConstr) |
| cond(iff(Cond), TempConstr)

```

We use the Prolog predicate `temporal_constraint/1` to represent a (if-conditional, iff-conditional) temporal constraint.

```
temporal_constraint(Constraint)
```

Constraint is a term representing a temporal constraint, or an if-conditional temporal constraint, or an iff-conditional temporal constraint. The followings are two predicates representing temporal constraints of the case study presented in Chapter 6.

```
temporal_constraint(before('Roof insulation', 'Heating rough assembly')).

temporal_constraint(cond(iff(eq('San', 0)),
                           is_absent('Sanitary rough assembly'))).
```

10.5.5 Coupling Constraints

The Prolog predicate `coupling_constraint/1` represents a coupling constraint.

```
coupling_constraint(Constraint)
```

Constraint is a coupling constraint. It is defined as explained in Section 10.3. Process variables names are represented in *Constraint* using atoms. Node variables and cardinality variables are represented in *Constraints* using the term $(Var, Node, MetaPath)$ and the term $(Label, ParentNode, ChildNode, Card)$, respectively. The followings are two predicates representing coupling constraints of the case study presented in Chapter 6.

```
coupling_constraint(eq(times(('Width', 'Building', []),
                           ('Length', 'Building', [])), 'BuildingArea')).

coupling_constraint(eq(('floor', 'Story', 'Floor', 'Card'), 'Floor')).
```

Chapter 11

CSP-based Automatic Instance Generation

The validity checker introduced in Chapter 10, allows a user to check the validity of a model by automatically generating PRODPROC instances. This validity checker implements a search algorithm that automatically generates a PRODPROC instance, given the Prolog representation of a PRODPROC model. The algorithm exploits the CSP encoding of PRODPROC models and (partial) candidate instances presented in Chapter 7.

The general structure of the algorithm for PRODPROC instance generation is shown in Algorithm 1. The algorithm takes as input a PRODPROC model M , two integers min , max ($max \geq min > 0$), and an integer $tLimit$. It returns as output either a PRODPROC instance $\langle I_{Prod}, I_{Proc} \rangle$ such that the instance tree has n nodes, with $min \leq n \leq max$, and the production process total duration is less or equal to $tLimit$ (when such an instance exists), or a tuple $\langle nil, nil \rangle$ (when no instance can be found).

Algorithm 1 PRODPROC instance generator

Input PRODPROC model M , integers min , max s.t. $max \geq min > 0$, integer $tLimit$

Output $\langle Prod, Proc \rangle$

```
1: ACTIVITYHIERARCHY( $M, AH$ )
2: INSTANTIATEIT( $M, IT$ )
3: CREATEPRODCSP( $M, IT, CSP$ )
4: PROPAGATE( $CSP, PropExit$ )
5: if  $PropExit = true$  then
6:   INSTANTIATENODEPOOL( $M, IT, NodePool$ )
7:   GENERATEPP( $M, min, max, tLimit, AH, IT, CSP, [], NodePool, Prod, Proc$ )
8: else
9:    $Prod \leftarrow nil$ 
10:   $Proc \leftarrow nil$ 
11: end if
```

Algorithm 1 starts by calling the procedure `ACTIVITYHIERARCHY`, this procedure generates a tree representing the activity hierarchy determined by composite activities in the model. Then, the procedure `INstantiateIT` is called (line 2). `INstantiateIT` first creates an instance of the root node of the product model graph of M , then it creates instances of nodes connected to the root node through edges of the product model graph having constant cardinalities. The same is done for each generated node instance, i.e., for each new node instance, children are generated considering edges of the product model graph having constant cardinalities. Once an initial instance tree IT has been generated, the procedure `CREATEPRODCSP` is called (line 3). `CREATEPRODCSP` implements the functions presented in Chapter 5 and Chapter 7 in order to obtain a CSP encoding a product configuration problem. The constraints of the CSP are then propagated using the procedure `PROPAGATE`. If the constraint propagation leads to empty variable domains, then no `PRODCSP` instance can be generated (lines 9 and 10). Otherwise, the procedure `GENERATEPP` is called (line 7).

The procedure `GENERATEPP`, shown in Algorithm 2, implements a backtracking search algorithm that aims to find a `PROMO` instance, and a corresponding `MART` instance. First, the procedure `ADDNODES` is called (line 2) to generate a `PROMO` instance whose instance tree has n nodes, with $min \leq n \leq max$. If such an instance exists, the procedure `PROCESSINSTANCE` is called in order to compute a corresponding `MART` instance (line 4). If no `MART` instance is found, then `GENERATEPP` searches for a new product instance. First, it backtracks to the last choice point in the search tree for product instance generation (line 6). Then, if such a choice point exists, it searches for a new `PROMO` and `MART` instance (line 8).

Algorithm 2 Procedure `GENERATEPP`

```

1: procedure GENERATEPP( $M, min, max, tL, AH, IT, CSP, States, Pool, Prod, Proc$ )
2:   ADDNODES( $M, min, max, IT, CSP, NodePool, States, Prod$ )
3:   if  $Prod \neq nil$  then
4:     PROCESSINSTANCE( $tL, M, AH, Prod, Proc$ )
5:     if  $Proc = nil$  then
6:       RESTORESTATE( $States, OS, OIT, OCSP, OP$ )
7:       if  $OldS \neq []$  then
8:         GENERATEPP( $M, min, max, tL, AH, OIT, OCSP, OS, OP, Prod, Proc$ )
9:       else
10:         $Prod \leftarrow nil$ 
11:      end if
12:    end if
13:  else
14:     $Proc \leftarrow nil$ 
15:  end if
16: end procedure

```

Both the procedure `ADDNODES` and the procedure `PROCESSINSTANCE` exploit the CSP encoding functions presented in Chapter 7. In the following sections, we give a detailed

description of these two procedures. We start presenting the ADDNODES procedure in Section 11.1. Then we discuss implementation issues related to resource constraints in Section 11.2, and describe the procedure PROCESSINSTANCE in Section 11.3.

11.1 Generation of Product Instances

The procedure ADDNODES, shown in Algorithm 3, implements a backtracking search algorithm that aims to find a PROMO instance whose instance tree has n nodes, with $\min \leq n \leq \max$. It has as arguments a PRODPROC model M , two integers \min , \max , a tree of node instances IT , a CSP CSP corresponding to the tree IT , a list of node instances $NodePool$ (instantiated in Algorithm 1 line 6), and a list of choice points $States$ (i.e., a list of tuples containing data for backtracking). At the end of the computation, the argument $Prod$ will be either a PROMO instance, or nil .

Algorithm 3 Procedure ADDNODES

```

1: procedure ADDNODES( $M, \min, \max, IT, CSP, NodePool, States, Prod$ )
2:   MANDATORYNODES( $CSP, IT, Nodes$ )
3:   if  $Nodes \neq []$  then
4:     ADDMANDATORYNODES( $M, \min, \max, IT, Nodes, NodePool, States, Prod$ )
5:   else
6:     ADDNONMANDATORYNODES( $M, \min, \max, IT, NodePool, States, Prod$ )
7:   end if
8: end procedure

```

First of all, ADDNODES calls the procedure MANDATORYNODES (line 2) to find out if there are cardinality variables whose domains have been reduced to a singleton by constraint propagation (i.e., cardinality variables with a value assigned to), and computes a list of nodes $Nodes$ for which to create instances. If the list $Nodes$ is not empty, then the procedure ADDMANDATORYNODES is called (line 4). Otherwise, new node instances are created calling the procedure ADDNONMANDATORYNODES (line 6).

Algorithm 4 shows the procedure ADDNONMANDATORYNODES. This procedure adds to the instance tree IT new node instances, choosing an edge (whose cardinality is a variable) exiting from a node in $Pool$. First, the procedure computes the number of nodes, i.e., N , in the current instance tree (line 2). If N is between the minimum (\min) and maximum (\max) number of required node instances, ADDNONMANDATORYNODES only creates the CSP corresponding to the current instance tree (line 26), and calls the procedure LABELNODEVARS to assign values to node instance variables (line 27). If N is greater than \max , BACKTRACK is called (line 29). If N is less than \min , ADDNONMANDATORYNODES creates a choice point calling the procedure SAVESTATE (line 4), and adds new nodes to IT calling ADDNODESFROMPOOL (line 5). Each time an instance of a node M is created, edges exiting from M having fixed cardinalities are examined in order to create node instances that are mandatory given the product model graph. If the number of node instances $newN$ is still not sufficient, the CSP corresponding to the new instance tree is computed

(line 8), and its constraints are propagated using the procedure PROPAGATEBT (line 9). If PROPAGATEBT sets *Exit* to 0, the newly created node instances are added to the node pool (line 11), and ADDNODES is called to add further node instances (line 12). If *newN* is between *min* and *max*, ADDNONMANDATORYNODES only creates the CSP corresponding to the current instance tree (line 18), and calls the procedure LABELNODEVARS to assign values to node instance variables (line 19). If *newN* is greater than *max*, BACKTRACK is called (line 21).

Algorithm 4 Procedure ADDNONMANDATORYNODES

```

1: procedure ADDNONMANDATORYNODES(M,min,max,IT,Pool,States,Prod)
2:   NODENUMBER(IT,N)
3:   if N < min then
4:     SAVESTATE(States,IT,CSP,Pool,NewSts)
5:     ADDNODESFROMPOOL(M,IT,Pool,NewIT)
6:     NODENUMBER(NewIT,NewN)
7:     if newN < min then
8:       CREATEPRDPCSP(M,NewIT,NewCSP)
9:       PROPAGATEBT(NewCSP,NewIT,NewSts,M,min,max,Exit,PrBT)
10:      if Exit = 0 then
11:        ADDTOPPOOL(Pool,NewIT,NewP)
12:        ADDNODES(M,min,max,NewIT,NewCSP,NewP,NewSts,Prod)
13:      else
14:        Prod ← PrBT
15:      end if
16:    else
17:      if newN ≤ max then
18:        CREATEPRDPCSP(M,NewIT,NewCSP)
19:        LABELNODEVARS(NewIT,NewCSP,NewSts,M,min,max,Prod)
20:      else
21:        BACKTRACK(NewSts,M,min,max,Prod)
22:      end if
23:    end if
24:  else
25:    if N ≤ max then
26:      CREATEPRDPCSP(M,IT,CSP)
27:      LABELNODEVARS(IT,CSP,States,M,min,max,Prod)
28:    else
29:      BACKTRACK(States,M,min,max,Prod)
30:    end if
31:  end if
32: end procedure

```

Algorithm 5 shows the procedure PROPAGATEBT. This procedure starts by computing a CSP *LastCSP* (line 2) adding to the CSP *CSP* constraints related to the last added nodes

(i.e., constraints on cardinality variables). If the propagation of *LastCSP* constraints fails, the procedure BACKTRACK is called (line 13). Otherwise, the procedure TRYTOLABEL is called to determine if it is possible to add new nodes to the current instance tree (line 5). If it is so, PROPAGATEBT terminates assigning 0 to *Exit* (line 7), otherwise the procedure BACKTRACK is called (line 9). The procedure TRYTOLABEL generates a CSP like *LastCSP*, but with an additional constraints of the form $C_1 > 0 \vee C_2 > 0 \vee \dots \vee C_h > 0$, where $\forall i \in \{1, \dots, h\}$ C_i is a cardinality variable with no value assigned to. Then, it tries to label the variables in the CSP. If a solution is found, then it is possible to add new nodes to the current instance tree. This procedure allows to prune early branches of the search tree for PROMO instance generation.

Algorithm 5 Procedure PROPAGATEBT

```

1: procedure PROPAGATEBT(CSP,IT,States,M,min,max,Exit,Prod)
2:   LASTACTIONCONSTRAINTS(States,CSP,LastCSP)
3:   PROPAGATE(LastCSP,PropExit)
4:   if PropExit = true then
5:     TRYTOLABEL(IT,M,States,LabelExit)
6:     if LabelExit = true then
7:       Exit  $\leftarrow$  0
8:     else
9:       BACKTRACK(States,M,min,max,Prod)
10:      Exit  $\leftarrow$  1
11:    end if
12:  else
13:    BACKTRACK(States,M,min,max,Prod)
14:    Exit  $\leftarrow$  1
15:  end if
16: end procedure

```

The procedure BACKTRACK, shown in Algorithm 6, first backtracks to the last choice point calling the procedure RESTORESTATE (line 2). Then, if such a choice point exists, it calls the procedure ADDNODES to continue the search from the last choice point (line 4).

Algorithm 6 Procedure BACKTRACK

```

1: procedure BACKTRACK(States,M,min,max,Prod)
2:   RESTORESTATE(States,OldStates,IT,CSP,NodePool)
3:   if NewStates  $\neq$  nil then
4:     ADDNODES(M,min,max,IT,CSP,NodePool,OldStates,Prod)
5:   else
6:     Prod  $\leftarrow$  nil
7:   end if
8: end procedure

```

Algorithm 7 shows the procedure LABELNODEVARS. This procedure is called to label

variables of node instances in order to compute a PROMO instance. LABELNODEVARS first adds to the CSP CSP (corresponding to the current instance tree) constraints related to the last added nodes (line 2), and constraints of the form $C = 0$ for each cardinality variable C that has 0 in its domain, and that has not been assigned (line 3). If the propagation of the resulting CSP fails, BACKTRACK is called (line 13). Otherwise, variables are labelled calling LABELVARIABLES (line 6). If the labeling fails BACKTRACK is called (line 10), otherwise LABELNODEVARS calls the procedure PRODINSTANCE (line 8) to construct a PROMO instance $Prod$ from the instance tree IT and assignment computed for CSP variables.

Algorithm 7 Procedure LABELNODEVARS

```

1: procedure LABELNODEVARS( $IT, CSP, States, M, min, max, Prod$ )
2:   LASTACTIONCONSTRAINTS( $States, CSP, LastCSP$ )
3:   CARDZEROCONSTRAINTS( $LastCSP, States, IT, CurrentCSP$ )
4:   PROPAGATE( $CurrentCSP, PropExit$ )
5:   if  $PropExit = true$  then
6:     LABELVARIABLES( $CurrentCSP, LabelExit$ )
7:     if  $LabelExit = true$  then
8:       PRODINSTANCE( $IT, CurrentCSP, Prod$ )
9:     else
10:      BACKTRACK( $States, M, min, max, Prod$ )
11:    end if
12:  else
13:    BACKTRACK( $States, M, min, max, Prod$ )
14:  end if
15: end procedure

```

The procedure ADDMANDATORYNODES is shown in Algorithm 8. Its purpose is to add to the current instance tree IT node instances that are mandatory, i.e., determined by edges whose cardinality variable domains have been reduced to singletons by constraint propagation. First, the procedure ADDNODESTOIT is called to create instances for nodes in $Nodes$, and obtain a new instance tree $NewIT$ (line 2). Each time an instance of a node M is created, edges exiting from M having fixed cardinalities are examined in order to create node instances that are mandatory given the product model graph. If the number of node instances in $NewIT$, i.e., N , is less than the minimum required (min), the CSP corresponding to the new instance tree is computed (line 5), and its constraints are propagated using the procedure PROPAGATEBT (line 6). If PROPAGATEBT sets $Exit$ to 0, the newly created node instances are added to the node pool (line 8), and ADDNODES is called to add further node instances (line 9). If N is between min and the maximum number of required node instances (max), ADDMANDATORYNODES only creates the CSP corresponding to the current instance tree (line 15), and calls the procedure LABELNODEVARS to assign values to node instance variables (line 16). If N is greater than max , BACKTRACK is called (line 18).

Notice that the procedure ADDNODES backtracks only on choice points determined by the creation of non-mandatory node instances. Hence, when a process for a product

Algorithm 8 Procedure ADDMANDATORYNODES

```

1: procedure ADDMANDATORYNODES( $M, min, max, IT, Nodes, Pool, States, Prod$ )
2:   ADDNODESTOIT( $M, IT, Nodes, NewIT$ )
3:   NODENUMBER( $NewIT, N$ )
4:   if  $N < min$  then
5:     CREATEPROCSP( $M, NewIT, NewCSP$ )
6:     PROPAGATEBT( $NewCSP, NewIT, States, M, min, max, Exit, ProdBT$ )
7:     if  $Exit = 0$  then
8:       ADDTOPPOOL( $Pool, NewIT, NewPool$ )
9:       ADDNODES( $M, min, max, NewIT, NewCSP, NewPool, States, Prod$ )
10:    else
11:       $Prod \leftarrow ProdBT$ 
12:    end if
13:  else
14:    if  $N \leq max$  then
15:      CREATEPROCSP( $M, NewIT, NewCSP$ )
16:      LABELNODEVARS( $NewIT, NewCSP, States, M, min, max, Prod$ )
17:    else
18:      BACKTRACK( $States, M, min, max, Prod$ )
19:    end if
20:  end if
21: end procedure

```

instance $Prod$ is not found, and procedure GENERATEPP backtracks to search for a new product instance (Algorithm 2, line 6), ADDNODES produces a product instance whose instance tree is different from the one of $Prod$. That is, GENERATEPP does not search for a process instance for each possible product instance having a particular instance tree. We assume that the existence of a process instance for a given product instance does not strongly depends on assignments to node instance variables. This assumption helps in reducing the search space for PRODPROC instance generation.

11.2 Dealing with Resource Constraints

The resource constraints (and the temporal constraints) in a PRODPROC model define a scheduling problem with multiple tasks and multiple machines. As stated in Chapter 7, resource constraints can be encoded in a CSP using an ad-hoc **cumulatives** constraint. Such a constraint is not present either in SWI Prolog or in other constraint programming systems. Moreover, SWI Prolog does not have a built-in mechanism for defining new global constraints. However, for automatic instance generation purposes the **cumulatives** constraint presented in Chapter 7 is not indispensable. In fact, it is possible to use resource constraints to generate a set of constraints that are implied by the **cumulatives** constraint. Such a set can then be added to the CSP related to a (partial) candidate MART instance

to compute a production process instance.

Let us consider, for example, a resource $\langle R, [0,6] \rangle$, three activities A , B and C , and the following resource and temporal constraints.

$$\begin{aligned} &\langle A, R, \langle Q_A, [-2, -1] \rangle, FromStartToEnd \rangle, \\ &\langle B, R, -2, AfterStart \rangle, \quad \langle C, R, 3, BeforeEnd \rangle, \\ &initialValue(R, 3), \\ &A \text{ includes } C, \quad B \text{ overlaps } C. \end{aligned}$$

Let $I_A, F_A, I_B, F_B, I_C, F_C$, be finite domain variables with domain $[1,6]$, i.e., $[1, 2 \cdot k]$ with k the number of activities producing/consuming R . These variables represent the start and end time instants of activities A , B and C . We can compute the possible dispositions of A , B and C on the timeline solving a CSP on variables $I_A, F_A, I_B, F_B, I_C, F_C$, whose constraints are the temporal constraints instantiated on these variables, and the constraints $I_A < F_A, I_B < F_B, I_C < F_C$. That is, a CSP $C_{IF} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where:

$$\begin{aligned} \mathcal{V} &= \{I_A, F_A, I_B, F_B, I_C, F_C\}, \\ \mathcal{D} &= \{I_A \in [1,6], F_A \in [1,6], I_B \in [1,6], F_B \in [1,6], I_C \in [1,6], F_C \in [1,6]\}, \\ \mathcal{C} &= \{I_A < F_A, I_B < F_B, I_C < F_C, \\ &\quad I_B > I_A \wedge I_B < F_A \wedge F_B < F_A, \\ &\quad I_B < I_C \wedge F_B > I_C \wedge F_B < F_C\}. \end{aligned}$$

A solution of the CSP C_{IF} represents a qualitative scheduling of A , B and C on the timeline. For example, the solution $S = \{I_A = 1, F_A = 6, I_B = 2, F_B = 4, I_C = 3, F_C = 5\}$, represents the scheduling depicted in Figure 11.1. The scheduling is qualitative in the sense that it abstracts from activity durations, and time is discretized in a number of instants equal to two times the number of considered activities. From a solution of C_{IF} we can compute a constraint on variables $t_A^{start}, t_A^{end}, t_B^{start}, t_B^{end}, t_C^{start}, t_C^{end}$. For example, from the solution S we derive the following constraint.

$$Cond \equiv t_A^{start} < t_B^{start} \wedge t_B^{start} < t_C^{start} \wedge t_C^{start} < t_B^{end} \wedge t_B^{end} < t_C^{end} \wedge t_C^{end} < t_A^{end}$$

Given a qualitative scheduling and the resource constraints, it is possible to generate a constraint describing the usage of a resource during activity executions. In particular, we associate to each activity a couple of variables representing the quantity of a resource when the activity starts, and when the activity ends. We associate to A , B , and C the variables $S_A, E_A, S_B, E_B, S_C, E_C$, respectively, each of these variables has domain $[0,6]$. From the qualitative scheduling of Figure 11.1 we generate the following constraint.

$$\begin{aligned} Cons \equiv & IV_R = 3 + 3 \wedge S_A = IV + Q_A \wedge S_B = S_A - 2 \wedge S_C = S_B \wedge E_B = S_C \wedge \\ & \wedge E_C = E_B - 3 \wedge E_A = E_C - Q_A \end{aligned}$$

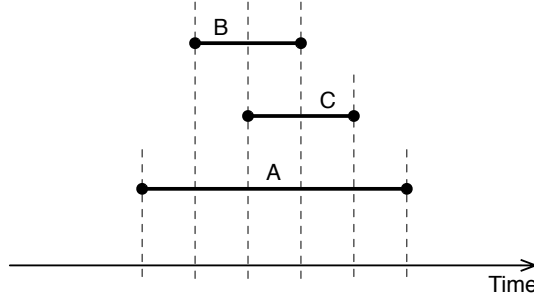


Figure 11.1: Qualitative scheduling of three activities.

This constraint represents the variation of quantity of resource R with respect to time, i.e., the function depicted in Figure 11.2 (in the figure we assume $Q_A = 1$). A constraint $Cond \wedge Cons$ represent a possible qualitative scheduling of the activities producing/consuming a resource, and the corresponding variation of quantity of the resource. In general, we can generate a constraint of the form $Cond \wedge Cons$ for each possible solution of a CSP on I and F variables, and for each resource in a model.

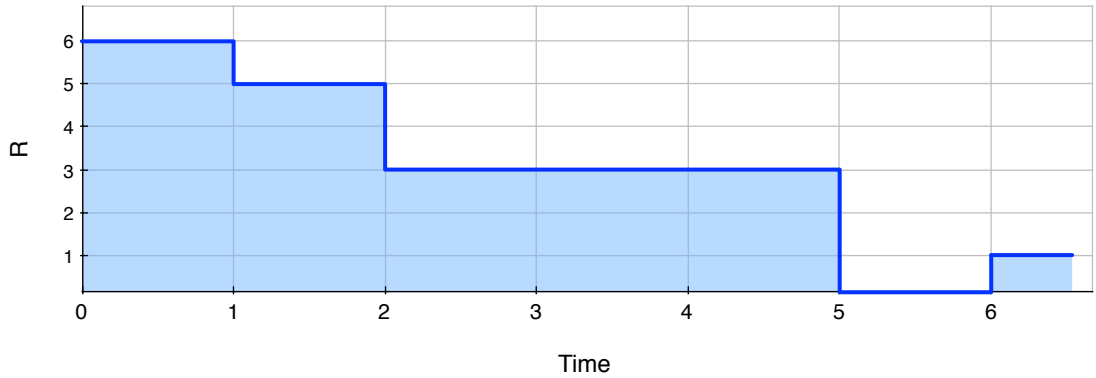


Figure 11.2: Resource quantity variations with respect to time.

Algorithm 9 shows the procedure CSPRCs, it computes, for each resource R in a model, a CSP on I and F variables to determine possible qualitative scheduling of activities producing/consuming R . The arguments of the procedure CSPRCs are: a solved CSP *CoupCSP* on product variables and multiple instance variables obtained from coupling constraints, a set of activity instances *ActInsts*, the activity hierarchy *AH* of the PRODPROC model, the set of temporal constraint *TCs* in the PRODPROC model. The procedure computes a set of CSP *CSPs* in the following way. For each resource R , CSPRCs collects the constraints on R (line 4), and compute the list *Racts* of activity instances producing/consuming R (line 5). If *Racts* is not empty, the procedure RCCSP is called (line 9) and the CSP on I and F variables is computed.

The procedure RCCSP is shown in Algorithm 10. First, the procedure computes the

transitive closure of temporal constraints on activities in *Racts* (i.e., activities producing/consuming a resource *R*), and collects the temporal constraints on activities in *Racts* (lines 2 and 3). The transitive closure of temporal constraints is computed as explained in Section 4 and 5 of [All83]. For each couple $\langle A, B \rangle$ of activities in *Racts*, we search for all paths of temporal constraints between *A* and *B*, and compute the transitivity constraints along them. The procedure **TEMPCONSTR** (line 3) also determines temporal constraints that are implied by constraints among composite activities. For example, let us suppose *A* and *B* are activities belonging to the composite activities *C* and *D*, respectively. If in the **PRODPROC** model there are the temporal constraints *C before E* and *E before D*, the procedure **TEMPCONSTR**, exploiting the activity hierarchy and the transitive closure of temporal constraints, determines the constraint *A before B*. After the collection of temporal constraints, *I* and *F* variables for activities in *Racts*, and the respective domain constraints are created (lines 4 and 5). Constraints of the form $I < F$ are generated by calling the procedure **BASICCONSTRAINT** (line 6).

Algorithm 9 Procedure CSPRCs

```

1: procedure CSPRCs(CoupCSP, ActInsts, AH, TCs, CSPs)
2:   CSPs  $\leftarrow$  []
3:   for all resources R in M do
4:     RESOURCECONSTRAINTS(R, M, Rconstraints)
5:     ACTIVITIESR(Rconstraints, ActInsts, Racts)
6:     if Racts  $\neq$  [] then
7:       LENGTH(Racts, K)
8:       twoTK  $\leftarrow K * 2$ 
9:       RCCSP(ActInsts, Racts, AH, TCs, CoupCSP, twoTK, CSP)
10:      ADDTOLIST(CSP, CSPs)
11:    end if
12:  end for
13: end procedure

```

Algorithm 10 Procedure RCCSP

```

1: procedure RCCSP(ActInsts, Racts, AH, TCs, CoupCSP, twoTK, CSP)
2:   TRANSITIVECLOSURE(Racts, TCs, CoupCSP, TranTCs)
3:   TEMPCONSTR(Racts, TCs, CoupCSP, TCsIFs)
4:   GENERATEIFVARS(Racts, IFs)
5:   DOMAINIFs(IFs, 1, twoTK, IFsDoms)
6:   BASICCONSTRAINTS(IFs, BCs)
7:   SYMMETRIES(IFs, Syms)
8:   INSTTCIFs(Racts, TranTCs, TCsIFs, ActInsts, IFs, AH, TCs, CoupCSP, ITCs)
9:   CSPIF(IFs, IFsDoms, BCs, Syms, ITCs, CSP)
10: end procedure

```

The constraints determined by the procedures **TRANSITIVECLOSURE**, **TEMPCONSTR**,

and BASICCONSTRAINTS, lead to a CSP that may have a lot of symmetric solutions. For example, the CSP for activities A , B and C presented before, has among its solutions the assignments $S_1 = \{I_A = 1, F_A = 5, I_B = 1, F_B = 3, I_C = 2, F_C = 4\}$ and $S_2 = \{I_A = 2, F_A = 6, I_B = 2, F_B = 4, I_C = 3, F_C = 5\}$, both S_1 and S_2 lead to the following constraint.

$$t_A^{start} = t_B^{start} \wedge t_B^{start} < t_C^{start} \wedge t_C^{start} < t_B^{end} \wedge t_B^{end} < t_C^{end} \wedge t_C^{end} < t_A^{end}$$

The procedure SYMMETRIES (line 7) generates a set of constraints that remove symmetric solutions. In particular, SYMMETRIES generates a **global_cardinality** constraint [Rég96], and a set of implications. For instance, for the activities A , B and C , it generates the following constraints.

$$\begin{aligned} &\text{global_cardinality}([I_A, F_A, I_B, F_B, I_C, F_C], \\ &\quad [(1, V_1), (2, V_2), (3, V_3), (4, V_4), (5, V_5), (6, V_6)]), \\ &V_1 = 0 \Rightarrow V_2 = 0, V_2 = 0 \Rightarrow V_3 = 0, V_3 = 0 \Rightarrow V_4 = 0, \\ &V_4 = 0 \Rightarrow V_5 = 0, V_5 = 0 \Rightarrow V_6 = 0, \end{aligned}$$

This set of constraints states that, if a value $i \in \{1, 6\}$ is not assigned to any of the I and F variables, then no value $j > i$ can be assigned to those variables. Notice that the CSP on A , B and C with the addition of the above **global_cardinality** constraint and implications, does not have S_2 as one of its solutions. In general, if k is the number of activities producing/consuming a resource, SYMMETRIES generates a constraint **global_cardinality**($IFs, Pairs$), where IFs is the list of I and F variables, and $Pairs$ is a list of pairs of the form (i, V_i) , for each $i \in [1, 2 \cdot k]$, and a set of implications $V_j = 0 \Rightarrow V_{j+1} = 0$, for each $j \in [1, 2 \cdot k - 1]$.

Once constraints for symmetric-solution elimination have been computed, RCCSP instantiate temporal constraints on activity instances in *Racts* (line 8), and generate a CSP on I and F variables (line 9).

Given a set of *CSPs* (one for each resource produced/consumed by activity instances) on I and F variables, it is possible to compute a satisfiable set of constraints of the form $Cond \wedge Cons$ by sequentially labeling the CSPs in *CPSs*. That is, we search for a solution of the first CSP in *CSPs*, and for each $i > 1$, we search for a solution of a CSP obtained adding to the i -th CSP in *CSPs* a set of constraints determined by solutions of previously solved CSPs (activity instances may produce/consume more than a resource, hence the same activity instances may be involved in different CSPs). Once the i -th CSP has been solved, we compute the constraints $Cond$ and $Cons$, and verify the satisfiability of $Cons$. If $Cons$ is satisfiable we proceed with the $(i + 1)$ -th CSP. Otherwise, we search for another solution of the i -th CSP. If no solution is found for the i -th CSP we search for another solution of the $(i - 1)$ -th CSP. This algorithm can be easily implemented in Prolog with a predicate like the one shown in Algorithm 11.

In Algorithm 11, the predicate **prevCSPsConstraints** adds to a CSP constraints derived from previously computed solutions. The predicate **solveCSP** search for a solution of

a CSP. If `solveCSP` fails, Prolog backtracks to the previously considered CSP, and searches for another solution for it. The predicate `condCons` computes the constraints *Cond* and *Cons* given the solution of a CSP, while `checkCond` checks the satisfiability of *Cons* creating and solving a CSP on variables in *Cons*, and with *Cons* as the only constraint. If `CHECKCOND` fails, Prolog backtracks to `solveCSP`, and searches for a new solution for the CSP under consideration.

Algorithm 11 Prolog predicate for sequential labeling of a list of CSPs

```
sequentialLabel([],_,RCs,RCs).
sequentialLabel([CSP|CSPs],ActInsts,PrevCSPs,[Cond,Cons|Tail1],Tail2) :-
    once(prevCSPsConstraints(PrevCSPs,CSP,NewCSP)),
    solveCSP(NewCSP),
    once(condCons(NewCSP,ActInsts,Cond,Cons)),
    checkCond(Cons),
    sequentialLabel(CSPs,ActInsts,[NewCSP|PrevCSPs],Tail1,Tail2).
```

11.3 Generation of Process Instances

The procedure `PROCESSINSTANCE`, shown in Algorithm 12, aims at generating a MART instance given a `PROMO` instance. That is, it searches for a production process for the realization of a generated product variant.

First, `PROCESSINSTANCE` generate a CSP *CoupCSP* on process variables and multiple instance variables, whose constraints are instantiated coupling constraints (line 2). Then, `LABELCOUPLING` is called to search for a solution of *CoupCSP* (line 4), i.e., an assignments of values to process variables and multiple instance variables (we assume that `LABELCOUPLING` produces a different solution each time it is called). If a solution for *CoupCSP* is found, a set of activity instance is generated by calling the procedure `ACTIVITYINSTANCES` (line 6). `ACTIVITYINSTANCES` generates a set of activity instances through a fix point computation that examines (if-conditional, iff-conditional, not conditional) *must_be_executed*, *is_absent*, *not_co_existent*, and *succeeded_by* temporal constraints. This computation stops when no more activity instances can be created. We consider optional the activities involved in if-conditional *must_be_executed* and *is_absent* constraints whose conditions are false, and assume that it is possible to obtain a process instance both with and without instances of such activities. This assumption reduces the search space for process instance generation, since we do not need to backtrack on activity instance generation when a set of activity instances does not lead to the creation a process instance.

Once a set of activity instances has been computed, `PROCESSINSTANCE` calls the procedure `MODELTEMPORALCONSTRAINTS` (line 7) to collect temporal constraints in the `PRODPROC` model, and the procedure `ADDBEFORECONSTRAINTS` (line 8) to add to the set of temporal constraints *TCs* a set of *before* constraints on multiple instances activities.

For example, given a multiple instance activities A for which instances a_1, a_2, a_3 , have been created, the procedure `ADDBeforeConstraints` creates a constraint A before A . The instantiation of this constraint leads to the constraints a_1 before a_2 , a_1 before a_3 , a_2 before a_3 . This *before* constraints impose an ordering on instances of multiple instance activities, and reduce the search space for process instances generation. If we do not find any process instance in presence of this additional *before* constraints, it is guaranteed that it is not possible to find process instances even removing them. The scheduling of instances of a multiple instance activity is determined only by resource constraints involving them (if any), if the scheduling imposed by *before* constraints is in conflict with resource constraints, then each other possible scheduling will be in conflict too.

The set of temporal constraints computed by `ADDBeforeConstraints` is instantiated calling the procedure `INstantiateTCS` (line 9). The procedure `DefaultTCS` (line 10) implements the function α presented in Chapter 5, and generates default constraints on duration, start time, and finishing time of each activity instance, while `EncodeProdRelCs` (line 11) add to the `PRODPROC` model the resource (and the other) constraints representing product related constraints. Once product related constraints have been encoded in the corresponding resource constraints, the procedure `CSPRCs` (line 12) is called to compute a CSP for each resource in the model, as explained in Section 11.2.

Algorithm 12 Procedure `PROCESSINSTANCE`

```

1: procedure PROCESSINSTANCE( $tL, M, Prod, AH, Prod, Proc$ )
2:   COUPLINGCSP( $M, Prod, CoupCSP$ )
3:   repeat
4:     LABELCOUPLING( $CoupCSP, CoupExit$ )
5:     if  $CoupExit = true$  then
6:       ACTIVITYINSTANCES( $M, CoupCSP, ActInsts$ )
7:       MODELTEMPORALCONSTRAINTS( $M, TCS$ )
8:       ADDBeforeConstraints( $TCS, ActInsts, NewTCS$ )
9:       INstantiateTCS( $NewTCS, CoupCSP, ActInsts, ITCs$ )
10:      DefaultTCS( $ActInsts, ImTCS$ )
11:      ENCODEPRODRELCS( $M, Mpr$ )
12:      CSPRCs( $Mpr, CoupCSP, ActInsts, AH, NewTCS, CSPs$ )
13:      repeat
14:        SEQUENTIALLABEL( $CSPs, ActInsts, RCs$ )
15:        INSTANTIATEDCS( $ActInsts, CoupCSP, RCs, DCs$ )
16:        PRCQCONSTRAINTS( $Mpr, RCs, PRQs$ )
17:        LABELPROC( $tL, ActInsts, ITCs, ImTCS, RCs, DCs, PRQs, Proc$ )
18:      until  $Proc = nil \wedge RCs \neq nil$ 
19:    else
20:       $Proc \leftarrow nil$ 
21:    end if
22:  until  $Proc = nil \wedge CoupExit = 1$ 
23: end procedure

```

The procedure `PROCESSINSTANCE` sequentially label the CSPs computed by `CSPRCs` by calling the procedure `SEQUENTIALLABEL` (line 14). This procedure computes a set of $Cond \wedge Cons$ constraints implied by resource constraints (we assume that `SEQUENTIALLABEL` computes a new set of constraints each time it is called). Duration constraints, and other constraints related to product related constraints, are instantiated calling the procedures `INSTANTIATEDCS` and `PRCQCONSTRAINTS`, respectively (lines 15 and 16). Then, the procedure `LABELPROC` is called (line 17) to compute and solve a CSP on activity instance durations, start and end times, and resource quantity variables, whose constraints are instantiated temporal constraints, instantiated duration constraints, and $Cond \wedge Cons$ constraints implied by resource constraints. If this CSP has a solution, then *Proc* represents a process instance, and `PROCESSINSTANCE` terminates. Otherwise, *Proc* = *nil*, and `PROCESSINSTANCE` searches for a new set of $Cond \wedge Cons$ constraints. If `SEQUENTIALLABEL` assign *nil* to *RCs*, then all the possible set of $Cond \wedge Cons$ constraints have been generated, and `PROCESSINSTANCE` searches for a new solution of *CoupCSP*. If *CoupCSP* set *CoupExit* to *false*, then *CoupCSP* has no (new) solution, and `PROCESSINSTANCE` terminates assigning *nil* to *Proc* (line 20).

The procedure `PROCESSINSTANCE`, as well as the other procedures presented in this chapter, has been implemented in Prolog. In particular, the behavior determined by the two repeat/until loops, and the behavior of the procedures `LABELCOUPLING` and `SEQUENTIALLABELING`, have been obtained by exploiting Prolog built-in backtracking mechanism.

Chapter 12

ProdProc Modeler

In this chapter, we describe the graphical user interface of PRODPROC Modeler, the modeling tool introduced in Chapter 10. In particular, we will exploit the case study presented in Chapter 6 in order to show how to use PRODPROC Modeler to create a PRODPROC model, and to check a model syntactic correctness and validity.

Figure 12.1 shows the main window of PRODPROC Modeler. It consists of four main components, namely, a menu bar, an area for drawing the product model (denoted as “Product canvas” in Figure 12.1), an area for drawing the process model (denoted as “Process canvas” in Figure 12.1), and a mode-selection menu for choosing the mode of operation in the drawing areas.

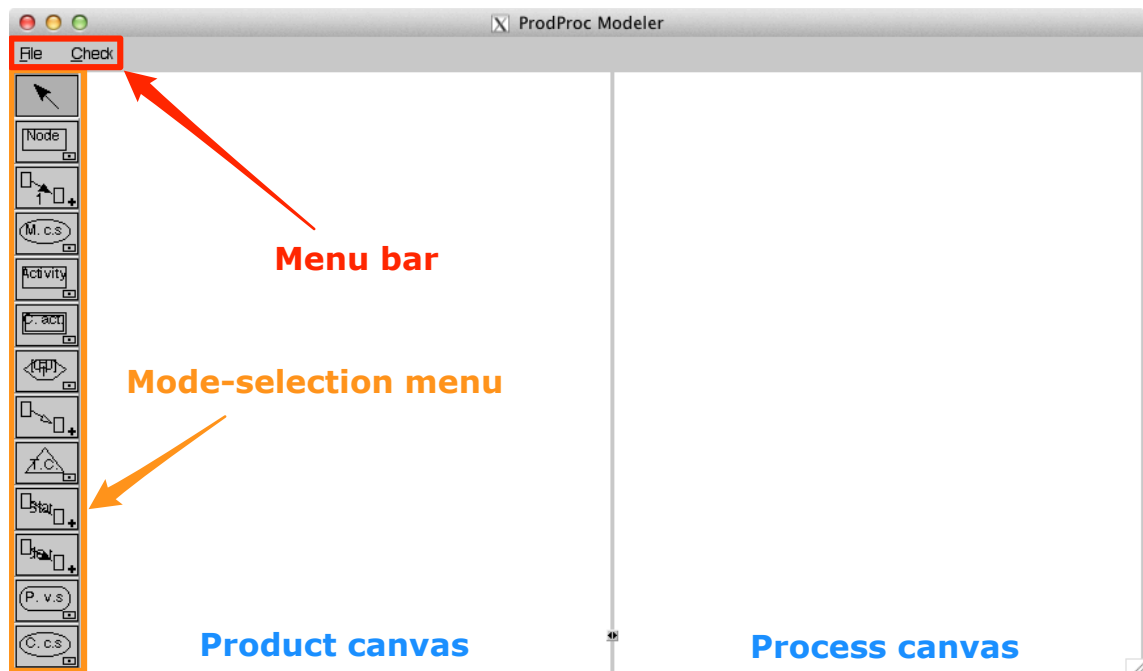


Figure 12.1: PRODPROC Modeler main window.

The PRODPROC Modeler menu bar, highlighted with a red rectangle in Figure 12.1, has two menus. The “File” menu, shown in Figure 12.2a, presents three commands for file operations, and one command for closing the modeling tool.

Save as: prompts the user for a file name *model_name*, and saves the PRODPROC model into the files *model_name.prp* and *model_name.pl*. The *.prp* file contains the graphical representation of the model, while the *.pl* file contains the Prolog representation of the model.

Save: saves a PRODPROC model into the files previously created with “Save as”.

Open: prompts the user for a *.prp* file, and shows in the “Product canvas” and in the “Process canvas” the content of the file.

Quit: quits the modeling tool.

The “Check” menu, shown in Figure 12.2b, presents two commands that can be activated once a model has been opened.

Syntax: starts the execution of the syntax checker on a PRODPROC model.

Validity: starts the execution of the validity checker on a PRODPROC model.

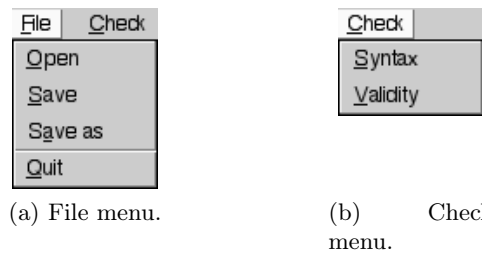


Figure 12.2: PRODPROC Modeler menus.

The PRODPROC Modeler mode-selection menu, highlighted with an orange rectangle in Figure 12.1, allows the user to change the mode of operation in the drawing areas, i.e., the event associated to a (mouse) left-click on the drawing areas. Table 12.1 shows the elements of the mode-selection menu (column *Selector*), and the operation modes associated to them (column *Operation mode*). The activation of a selector does not only change the operation mode in the drawing areas, it also changes the mouse cursor shape. The first selector in Table 12.1 activates the selection mode. When the select mode is active, the mouse cursor is an arrow, a left-click on an object makes it the selection, a shift-left-click on an object adds or deletes it to/from the selection, and a left-dragging indicates an area in which all objects should be selected. Each other selector in Table 12.1 causes a left-click (or a left-dragging) to draw an object. These selectors consist of an icon that indicates the mode and a small version of what will be drawn, plus a representation of the shape the mouse cursor will take (bottom-right corner of a selector). The different (drawing) operation modes are described in detail in Section 12.1 and Section 12.2.














Selector	Operation mode
	Select
	Draw PROMo nodes
	Draw PROMo edges
	Draw PROMo model constraint sets
	Draw MART process variable sets
	Draw MART activities
	Draw MART composite activities
	Draw MART resources
	Draw MART atomic temporal constraints
	Draw MART temporal constraints
	Draw MART resource constraints
	Draw MART <i>produces_for</i> constraints
	Draw MART coupling constraint sets

Table 12.1: PRODPROC modeler operation mode selectors.

While the effect of a left-click depends on the active operation mode, a right-click on a drawing area or on an object has always the same effect, i.e., it causes a pop-up menu to

appear. A right-click on a drawing area causes the pop-up menu shown in Figure 12.3a to appear, this menu has two commands.

Select mode: activates the select mode;

Paste: paste the content of the clipboard.

The pop-up menu shown in Figure 12.3b appears after a right-click on an object, it has three commands.

Cut: deletes the object from the drawing area and saves it to the clipboard;

Copy: saves the object to the clipboard;

Edit properties: opens the attribute window for the object.

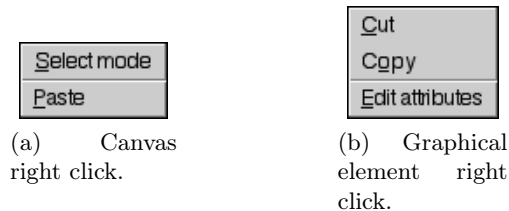


Figure 12.3: Right click menus.

To create a PRODPROC model a user does not only have to draw a product model graph in the “Product canvas”, and a temporal constraint network in the “Process canvas”, she/he also has to define non-graphical elements/characteristics of a PRODPROC model, e.g., node variables, edge cardinalities, process variables, etc. The command “Edit properties” of the pop-up menu in Figure 12.3b is of central importance for this purpose. It allows a user to define non-graphical elements/characteristics of a model through ad-hoc forms, i.e., the object attribute windows. PRODPROC Modeler checks values inserted in these forms, and shows messages like the ones in Figure 12.4 when (syntactic) errors are discovered.

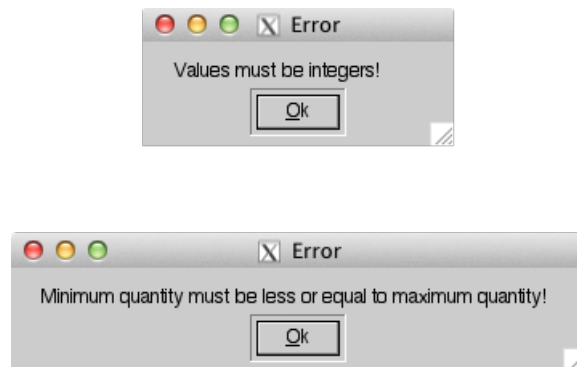


Figure 12.4: Error messages.

12.1 Creation of Product Models

In this section, we show how to create a product model using PRODPROC Modeler. First, we explain how to draw PROMO nodes and edges in order to define a product model graph. Then, we describe how to define node, edge, and model constraints.

12.1.1 Drawing a Product Model Graph

A product model graph can be created in PRODPROC Modeler drawing PROMO nodes, and connecting them using PROMO edges in the “Process canvas”.

PROMO nodes. When the operation mode “Draw PROMO nodes” is active, a left-click in the “Product canvas” draws a rectangle representing a node named “New node”, as shown in Figure 12.5. Once a node has been drawn, it is possible to assign to it a name and node

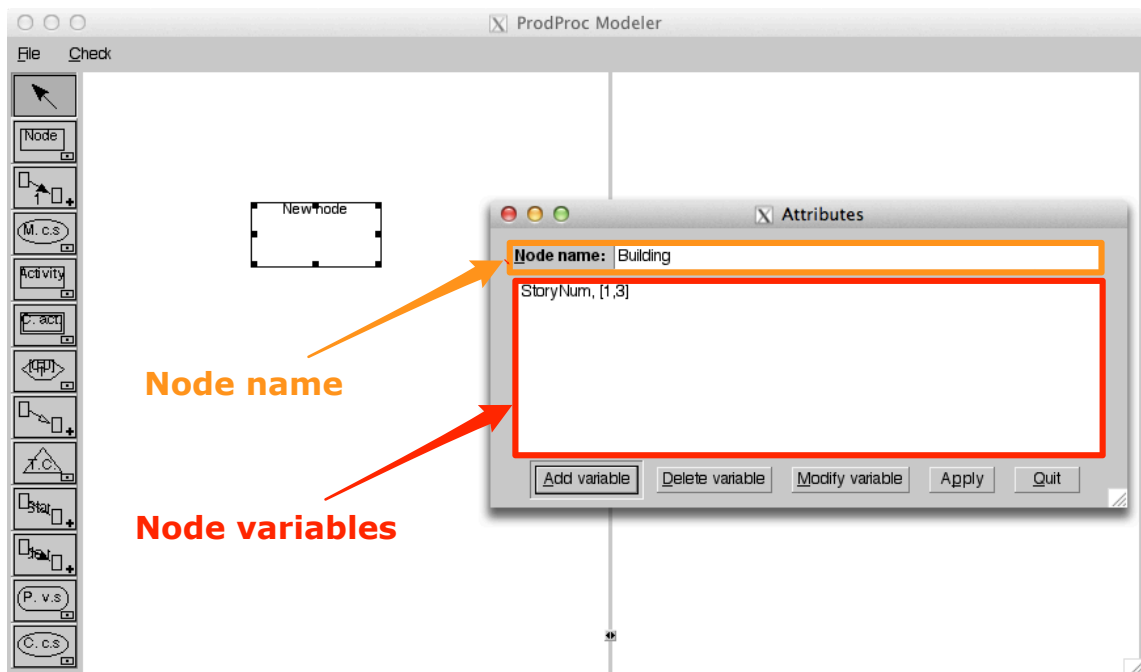


Figure 12.5: PROMO node creation.

variables. The command “Edit attributes” of the pop-up menu that appears after a right-click on the new node, opens a window titled “Attributes” like the one shown in Figure 12.5. The figure shows the (partial) definition of the node “Building” of the case study presented in Chapter 6. The name of the node can be set using the text box highlighted with an orange rectangle in the attribute window. The red rectangle in Figure 12.5 highlights the list box showing the node variables of the node. Variables in this list can be selected and then modified by clicking the button “Modify variable”, or deleted by clicking the button “Delete variable”. The button “Add variable” opens a window for node variable creation,

while the button “Modify variable” opens a window for variable modification. Finally, the button “Apply” saves the inserted data to the node.

Figure 12.6a shows the window for node variable creation. The radio buttons labeled “Variable type” allow one to set the variable type, i.e., either “String” or “Integer”. When the selected type is “Integer”, the radio buttons labeled “Integer domain format” allow one to choose between a set of integers or an interval as the domain format. The name of the variable can be set using the text box “Variable name”, while the set of strings or integers defining the variable domain can be set (as a comma separated list of integer or strings) using the text box “Values”. The numeric text boxes “Min value” and “Max value” allow one to set the minimum and maximum value of the interval representing the domain of an integer variable. The button “Add” creates a node variable with the name and domain chosen by the user.

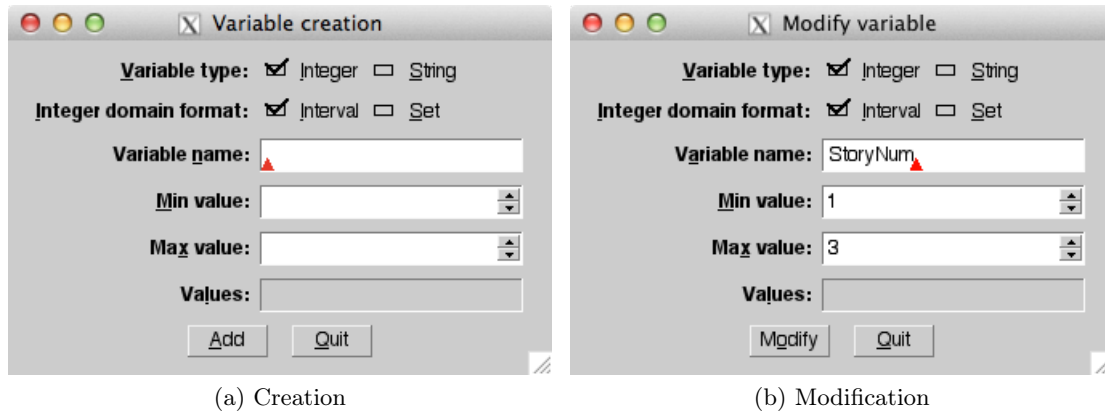


Figure 12.6: Creation and modification of node variables.

Figure 12.6b shows the window for node variable modification. It is similar to the one for node variable creation. When it is opened, radio buttons and text boxes are set accordingly to the variable that the user wants to modify. The button “Modify” modifies the selected node variable with the new data chosen by the user.

PROMO edges. An edge between two nodes can be drawn in the “Product canvas” when the mode “Draw PROMO edges” is active. To draw an edge a user has to left-click on the parent node of the edge he wants to draw, drag the cursor to the desired child node, and release the left mouse button when the cursor is over the child node (the parent and the child node can be the same node in the “Product canvas”). This gesture creates an edge with label “New edge” and cardinality equal to 1, as shown in Figure 12.7. The figure shows the (partial) definition of the edge “upper story” of the case study presented in Chapter 6. Once an edge has been drawn, it is possible to define its label and cardinality using the edge attribute window shown in Figure 12.7. The edge label can be set using the text box highlighted with a red rectangle in the attribute window. The orange rectangle in Figure 12.7 highlights the controls for setting the edge cardinality. The radio buttons

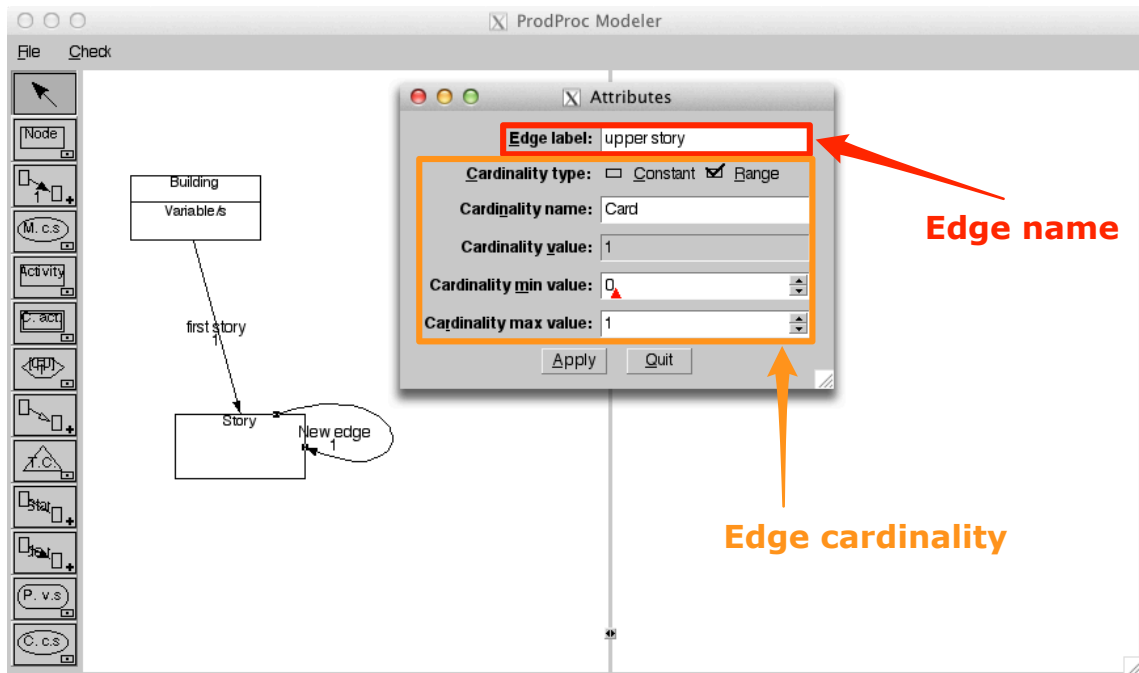


Figure 12.7: PROMO edge creation.

labeled “Cardinality type” allow one to choose between a constant cardinality or a cardinality variable. When the cardinality is a constant, its value can be set using the numeric text box “Cardinality value”. When the cardinality is a variable, its name and domain can be set using the text box “Cardinality name”, and the numeric text boxes “Cardinality min value”, and “Cardinality max value”. The button “Apply” saves the inserted data to the edge.

12.1.2 Defining Constraints on Product Characteristics

The current version of PRODPROC Modeler, being a prototype, does not have an ad-hoc interface for easily defining constraints on product characteristics, temporal constraints, and duration constraints. However, all the different types of constraints can be defined by writing them in text boxes as Prolog terms, as explained in Chapter 10.

Node constraints. Node constraints for a node for which node variables have already been defined can be created using the node attribute window. Figure 12.8 shows the attribute window of the node “Building”, for which two node variables have been defined. The red rectangle highlights the list box showing the node constraints of the node. Constraints in this list can be selected and then modified by clicking the button “Modify constraint”, or deleted by clicking the button “Delete constraint”. The button “Add constraint” opens a window titled “Constraint creation” (also shown in Figure 12.8) for node constraint definition. This window has a text box for writing the Prolog term corresponding to the

constraint to create, and an “Add” button for adding to the set of node constraints the constraint in the text box. The button “Modify constraint” of the attribute window opens a window (similar to the one for constraint creation) for constraint modification.

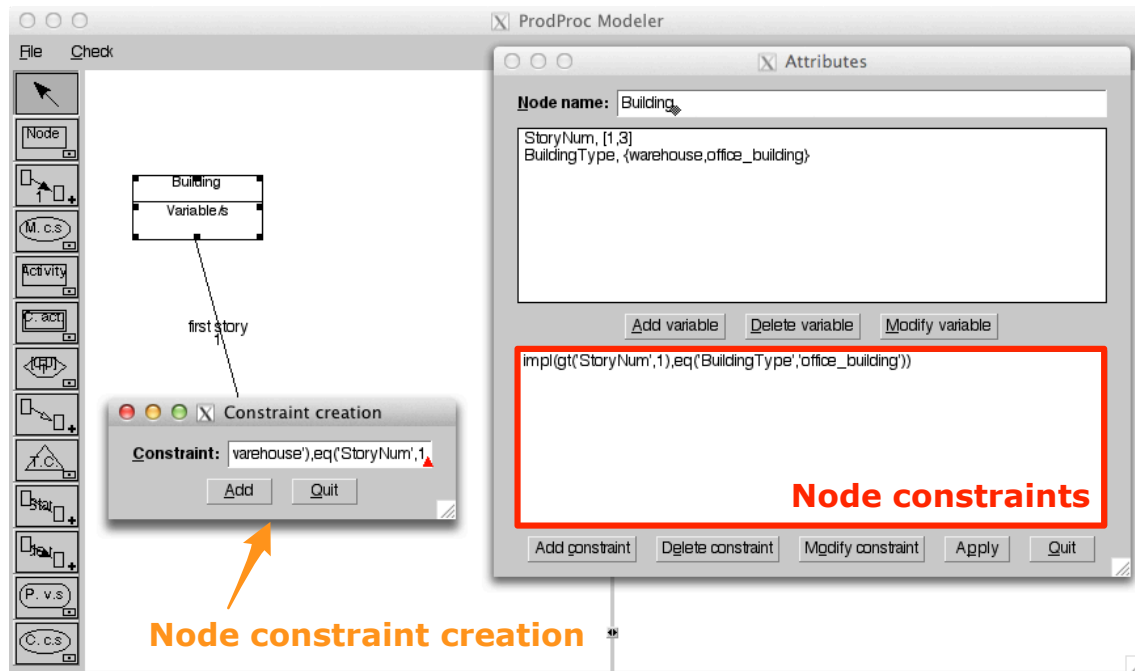


Figure 12.8: Node constraint creation.

Cardinality constraints. Cardinality constraints for an edge whose cardinality is a variable can be created using the edge attribute window. Figure 12.9 shows the attribute window of the edge “upper story”. The red rectangle highlights the list box showing the cardinality constraints of the edge. Constraints in this list can be selected and then modified by clicking the button “Modify constraint”, or deleted by clicking the button “Delete constraint”. The button “Add constraint” opens a window titled “Constraint creation” (also shown in Figure 12.9) for cardinality constraint definition. This window has a text box for writing the Prolog term corresponding to the constraint to create, and an “Add” button for adding to the set of cardinality constraints the constraint in the text box. The button “Modify constraint” of the attribute window opens a window for constraint modification. This window is similar to the one for constrain creation.

Model constraints. When the mode “Draw PROMO model constraint sets” is active, a user can draw in the “Product canvas” ellipses representing sets of model constraints. Figure 12.10 shows a set of model constraints and its attribute window. The text box “Model constraint set” can be used to assign a name to the set of model constraints. The red rectangle in Figure 12.10 highlights the list box showing model constraints in the set. Constraints in this list can be selected and then modified by clicking the button

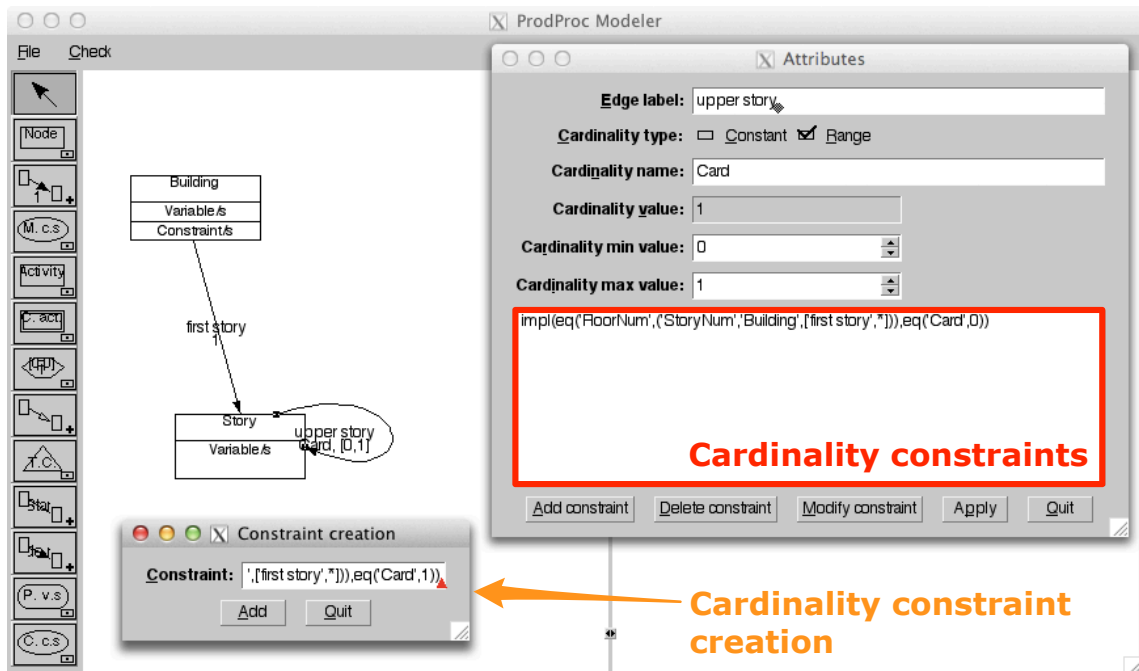


Figure 12.9: Cardinality constraint creation.

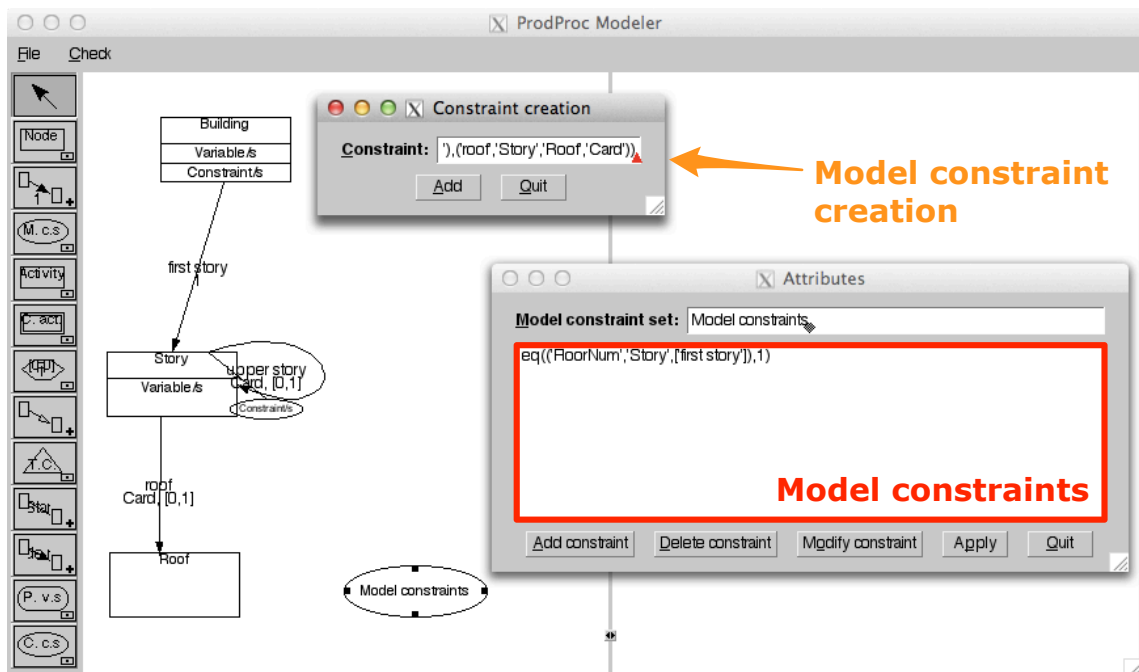


Figure 12.10: Model constraint creation.

“Modify constraint”, or deleted by clicking the button “Delete constraint”. The button “Add constraint” opens a window titled “Constraint creation” (also shown in Figure 12.10) for model constraint definition. This window has a text box for writing the Prolog term corresponding to the constraint to create, and an “Add” button for adding the constraint in the text box to the set of model constraints. The button “Modify constraint” of the attribute window opens a window for constraint modification. This window is similar to the one for constrain creation.

12.2 Creation of Process Models

In this section, we show how to use PRODPROC Modeler to create a process model, and explain how to couple it with a product model by means of coupling constraints.

Process variables. When the mode “Draw MART process variable sets” is active, a user can draw in the “Process canvas” rounded boxes representing sets of process variables. Figure 12.11 shows a set of process variables, its attribute window, and the window for process variable creation. The attribute window has a text box, labeled “Set name”, that

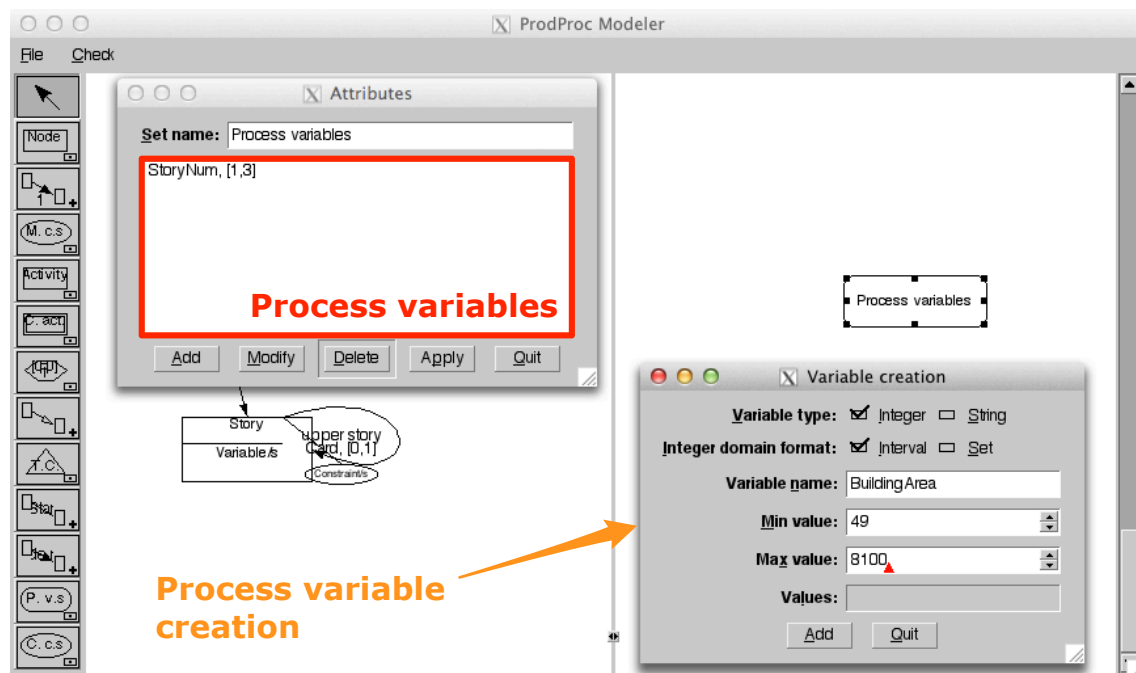


Figure 12.11: Process variables creation.

allows the user to assign a name to the set of process variables, and a list box (highlighted by the red rectangle in Figure 12.11) showing the elements of the set of process variables. The list box in Figure 12.11 shows that the set of process variables contains a variable named “StoryNum”. Variables in the list box can be selected and then modified by clicking

the button “Modify”, or deleted by clicking the button “Delete”. The button “Add” opens a window titled “Variable creation” (also shown in Figure 12.11) for process variable definition. This window has the same controls of the window for node variable creation (cf. Figure 12.6a), the “Add” button adds a new variable to the set of process variables. The button “Modify” of the attribute window opens a window for process variable modification. This window is similar to the one for process variable creation.

Activities, temporal constraints, resource constraints. Activities can be drawn in the “Process canvas” of PRODPROC Modeler when either the mode “Draw MART activities”, or the mode “Draw MART composite activities” is active. When the latter mode is active, a left-click in the “Process canvas” draws a box labeled “New activity” like the one in Figure 12.12. The attribute window of a composite activity is also shown in Fig-

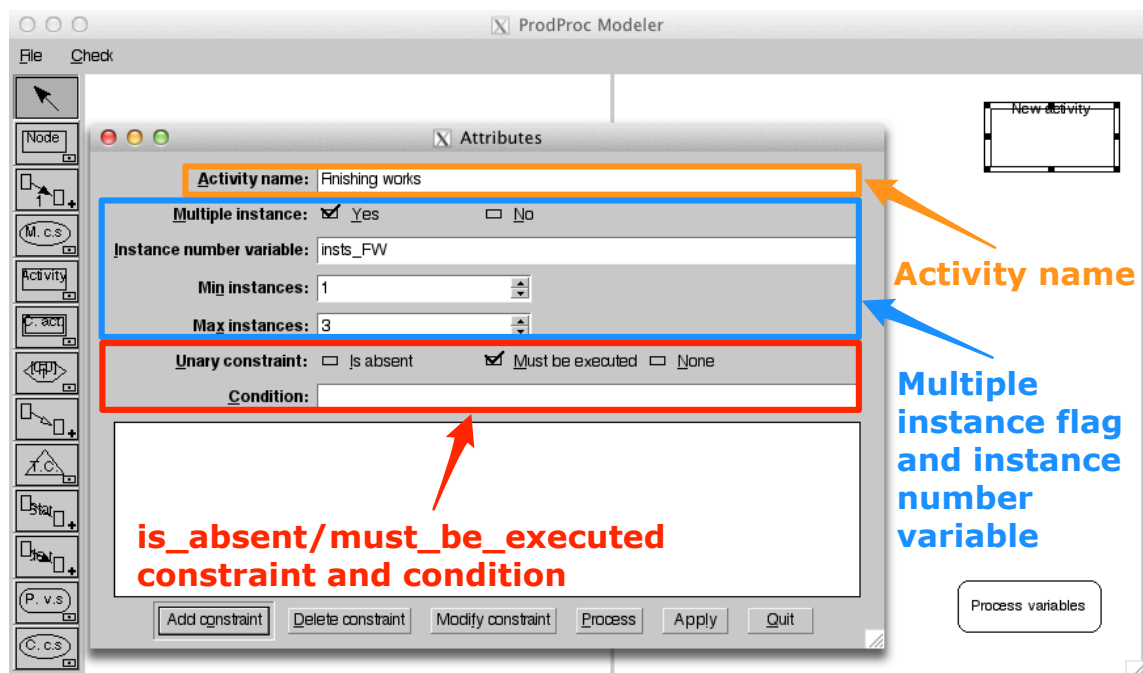


Figure 12.12: MART composite activity creation.

ure 12.12. The figure shows the definition of the activity “Finishing works” of the case study presented in Chapter 6. The orange rectangle in Figure 12.12 highlights the text box for defining the composite activity name. The blue rectangle highlights controls for multiple instance activity definitions. This controls, from top to bottom, are: two radio buttons for defining the activity as either a multiple instance activity or a single instance activity, a text box for defining the name of the instance number variable (the text box is active only when the radio button “Yes” is selected), and two numeric text boxes for defining the instance number variable domain (the numeric text boxes are active only when the radio button “Yes” is selected). Finally, the red rectangle in Figure 12.12 highlights three radio buttons and a text box. The radio buttons allow the user to either

impose an *is_absent/must_be_executed* constraint on the activity, or leave the activity not constrained by a unary temporal constraint. The text box allow the user to define a condition on the *is_absent/must_be_executed* constraint (the text box is not editable when the radio button labeled “None” is selected). Conditions can be defined using the Prolog terms *if(Cond)* (for if-conditional temporal constraints) and *iff(Cond)* (for iff-conditional temporal constraint), where *Cond* is a Prolog term representing a constraint on process variables. The button “Process” of a composite activity attribute window opens a window for the definition of the MART model associated to the composite activity. The button “Apply” saves the inserted data to the activity. The attribute windows for atomic activities is very similar to the one for composite activity, the only difference is the absence of the “Process” button.

Figure 12.13 shows the window for the definition of MART models of composite activities. The window is titled “Composite activity process”, it has a mode-selection menu

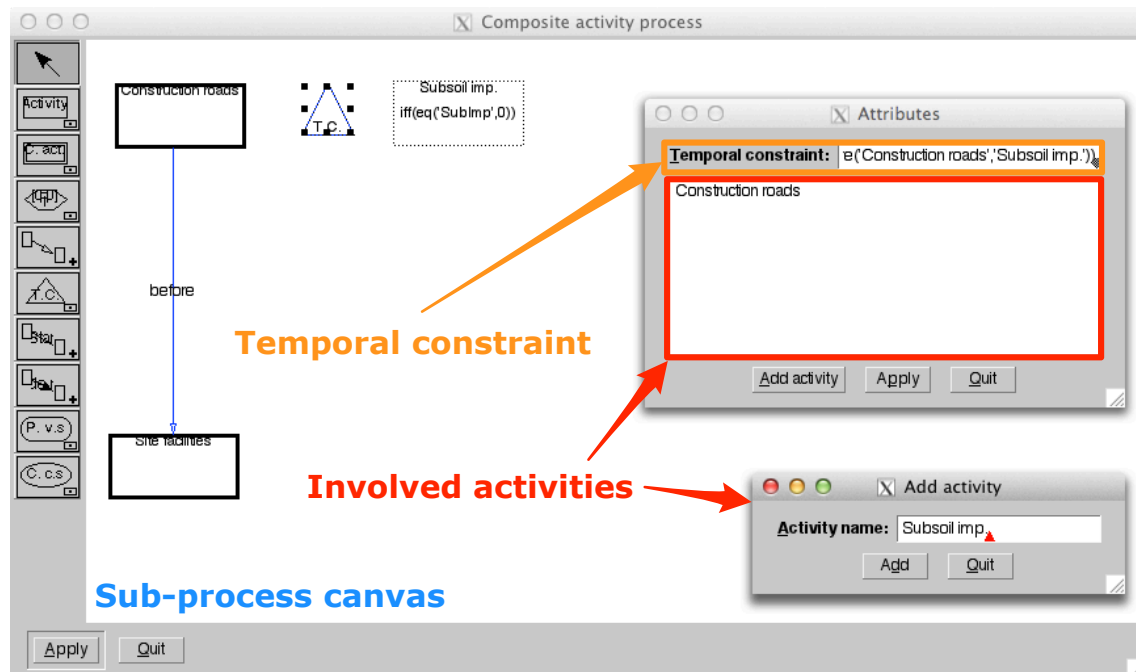


Figure 12.13: MART model of a composite activity and temporal constraint creation.

with a selector for activating the select mode, and selectors for activating the modes for MART elements creation. The window also have an area for drawing a process, denoted as “Sub-process canvas” in Figure 12.13, and a button “Apply” for saving the process to the composite activity. A “Sub-process canvas” has the same characteristics of the “Process canvas” shown in Figure 12.1. Hence, process variables and activities, when the corresponding mode is active, can be created with a left-click also in the “Sub-process canvas”. From now on, “Process canvas” will denote both the process drawing area of PRODPROC Modeler main window, and the process drawing area of a “Composite activity process” window. Figure 12.13 also shows windows for the creation of temporal constraints, some

atomic activities and an atomic temporal constraint. In particular, it shows the partial definition of the process associated to the composite activity “Preparation and development of the building site” of the case study presented in Chapter 6.

A temporal constraint can be drawn in the “Process canvas” when the mode “Draw MART temporal constraints” is active. A left-click draws a triangle like the one in Figure 12.13. The attribute windows of a temporal constraint is also shown in Figure 12.13. The orange rectangle highlights a text box for writing the Prolog term corresponding to the temporal constraint to create. The red rectangle highlights the list box showing activities involved in the temporal constraint. Each activity shown in this list will be connected to the triangle with a blue line. The button “Add activity” allows one to add an activity to the list of involved activities. In particular, it opens a window titled “Add activity” (cf. Figure 12.13) having a text box where the user can write the name of an activity, and a button “Add” to add the name inserted in the text box to the list of activities involved in the temporal constraint. The button “Apply” of the attribute window of a temporal constraint saves the inserted data to the temporal constraint.

An atomic temporal constraint between two activities can be drawn in the “Process canvas” when the mode “Draw MART atomic temporal constraints” is active. To draw an atomic temporal constraint a user has to left-click on an activity, drag the cursor to another activity, and release the left mouse button when the cursor is over the second activity. This gesture creates a blue arrow labeled “before”, representing a *before* constraint, as shown in Figure 12.13. Once such an arrow has been drawn, it is possible to change the atomic constraint it represents (i.e., its label) and to define a condition on the constraint using the atomic temporal constraint attribute window shown in Figure 12.14. The atomic constraint can be set using the drop-down list “Atomic constraint”. The text box “Condition” can be used to impose a condition on the constraint. The button “Apply” saves the inserted data to the atomic temporal constraint.



Figure 12.14: MART atomic constraint attributes.

Resources can be drawn in the “Process canvas” when the mode “Draw MART resources” is active. Resources are represented as six-sided polygons, labeled with the resource domain, initial value and name. Figure 12.15 shows a resource and its attribute window. The orange rectangle highlights the text box for the resource name definition. The red rectangle highlights the numeric text boxes for defining the resource minimum and maximum quantity. Finally, the blue rectangle highlights the numeric text box for setting the resource initial value. The button “Apply” saves the inserted data to the resource.

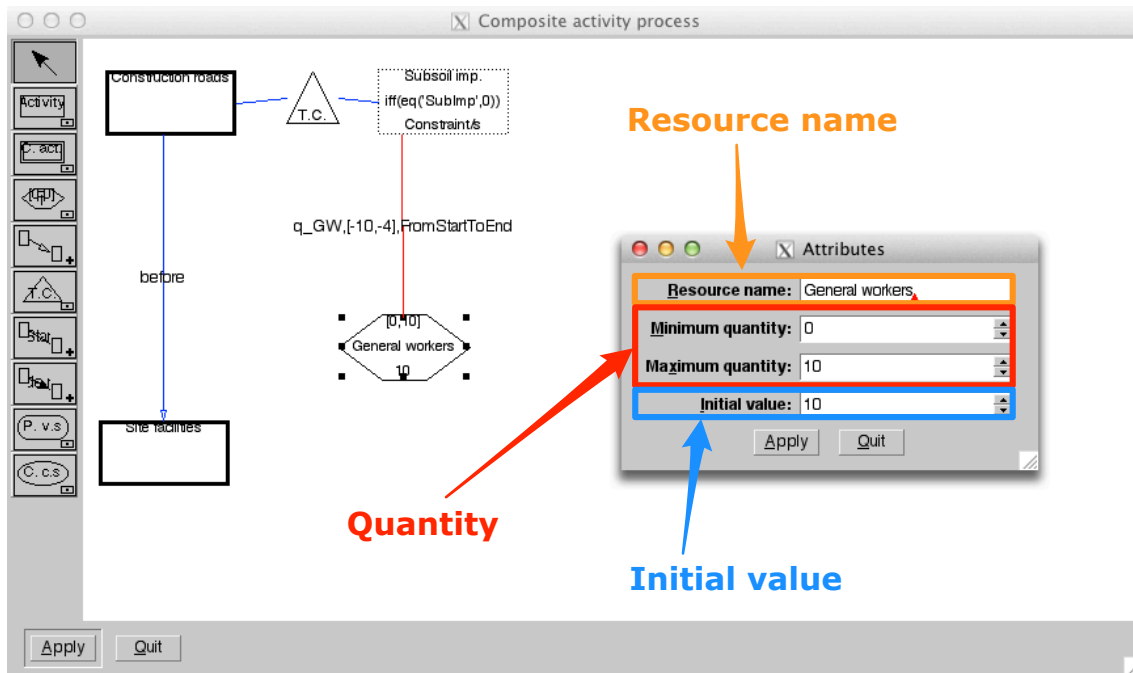


Figure 12.15: MART resource creation.

Figure 12.15 also shows a resource constraint. A resource constraint between an activity and a resource can be drawn in the “Process canvas” when the mode “Draw MART resource constraints” is active. To draw a resource constraint a user has to left-click on an activity (or a resource), drag the cursor to a resource (resp. an activity), and release the left mouse button when the cursor is over the resource (resp. the activity). This gesture creates a red line labeled with the produced/consumed resource quantity and the time extent, as shown in Figure 12.15. Once a resource constraint has been drawn, it is possible to change its parameters using the resource constraint attribute window shown in Figure 12.16. The radio buttons labeled “Quantity type” of the attribute window allows one to chose between a constant or an integer variable for representing the quantity of resource consumed/produced. The numeric text box “Quantity” allows one to set the integer quantity consumed/produced, this text box is active when the radio button “Constant” is selected. The text box “Quantity variable”, and the numeric text boxes “Minimum quantity”, and “Maximum quantity”, are active when “Quantity type” is set to “Variable”, and allow one to define the name and the domain of the variable representing the quantity of resource consumed/produced. The drop-down list “Time extent” allows one to set the time extent of the resource constraints, while the text box condition allows one to define a condition on the resource constraint. Finally, the button “Apply” saves the inserted data to the resource constraint.

The attribute window for atomic and composite activity also allows the user to define duration constraints. Figure 12.17 shows the attribute window of an atomic activity named “Subsoil imp.”. The red rectangle highlights the list box showing the activity duration

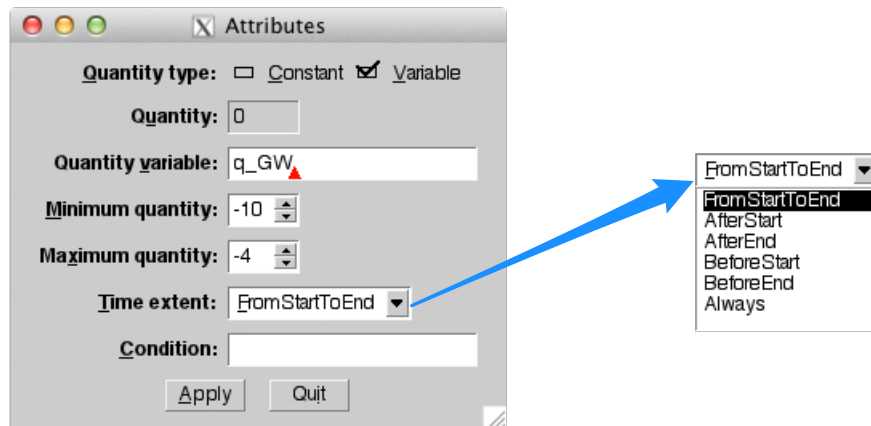


Figure 12.16: MART resource constraint creation.

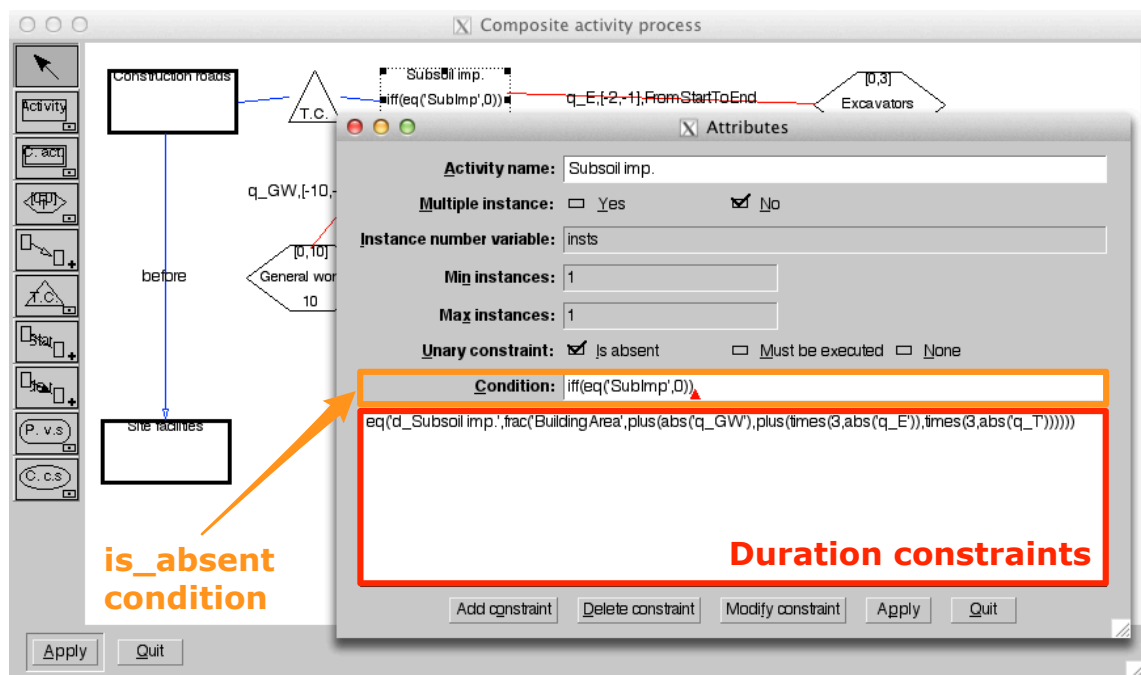


Figure 12.17: Activity duration constraints and condition on temporal constraint.

constraints. Constraints in this list can be selected and then modified by clicking the button “Modify constraint”, or deleted by clicking the button “Delete constraint”. The button “Add constraint” opens a window titled “Constraint creation” like the one for node constraint definition (cf. Figure 12.8). This window has a text box, for writing the Prolog term corresponding to the constraint to create, and an “Add” button for adding to the set of duration constraints the constraint in the text box. The button “Modify constraint” of the attribute window opens a window for constraint modification. This window is similar

to the one for constrain creation.

Figure 12.17 also shows a condition for an *is_absent* constraint (highlighted by an orange rectangle), and the graphical representation of such a constraint (i.e., a dotted box labeled with the activity name, and the *is_absent* constraint condition).

Coupling constraints. When the mode “Draw MART coupling constraint sets” is active, a user can draw in the “Process canvas” ellipses representing set of model constraints. Figure 12.18 shows a set of coupling constraints and its attribute window. The text box “Set name” can be used to assign a name to the set of coupling constraints. The red rectangle in Figure 12.18 highlights the list box showing coupling constraints in the set. Constraints in this list can be selected and then modified by clicking the button “Modify constraint”, or deleted by clicking the button “Delete constraint”. The button “Add constraint” opens a window titled “Constraint creation” (also shown in Figure 12.18) for coupling constraint definition. This window has a text box, for writing the Prolog term corresponding to the constraint to create, and an “Add” button for adding the constraint in the text box to the set of model constraints. The button “Modify constraint” of the attribute window opens a window (similar to the one for constrain creation) for constraint modification.

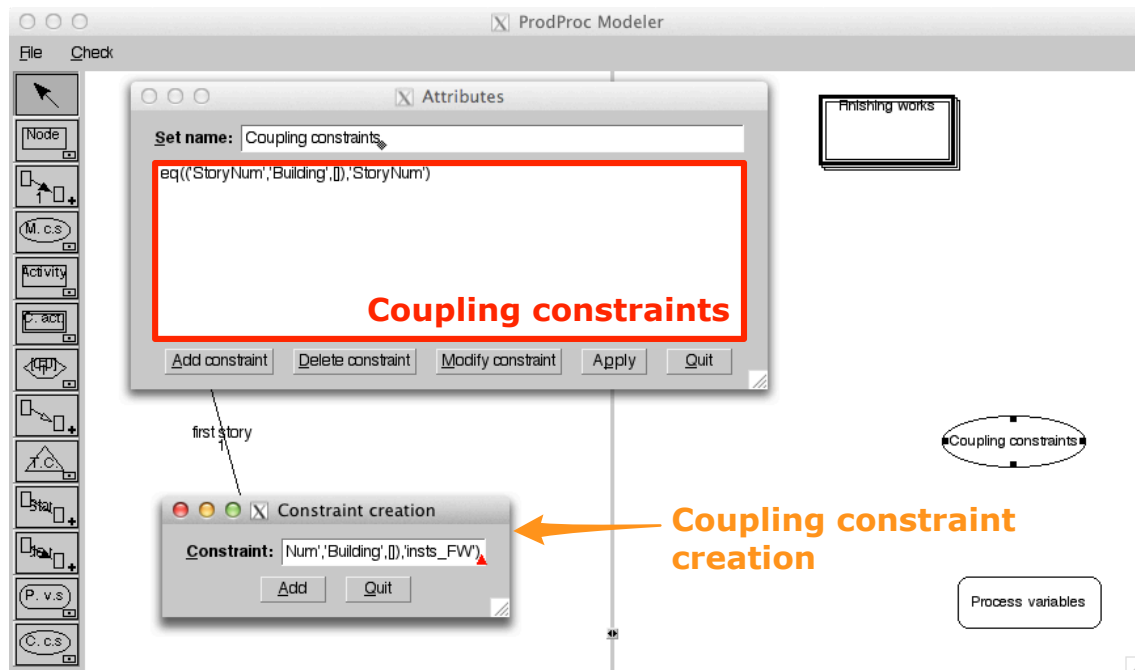
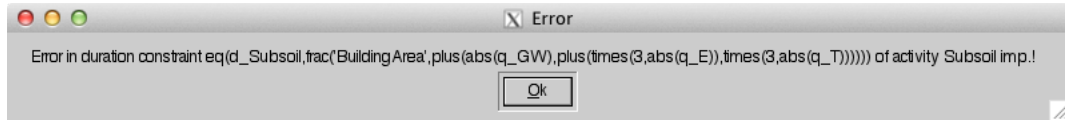


Figure 12.18: Coupling constraint creation.

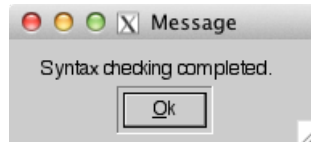
12.3 Syntax and Validity Checking

In this section, we describe the “Syntax” and “Validity” commands of the “Check” menu of PRODPROC Modeler, shown in Figure 12.2b.

The command “Syntax” starts the execution of the syntax checker on a PRODPROC model. The syntax checker verifies the syntactic correctness of a model with respect to the PRODPROC framework specifications presented in Chapter 5. Once the syntax checker execution terminates, the message in Figure 12.19b is shown. For each error discovered by the syntax checker, a message like the one in Figure 12.19a appears on screen.



(a) Syntax error.



(b) Syntax checking completed.

Figure 12.19: Syntax checking messages.

The command “Validity” opens the window titled “Validity checking” shown in Figure 12.20. The window has four numeric text boxes for setting validity checker parameters. The numeric text boxes “Min node number” and “Max node number” allows the user to set the minimum and the maximum number of node instances for product instance generation. The maximum duration of the process instance can be set using the numeric text box “Max time”. Finally, the numeric text box “Time out” allows the user to impose a limit on certain time consuming computation executed by the validity checker. In particular, the labeling of the CSP for determining a process instance, and the sequential labeling of CSPs obtained from resource constraints, are stopped after a number of seconds equal to the one given in “Time out”.

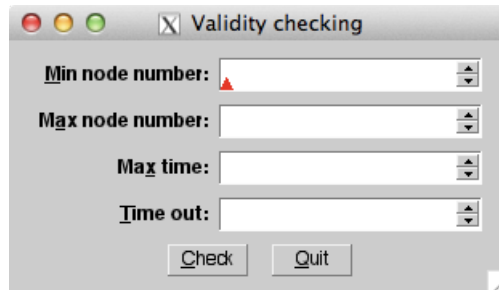


Figure 12.20: Parameters for validity checking.

The button “Check” of the window “Validity checking” starts the algorithm for the generation of a PRODPROC instance, and opens a window titled “Instance generation” like the one shown in Figure 12.21. The algorithm for instance generation is executed as a separate thread. The window “Instance generation” has an area where data on the ongoing computation are reported. The constituting elements of computed product and process instances (i.e., node instances, node variable assignments, activity instances, etc.) are listed in this area. The “Instance generation” has also a “Stop” button to kill the thread executing the instance generation algorithm. Figure 12.21 shows a fragment (i.e., the node instances, and some *has-part* relations between them) of a product instance for the building case study presented in Chapter 6. A fragment of the corresponding process instance (i.e., assignments to process variables, and an activity instance) is shown in Figure 12.22.



Figure 12.21: Validity checking result: product instance.

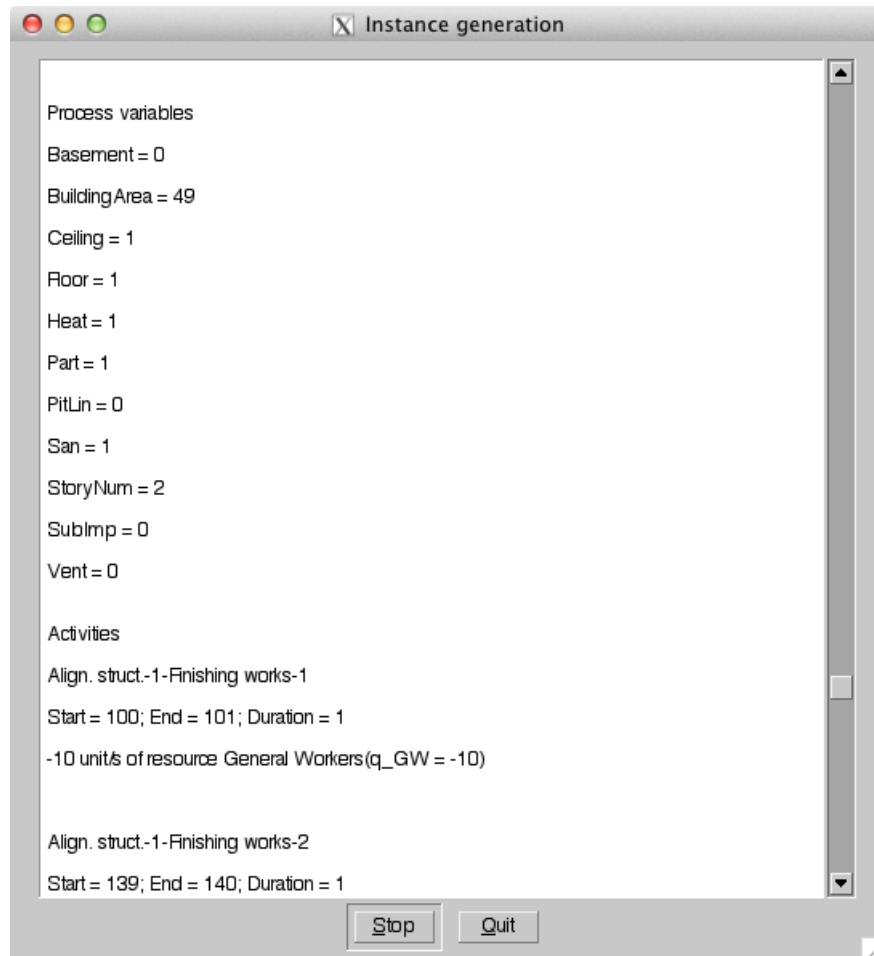


Figure 12.22: Validity checking result: process instance.

Chapter 13

Conclusions

In this dissertation we focused on the problem of product and process modeling and configuration. In particular, we pointed out the lack of a tool covering both physical and production aspects of configurable products. To overcome this absence, we proposed a declarative constraint-based framework called PRODPROC, that allows one to model a configurable products and its production process. This chapter presents a summary of the contributions towards this goal, and an outlook on some further research questions that we did not answer in this dissertation.

13.1 Summary and Main Contributions

The first part of this dissertation introduced the notion of mass customization and configuration systems, summarizing the state of the art of research about methodologies and techniques for product and process configuration, and focusing on the coupling of product with process configuration. It also presented the Constraint Programming approach to programming, and an example of product configurator based on Constraint Logic Programming.

In the second part, we developed a declarative constraint-based framework for product and production process modeling, called PRODPROC. We illustrated the distinctive features of the two languages composing PRODPROC, that is, the language PROMO for configurable product modeling, and the language MART for production process modeling, and gave an account of their behavior by means of product and process instances. In order to allow a user to easily to define product and process structures, we defined a graphical representation for most of PRODPROC modeling elements. To demonstrate the modeling capabilities of our framework, we considered a prefabricated component building and its construction process as a case study. Moreover, we outlined how it is possible to encode a product and process configuration problem into a CSP, in order to exploit Constraint Programming to implement an automatic or interactive configuration system based on PRODPROC. We also shown how a PRODPROC model can be encoded into a GCSP, suggesting a possible way to give a constraint-based semantics to PRODPROC. Finally, we compared our framework to some of the existing product configuration systems and

process modeling tools, to put in evidence its strength and limitations.

The third part of this dissertation focused on the implementation of a configuration system on top of the PRODPROC framework. The system, called PRODPROC Modeler, has been implemented using SWI Prolog and exploits Constraint Logic Programming. We first outlined a possible Prolog encoding of PRODPROC model. Then, we described an algorithm for automatic generation of PRODPROC instances, based on the CSP encoding presented in the second part of this dissertation. Finally, we described the graphical interface of PRODPROC Modeler, showing how it allows a user to model a product and its production process using the PRODPROC framework, to check the syntactic correctness of a model, and to automatically generate PRODPROC instances to verify the validity of a model.

The PRODPROC framework presents features that, to the best of our knowledge, are not present in other existing product or process modeling languages. These are, for example, product model graphs, model constraints, resource variables and resource constraints, activity duration constraints. Moreover, all these innovative features belong to a single framework that allows one to model products, their production processes, and to couple products with processes using constraints.

13.2 Future Research

PRODPROC and PRODPROC Modeler can be extended in three different directions.

Real number constraints. In the current version of PRODPROC, a sort `real` for real numbers is not included. As far as the products to model depend on mechanical specifications that define a fixed precision for numerical characteristics, one can restrict his/her attention to numbers with a fixed number of decimal digits. Integers clearly suffice for the management of these numbers. If the number of digits after the radix point is low, one can use a single integer for representing a fixed-point number. Such a solution is inappropriate for fixed-point numbers with a high number of decimal digits. In such a case, indeed, by multiplying integer numbers representing fixed-point numbers one can exceed the max-integer of the finite domain solver. To avoid this problem, one can use two integers for representing a fixed-point number, one for the integer part and one for the decimal part of the number, respectively. The only drawback of such a solution is that it requires the re-implementation of all the arithmetic operations (the first solution requires the re-implementation of division and multiplication only). To broaden the range of applications of PRODPROC, we are considering the possibility of adding the ability of dealing with real numbers to it. To this end, we had a look at existing solutions. One of them is provided by the Interval Constraint (IC) library of ECLⁱPS^e [AW06], an hybrid integer/real interval arithmetic constraint solver. Unfortunately, the IC implementation of the fundamental algorithm HC3 suffers from too many floating-point approximation errors. Another one is given by the system \mathcal{TOY} [EMFHG⁺07], that, unlike the finite domain solver of SWI Prolog, allows one to solve CSP involving both finite domain variables and real variables. However, the current version of this system seems to be still very preliminary. A further

alternative consists in the implementation of a hybrid solver that uses a double representation of finite domains (i.e., a finite set representation and an interval representation), and exploits two solvers: a finite domain solver (e.g., the CLPFD library of SWI Prolog) for finite domain constraint propagation, and an interval solver (e.g., *RealPaver* [GB06]) for dealing with mixed integer/real constraints.

Imperative features for process modeling. Imperative process modeling languages like BPMN and YAWL have among their modeling elements the so called splits and joins. In particular, both BPMN and YAWL support the following constructs: AND-join, OR-join, XOR-join, AND-split, OR-split, XOR-split. Even if in PRODPROC a user can model splits and joins by means of temporal constraints, a direct support for this constructs would make it easier to define highly structured processes. We plan to add split and join constructs to PRODPROC, and to implement them as propositional formulas on activities start and end times, and execution flags. PRODPROC could also be extended adding to it support for modeling elements such as activity looping and cancellation region.

Interactive configuration support. The configuration system we implemented on top of PRODPROC, i.e, PRODPROC Modeler, can be improved adding to it ad-hoc interfaces for the creation of product and process constraints. Moreover, it could be extend to a full-fledged configuration system supporting not only modeling (and automatic configuration), but interactive configuration too. In particular, the PCE library of SWI Prolog can be used to implement a graphical interface that allows a user to select a product, to make choices on it (i.e., to create node and activity instances, to assign values to variables, etc.), and to start an inference process. The algorithm we presented for the automatic generation of PRODPROC instances can be adapted to implement such an inference process, i.e., to encode partial configurations into CSPs and to compute consequences on user choices. The incrementality of the finite domain solver of SWI Prolog can be exploited to make the inference phase of the configuration process cheap. We can keep a CSP representation of the configuration under construction and add to it new constraint and variables whenever and update to the partial configuration takes place. Moreover, we can allow the user to add new constraint to the model under configuration, or on variables of the partial configuration, during the configuration process. To cope with dead-ends reached during the configuration process, and to allow a user to change his/her preference, an assignment revision process like the one of Morphos MCE [CRD⁺10] can be implemented.

Another aspect related to dead-ends reached during interactive configuration that can be considered is that of explanations. During the course of interactive configuration the user may encounter difficulties such as inconsistency or desirable values being eliminated due to conflicts with previous decisions. In such situations it is desirable for the interactive system to generate an explanation that focus on what has caused the problem or, even better, explains what can be done to overcome it. The QUICKXPLAIN algorithm [Jun04] and works on corrective explanations [OOF05] and representative explanations [OPFP07], can be considered as starting point to implement an explanation generation infrastructure for an interactive configuration system based on PRODPROC.

We plan to extend PRODPROC Modeler with the features listed above, to experiment it on different real-world application domains, and to compare it with commercial products, e.g., Oracle Configurator [Ora06], SAP Variant COnfigurator [BMU09], and Tacton Configuration [Tac11].

Appendix A

Syntax of PRODPROC Languages

ProdProcModel $:= \langle \textit{ProductModel}, \textit{ProcessModel}, \textit{CouplingConstrs} \rangle$

ProductModel $:= \langle \textit{ProductModelGraph}, \textit{ModelConstrs} \rangle$

ProductModelGraph $:= \langle \textit{Nodes}, \textit{Edges} \rangle$ s.t. there exists a node without entering edges

Nodes $:= \{N \mid N \text{ is a } \textit{Node}\}$

Node $:= \langle \textit{NodeName}, \textit{NodeVariables}, \textit{NodeConstrs} \rangle$

NodeName $:=$ an ASCII string

NodeVariables $:= \{V \mid V \text{ is a } \textit{Variable}\}$

Variable $:= \textit{IntVar} \mid \textit{StringVar}$

IntVar $:= \langle \textit{VarName}, \textit{IntDomain} \rangle$

StringVar $:= \langle \textit{VarName}, \textit{StringDomain} \rangle$

VarName $:=$ an ASCII string

Domain $:= \textit{IntDomain} \mid \textit{StringDomain}$

IntDomain $:= \{i_1, \dots, i_n\}$ s.t. $\forall j \in \{1, \dots, n\} \ i_j \in \mathbb{Z} \mid [\text{min}, \text{max}]$ s.t. $\text{min} \in \mathbb{Z} \wedge \text{max} \in \mathbb{Z}$

StringDomain $:= \{s_i, \dots, s_n\}$ s.t. $\forall j \in \{1, \dots, n\} \ s_j$ is an ASCII string

<i>NodeConstrs</i>	$:= \{c \mid c \text{ is a } \textit{NodeConstr}\}$
<i>NodeConstr</i>	$:=$ <i>Constraint</i> on a set of <i>VarName</i> of <i>Variable</i> of a <i>Node</i> and <i>MetaVariable</i> related to <i>Node</i> ancestors
<i>Edges</i>	$:= \{e \mid e \text{ is an } \textit{Edge}\}$
<i>Edge</i>	$:= \langle \textit{EdgeName}, \textit{Parent}, \textit{Child}, \textit{Card}, \textit{CardConstrs} \rangle$ where <i>Parent</i> and <i>Child</i> are <i>NodeName</i>
<i>EdgeName</i>	$:=$ an ASCII string
<i>Card</i>	$:= n \in \mathbb{N}^+ \mid \textit{IntVar}$
<i>CardName</i>	$:=$ an ASCII string
<i>CardConstrs</i>	$:= \{c \mid c \text{ is a } \textit{CardConstr}\}$
<i>CardConstr</i>	$:=$ <i>Constraint</i> on the <i>CardName</i> of an <i>Edge</i> , and a set of <i>VarName</i> of <i>Variable</i> of <i>Parent</i> and <i>MetaVariable</i> of <i>Parent</i> ancestors
<i>ModelConstrs</i>	$:= \{c \mid c \text{ is a } \textit{ModelConstr}\}$
<i>ModelConstr</i>	$:=$ <i>Constraint</i> on a set of <i>MetaVariable</i> \mid <i>GlobalConstraint</i> on a set of <i>MetaVariable</i> \mid <i>Constraint</i> on a set of <i>MetaCard</i>
<i>MetaVariable</i>	$:= \langle \textit{VarName}, \textit{NodeName}, \textit{MetaPath} \rangle$
<i>MetaPath</i>	$:= [] \mid [\textit{Label}] \mid [\textit{Label} \mid \textit{MetaPath}]$
<i>Label</i>	$:= \textit{EdgeName} \mid _ \mid \star$
<i>MetaCard</i>	$:= \langle \textit{EdgeName}, \textit{Parent}, \textit{Child}, \textit{CardName} \rangle$ where <i>Parent</i> and <i>Child</i> are <i>NodeName</i>

<i>ProcessModel</i>	$:= \langle Activities, ModelVariables, TempConstrs, \\ ProductConstrs, ResConstrs, DurationConstrs \rangle$
<i>Activities</i>	$:= \{A \mid A \text{ is an } Activity\}$
<i>Activity</i>	$:= AtomicActivity \mid MultInstActivity \mid CompActivity \mid MultCompoActivity$
<i>AtomicActivity</i>	$:= \langle ActivityName, t^{start}, t^{end}, d, ExecFlag \rangle$ where t^{start}, t^{end} , and d are integer decision variables, $t^{end} \geq t^{start} \geq 0$, and $d = t^{end} - t^{start}$
<i>ActivityName</i>	$:=$ an ASCII string
<i>ExecFlag</i>	$:= \langle exec, \{0,1\} \rangle$
<i>MultInstActivity</i>	$:= \langle ActivityName, t^{start}, t^{end}, d, ExecFlag, Inst \rangle$
<i>Inst</i>	$:= \langle inst, IntDomain \rangle$
<i>CompositeActivity</i>	$:= \langle ActivityName, t^{start}, t^{end}, d, ExecFlag, ProcessModel \rangle$
<i>MultCompoActivity</i>	$:= \langle ActivityName, t^{start}, t^{end}, d, ExecFlag, ProcessModel, Inst \rangle$
<i>ModelVariables</i>	$:= \{V \mid V \text{ is a } ModelVariable\}$
<i>ModelVariable</i>	$:= ProcessVariable \mid ResourceVariable$
<i>ProcessVariable</i>	$:= Variable$
<i>ResourceVariable</i>	$:= IntVar$
<i>TempConstrs</i>	$:= \{c \mid c \text{ is a } TempConstr\}$
<i>ProductConstrs</i>	$:= \{c \mid c \text{ is a } ProductConstr\}$
<i>ProductConstr</i>	$:= ActivityName \text{ produces } n \text{ } NodeName \text{ for } ActivityName \mid \\ ActivityName \text{ needs } n \text{ } NodeName \text{ from } ActivityName \text{ where } n \in \mathbb{N}^+$

<i>ResConstrs</i>	$:= \{c \mid c \text{ is a } ResConstr\}$
<i>ResConstr</i>	$:= \langle ActivityName, ResourceName, Quantity, TE \rangle \mid$ $\mid Condition \rightarrow ResConstr \mid Condition \leftrightarrow ResConstr \mid$ $\mid initialLevel(ResourceName, iv) \text{ where } iv \in \mathbb{N}$
<i>Quantity</i>	$:= q \in \mathbb{Z} \mid \langle VarName, IntDomain \rangle \text{ where } \forall i \in IntDomain. i < 0 \vee \forall i \in IntDomain. i > 0$
<i>TE</i>	$:= FromStartToEnd \mid AfterStart \mid AfterEnd \mid BeforeStart \mid BeforeEnd \mid Always$
<i>DurationConstrs</i>	$:= \{c \mid c \text{ is a } DurationConstr\}$
<i>DurationConstr</i>	$:= \langle ActivityName, Constraint \rangle \text{ with } Constraint \text{ involving } VarName \text{ of } Quantity$ and d related to $ActivityName$, and a set of <i>ProcessVariable</i> $VarName$
<i>CouplingConstrs</i>	$:= \{c \mid c \text{ is a } CouplingConstr\}$
<i>CouplingConstr</i>	$:= PrimitiveConstraint \text{ with } op \in \{=\} \text{ on a set of } MetaVariable, MetaCard,$ <i>ProcessVariable</i> $VarName$
<i>Constraint</i>	$:= PrimitiveConstraint \mid \neg Constraint \mid Constraint \vee Constraint \mid$ $\mid Constraint \wedge Constraint \mid Constraint \Rightarrow Constraint$
<i>PrimitiveConstraint</i>	$:= IntTerm \text{ op } IntTerm \text{ with } op \in \{<, \leq, =, \geq, >, \neq\} \mid$ $\mid valid_tuples(Vars, Tuples) \text{ where } Vars \text{ is a list of variables and}$ <i>Tuples</i> is a list of lists of integers $\mid StringTerm \text{ op } StringTerm \text{ with } op \in \{=, \neq\}$
<i>IntTerm</i>	$:= n \in \mathbb{Z} \mid VarName \text{ of } IntVar \mid MetaCard \mid MetaVariable \text{ for an } IntVar \mid$ $\mid IntTerm \oplus IntTerm \text{ with } \oplus \in \{+, -, *, /, mod\} \mid IntTerm \mid -IntTerm$
<i>StringTerm</i>	$:= \text{ASCII string} \mid VarName \text{ of } StringVar \mid MetaVariable \text{ for a } StringVar$
<i>GlobalConstraint</i>	$:= allDifferent(L) \text{ where } L \text{ is a list of } IntTerm \text{ or}$ $StringTerm \mid aggConstraint(f, L, op, IntTerm)$ where $f \in \{sum, avg\}$, L is a list of <i>IntTerm</i> , and $op \in \{<, \leq, =, \geq, >, \neq\}$

$TempConstr \quad := \quad AtomicTConstr \mid TempConstr \wedge TempConstr \mid$
 $\quad \quad \quad \mid TempConstr \vee TempConstr$

$AtomicTConstr \quad := \quad ActivityName \text{ before } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ after } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ meets } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ met_by } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ overlaps } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ overlapped_by } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ during } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ includes } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ starts } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ started_by } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ finishes } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ finished_by } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ equals } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ must_be_executed } \mid$
 $\quad \quad \quad \mid ActivityName \text{ is_absent } \mid$
 $\quad \quad \quad \mid ActivityName \text{ not_co_existent_with } ActivityName \mid$
 $\quad \quad \quad \mid ActivityName \text{ succeeded_by } ActivityName$

$CondTConstr \quad ::= \quad Condition \rightarrow TempConstr \mid Condition \leftrightarrow TempConstr$

$Condition \quad := \quad \text{Constraint on a set of } ProcessVariable \text{ and activity execution flags}$

Bibliography

- [AFM02] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
- [Äit91] H. Äit-Kaci. *Warren’s Abstract Machine: A Tutorial reconstruction*. MIT Press, 1991.
- [All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.
- [AMS07] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modeling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.
- [AP94] A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7:39–59, 1994.
- [APEU07] A. Azab, G. Perusi, H. ElMaraghy, and J. Urbanic. Semi-Generative Macro-Process Planning For Reconfigurable Manufacturing. In Pedro Cunha and Paul Maropoulos, editors, *Digital Enterprise Technology*, pages 251–258. Springer US, 2007.
- [Apt03] K. R. Apt. *Principle of Constraint Programming*. Cambridge University Press, 2003.
- [ASS06] F. Alizon, S. B. Shooter, and T. W. Simpson. Reuse of Manufacturing Knowledge to Facilitate Platform-Based Product Realization. *Journal of Computing and Information Science in Engineering*, 6(2):170–178, 2006.
- [AV08] M. Aldanondo and E. Vareilles. Configuration for mass customization: how to extend product configuration towards requirements and process configuration. *Journal of Intelligent Manufacturing*, 19(5):521–535, 2008.
- [AVDG08] M. Aldanondo, E. Vareilles, M. Djefel, and P. Gaborit. Towards an association of product configuration with production planning. In *Proceedings of ECAI’08 Workshop on Configuration Systems*, pages 41–46. University of Patras, 2008.
- [AW06] K.R. Apt and M.G. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.
- [BC02] N. Beldiceanu and M. Carlsson. A New Multi-resource cumulatives Constraint with Negative Heights. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP ’02,

- pages 63–79. Springer-Verlag, 2002.
- [BCM⁺03] F. Baader, D. Cavalese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [BCR10] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2010:07, SICS, 2010.
- [BDK07] J. Becker, P. Delfmann, and R. Knackstedt. Adaptive Reference Modeling: Integrating Configurative and Generic Adaptation Techniques for Information Models. In Jörg Becker and Patrick Delfmann, editors, *Reference Modeling*, pages 27–58. Physica-Verlag HD, 2007.
- [BF07] T. Blecker and G. Friedrich. *Mass Customization Information Systems in Business*. IGI Global, 2007.
- [BMU09] U. Blumöhr, M. Münch, and M. Ukalovic. *Variant Configuration with SAP*. SAP Press, 2009.
- [BOBS89] V. E. Barker, D. E. O’Connor, J. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32(3):298–318, 1989.
- [BR74] D. G. Bobrow and B. Raphael. New Programming Languages for Artificial Intelligence Research. *ACM Computing Surveys*, 6:153–174, 1974.
- [Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.
- [BS89] J. Bachant and E. Soloway. The Engineering of XCON. *Communications of the ACM*, 32(3):311–317, 1989.
- [BTRC05] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 381–390. Springer Berlin / Heidelberg, 2005.
- [BZ06] H. Bley and C. Zenner. Variant-oriented Assembly Planning. *CIRP Annals - Manufacturing Technology*, 55(1):23–28, 2006.
- [Cam11] D. Campagna. A Graphical Framework for Supporting Mass Customization. In *Proceedings of the IJCAI’11 Workshop on Configuration*, pages 1–8, 2011.
- [CF11] D. Campagna and A. Formisano. ProdProc - Product and Production Process Modeling and Configuration. In Fabio Fioravanti, editor, *Proceedings of the 26th Italian Conference on Computational Logic (CILC 2011)*, volume 810 of *CEUR Workshop Proceedings*, pages 261–279. CEUR-WS.org, 2011.
- [CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In Robert Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 162–164. Springer Berlin / Heidelberg, 2004.

- [CHE05] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [CHLS06] C. Choi, W. Harvey, J. Lee, and P. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin / Heidelberg, 2006.
- [CL94] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the 11th international conference on Logic programming*, pages 369–383. MIT Press, 1994.
- [Cle87] J. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.
- [Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, 1984.
- [Con09] Configit A/S. Configit Product Modeler. http://www.configit.com/products/configit_product_modeler.html, 2009.
- [CP09] C. Combi and R. Posenato. Controllability in Temporal Conceptual Workflow Schemata. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo Reijers, editors, *Business Process Management*, volume 5701 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin / Heidelberg, 2009.
- [CRD⁺10] D. Campagna, C. De Rosa, A. Dovier, A. Montanari, and C. Piazza. Morphos Configuration Engine: the Core of a Commercial Configuration System in CLP(FD). *Fundamenta Informaticae*, 105(1-2):105–133, 2010.
- [DHSA88] M. Dinebas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language CHIP. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 249–264, 1988.
- [EMFHG⁺07] S. Estévez-Martín, A. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, M. Sáenz-Pérez, and R. del Vado-Vírseda. A proposal for the cooperation of solvers in constraint functional logic programming. *ENTCS*, 188:37–51, 2007.
- [Fel07] A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Transactions on Engineering Management*, 54(1):41–56, 2007.
- [FFH⁺98] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.
- [FLM⁺03] E. Freuder, C. Likitvivatanavong, M. Moretti, F. Rossi, and R. Wallace. Computing explanations and implications in preference-based configurators. In Barry O’Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of *LNAI*, pages 76–92., 2003.
- [FM87] F. Frayman and S. Mittal. COSSACK: A Constraint-Based Expert System for Configuration Tasks. *Knowledge-Based Expert Systems in Engineering:*

- Planning and Design*, pages 143–166, 1987.
- [FM06] E. C. Freuder and A. K. Mackworth. *Handbook of Constraint Programming*, chapter 2. Constraint Satisfaction: An Emerging Paradigm, pages 13–27. Elsevier, 2006.
- [Fra03] D. Frankel. *Model Driven Architecture*. Wiley, 2003.
- [Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, 1978.
- [GAJV07] F. Gottschalk, W. Aalst, and M. Jansen-Vullers. Configurable Process Models — A Foundational Approach. In Jörg Becker and Patrick Delfmann, editors, *Reference Modeling*, pages 59–77. Physica-Verlag HD, 2007.
- [Gas74] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974.
- [GB06] L. Granvilliers and F. Benhamou. Realpaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156, 2006.
- [GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
- [Gru92] T. R. Gruber. Ontolingua: A Mechanism to Support Portable Ontologies. Technical Report KSL 91-66, Stanford University, 1992.
- [HBR10] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the Provop approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(6-7):519–546, 2010.
- [HD93] P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.
- [HE79] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international Joint Conference on Artificial Intelligence - Volume 1*, pages 356–364. Morgan Kaufmann Publishers Inc., 1979.
- [Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [HSJ⁺04] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Moller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems*, pages 131–138. 2004.
- [Hyv89] E. Hyvönen. Constraint reasoning based on interval arithmetic. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 193–199, Detroit, 1989.
- [ILO03] ILOG S.A. *ILOG Solver 6.0 User’s Manual*, 2003.

- [Jen04] R. M. Jensen. CLab: a C++ library for fast backtrack-free interactive product configuration. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04)*, 2004.
- [JH98] W. E. Juengst and M. Heinrich. Using resource balancing to configure modular systems. *IEEE Intelligent Systems*, 13(4):50–58, 1998.
- [JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [JSS07] R. J. Jianxin, T. Simpson, and Z. Siddique. Product family design and platform-based product development: a state-of-the-art review. *Journal of Intelligent Manufacturing*, 18:5–29, 2007.
- [Jun03] U. Junker. The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In *Proceedings of the IJCAI’03 Workshop on Configuration*, pages 13–20. 2003.
- [Jun04] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artificial intelligence, AAAI’04*, pages 167–172. AAAI Press, 2004.
- [Jun06] U. Junker. *Handbook of Constraint Programming*, chapter 24. Configuration, pages 837–873. Elsevier, 2006.
- [JZP04] J. R. Jiao, L. Zhang, and K. Prasanna. Process Variety Modeling for Process Configuration in Mass Customization: An Approach Based on Object-Oriented Petri Nets with Changeable Structures. *International Journal of Flexible Manufacturing Systems*, 16:335–361, 2004.
- [KCH⁺90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [Kol00] R. Kolisch. Integration of assembly and fabrication for make-to-order production. *International Journal of Production Economics*, 68(3):287–306, 2000.
- [KSJ⁺98] K. Kyo, K. Sajoong, L. Jaejoon, K. Kijoo, S. Euseob, and H. Moonhang. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
- [KWCJ04] B. Kulvatunyou, R. A. Wysk, H. Cho, and A. Jones. Integration framework of process planning based on resource independent operation summary to support collaborative manufacturing. *International Journal of Computer Integrated Manufacturing*, 17(5):377–393, 2004.
- [Lab03] P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artificial Intelligence*, 143:151–188, 2003.
- [LaR09] M. La Rosa. *Managing Variability in Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2009.
- [LDtHM11] M. La Rosa, M. Dumas, A. H. M. ter Hofstede, and J. Mendling. Configurable multi-perspective business process models. *Information Systems*,

- 36:313–340, April 2011.
- [Lho93] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI 1993)*, pages 232–238, 1993.
- [LS06] C. Lecoutre and R. Szymanek. Generalized Arc Consistency for Positive Table Constraints. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer Berlin / Heidelberg, 2006.
- [Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [MAMS05] V. Myllärniemi, T. Asikainen, T. Männistö, and T. Soininen. Kumbang configurator - a configurator tool for software product families. In *Proceedings of the IJCAI’05 Workshop on Configuration*, pages 51–56. 2005.
- [Mar91] J. P. Martins. The truth, the whole truth, and nothing but the truth: An indexed bibliography to the literature of truth maintenance systems. *AI Magazine*, 11(5):7–25, 1991.
- [McD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.
- [MF87] S. Mittal and F. Frayman. Making Partial Choices in Constraint Reasoning Problems. *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI)*, pages 631–636, 1987.
- [MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1395–1401. 1989.
- [MF90] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, pages 25–32. 1990.
- [MFG00] M. T. Martinez, J. Favrel, and P. Ghodous. Product Family Manufacturing Plan Generation and Classification. *Concurrent Engineering*, 8(1):12–23, 2000.
- [MH02] M. Milano and W. J. van Hoeve. Reduced Cost-Based Ranking for Generating Promising Subproblems. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, CP ’02*, pages 1–16. Springer-Verlag, 2002.
- [MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, 1988.
- [Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
- [MSW06] K. Marriott, P. J. Stuckey, and M. Wallace. *Handbook of Constraint Programming*, chapter 12. Constraint Logic Programming, pages 409–452. Elsevier, 2006.
- [MTS09] W. Mayer, R. Thiagarajan, and M. Stumptner. Service composition as generative constraint satisfaction. In *Proceedings of the 2009 IEEE Int.*

- Conference on Web Services, ICWS'09*, pages 888–895. IEEE Computer Society, 2009.
- [MW98] D. L. McGuinness and J. R. Wright. Conceptual modelling for configuration: A description logic based approach. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):333–344, 1998.
- [NBJT09] A. H. Nørgaard, M. R. Boysen, R. M. Jensen, and P. Tiedemann. Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration. In *Proceedings of the IJCAI'09 Workshop on Configuration*, 2009.
- [NK06] J. Nielsen and F. Kimura. A Resource Capability Model to Support Product Family Analysis. *JSME International Journal Series C Mechanical Systems, Machine Elements and Manufacturing*, 49(2):568–575, 2006.
- [NPW08] M. Nica, B. Peischl, and F. Wotawa. A Generative Constraint Model for Optimizing Software Deployment. In *Proceedings of the ECAI'08 Workshop on Configuration Systems*, pages 13–18, 2008.
- [Obj10] Object Management Group. *OMG Systems Modeling Language (OMG SysML) Version 1.2*, June 2010.
- [OOF05] B. O’Callaghan, B. O’Sullivan, and E. C. Freuder. Generating Corrective Explanations for Interactive Constraint Satisfaction. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 445–459. Springer Berlin / Heidelberg, 2005.
- [OPFP07] B. O’Sullivan, A. Papadopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. In *Proceedings of the 22nd national conference on Artificial Intelligence - Volume 1*, pages 323–328. AAAI Press, 2007.
- [Ora06] Oracle Corporation. Oracle configurator. <http://www.oracle.com/us/products/applications/ebusiness/scm/051314.html>, 2006.
- [OV90] W. Older and A. Vellino. Extending Prolog with constraint arithmetic on real intervals. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 14.1.1–14.1.4, 1990.
- [OV93] W. Older and A. Vellino. Constraint arithmetic on real intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–195. MIT Press, 1993.
- [PD99] B.J. Pine and S. Davis. *Mass customization: The new frontier in business competition*. Harvard Business School Press, 1999.
- [PSvdA07] M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full support for loosely-structured processes. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 287–287, 2007.
- [PSW00] P. Prosser, K. Stergiou, and T. Walsh. Singleton Consistencies. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP '02*, pages 353–368. Springer-Verlag, 2000.

- [Pug98] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th national/10th conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366. American Association for Artificial Intelligence, 1998.
- [QGLOB05] C.-G. Quimper, A. Golynski, A. López-Ortiz, and P. Beek. An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint. *Constraints*, 10:115–135, 2005.
- [RBW06] F. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
- [Rég96] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th national conference on Artificial intelligence*, volume 1 of *AAAI'96*, pages 209–215. AAAI Press, 1996.
- [RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [RK08] M. Razavian and R. Khosravi. Modeling Variability in Business Process Models Using UML. In IEEE Computer Society, editor, *Proceedings of the 5th International Conference on Information Technology: New Generations (ITGN'08)*, pages 82–87, 2008.
- [RR98] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models - Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.
- [RvdA07] M. Rosemann and W.M.P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.
- [Sch99] C. Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.
- [Sch01] K. Schierholt. Process configuration: Combining the principles of product configuration and process planning. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 15:411–424, 2001.
- [SF94] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, pages 10–20. Springer-Verlag, 1994.
- [SF96] D. Sabin and E. C. Freuder. Configuration as Composite Constraint Satisfaction. In George F. Luger, editor, *Proc. of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996.
- [SFH98] M. Stumptner, G. E. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:307–320, 1998.
- [SGN99] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 419–433. Springer-Verlag, 1999.

- [SH93] M. Stumptner and A. Haselböck. A Generative Constraint Formalism for Configuration Problems. In *Advances in Artificial Intelligence*, volume 728 of *LNCS*, pages 302–313. Springer Berlin / Heidelberg, 1993.
- [SKK03] C. Sinz, A. Kaiser, and W. Kuchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.
- [Smo95] G. Smolka. The Oz Programming model. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 5–8. Springer Berlin / Heidelberg, 1995.
- [SNS02] P. Simons, H. Niemelä, and T. Soinen. Extending and implementing the stable models semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
- [Som10] Hans Sommer. *Project Management for Building Construction*. Springer, 2010.
- [SP06] A. Schnieders and F. Puhlmann. Variability mechanisms in e-business process families. *Lecture Notes in Informatics*, 85:583–601, 2006.
- [STMS98] T. Soinen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:357–372, 1998.
- [SW98] D. Sabin and R. Weigel. Product Configuration Frameworks - A Survey. *Intelligent Systems and their Applications, IEEE*, 13(4):42–49, 1998.
- [Swe08] Swedish Institute of Computer Science, Intelligent Systems Laboratory. *SICStus Prolog User's Manual*, 4.0.3 edition, May 2008.
- [Tac11] Tacton Systems AB. Tacton configurator. <http://www.tacton.com>, 2011.
- [TCC05] Hwai-En Tseng, Chien-Chen Chang, and Shu-Hsuan Chang. Applying case-based reasoning for product configuration in mass customization environments. *Expert Systems with Applications*, 29(4):913–925, 2005.
- [tHvdAAR10] A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation - YAWL and its Support Environment*. Springer, 2010.
- [TO02] E. S. Thorsteinsson and G. Ottosson. Linear relaxations and reduced-cost based propagation of continuous variable subscripts. *Annals of Operations Research*, 115:15–29, 2002.
- [Uni11] University of Amsterdam, Human-Computer Studies. *SWI-Prolog 5.11 Reference Manual*, September 2011.
- [vdA99] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
- [vdMA04] E. R. van der Meer and H. R. Andersen. BDD-based Recursive and Conditional Modular Interactive Product Configuration. In *Proc. of Workshop on CSP Techniques with Immediate Application (CP04)*, pages 112–126, 2004.
- [vdMWA06] E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *Proceedings of the 2006 ACM symposium on Applied computing, SAC '06*, pages 409–414, New York, NY, USA, 2006. ACM.

- [vH01] W. J. van Hoeve. The alldifferent constraint: a survey. In *Proceedings of 6th Annual Workshop of the ERCIM Working Group on Constraints*, 2001.
- [War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, Artificial Intelligence Center, October 1983.
- [Wes07] M. Weske. *Business Process Management - Concepts, Languages, Architectures*. Springer, 2007.
- [WK03] J. Warmer and A. Kleppe. *The Object Constraint Language 2.0*. Addison-Wesley, 2003.
- [WM08] S. A. White and D. Miers. *BPMN modeling and reference guide: understanding and using BPMN*. Lighthouse Point, 2008.
- [WMT96] J. C. Wortmann, D. R. Muntslag, and P. J. M. Timmermans. *Customer-driven Manufacturing*. Chapman and Hall, 1996.
- [ZDV⁺08] L.Y. Zheng, H.F. Dong, P. Vichare, A. Nassehi, and S.T. Newman. Systematic modeling and reusing of process knowledge for rapid process configuration. *Robotics and Computer-Integrated Manufacturing*, 24(6):763–772, 2008.
- [ZJSF08] M. Zanker, D. Jannach, M. Silaghi, and G. Friedrich. A distributed generative csp framework for multi-site product configuration. In Matthias Klusch, Michal Pechoucek, and Axel Polleres, editors, *Cooperative Information Agents XII*, volume 5180 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin / Heidelberg, 2008.