A Framework for Constraint-Programming based Configuration

Queva, Matthieu Stéphane Benoit

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

A Framework for Constraint-Programming based Configuration

Matthieu Quéva

Kongens Lyngby 2011
IMM-PHD-2011-260

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Product configuration systems play an important role in the development of Mass Customisation, allowing the companies to reduce their costs while offering highly customised products. Such systems are often based on a configuration model, representing the product knowledge necessary to perform the configuration task. Several challenges arise when dealing with product configuration. One of those issues concerns how to model a configurable product family, i.e. how to represent the different types of configuration knowledge and their interactions. Another challenge is to provide adequate formalisms and efficient algorithms to solve the dependencies of the models at runtime.

In this dissertation, we present a constraint-based framework for configuration. The design of this framework is partly based on a study of product configuration requirements as well as a comparison of several general modelling languages. We then develop ProCoLa, a configuration-specific modelling language based on a conceptual framework that synthesizes, unifies and extends several approaches to modelling configuration in different design disciplines, e.g. physical products, software or services. A rigorous formalisation of the ProCoLa language is given and used to verify and analyse the configuration models.

Another goal of this dissertation is to describe the semantics of ProCoLa by providing a translation to a Constraint Satisfaction Problem (CSP) representation. For that purpose, several CSP formalisms are discussed and a new algorithm DnSTR is developed in order to solve the dynamic addition and retraction of table constraints at runtime. Finally, we present and evaluate a prototype implementation of ProCoLa and the configuration framework, including the integration in a development environment, tool support and interaction with UML, databases and spreadsheet applications.

Resumé

Produktkonfigurationssystemer har stort indflydelse på udviklingen af mass customization, fordi deres brug kan resultere i at virksomhederne reducerer omkostninger og samtidigt kan tilbyde specielt tilpassede produkter. Disse systemer baseres ofte på et konfigurationsmodel, der repræsenterer viden om produktet, som er nødvendigt til at gennemføre konfigurationsprocessen. Produktkonfiguration er en kompleks process med mange udfordringer, blandt andet hvordan man modellerer konfigurerbare produktfamilier, det vil sige, hvordan man repræsenterer forskellige slags viden om produktet og deres vekselvirkning.

Et andet problem er udviklingen af passende formalismer samt virksomme algoritmer til at løse modelafhængigheder. I denne afhandling præsenteres et constraint-baseret rammeværk for konfiguration. Rammeværkets struktur er delvist baseret på en studie af krav til produktkonfigurering samt en samligning af flere produktmodelleringssprog. Baseret på det udvikles sproget ProCoLa, et konfigureringsspecifik modelleringsprog baseret på et abstrakt rammeværk som kombinerer og udvider forskellige tilgang til konfiguration, både af produkter, software, og services. Konfigurationsmodellerne er verificeret og analyseret baseret på semantikken af ProCoLa.

Derudover beskriver afhandling semantikken af ProCoLa med hjælp af en oversættelse til et Constraint Satisfaction Problem (CSP). Baseret på en diskussion af forskellige tilgang til løsning af CSPs, en ny algoritme DnSTR er udviklet, som understøtter dynamisk tilføjelse og fjernelse af tabelle-constraints. Afsluttende præsenteres og evalueres en prototype af ProCoLa og konfigurationsrammeværket, samt deres integrations i en udviklingsomgivelse, værktøjunderstøttelse og interaktion med UML, samt databaser og regnearkprogrammer.

Preface

This thesis was prepared at the department of DTU Informatics, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science.

The PhD study has been carried out in the period of March 2008 to May 2011 under the supervision of Associate Professor Christian W. Probst, as well as Per Vikkelsøe (until August 2009) and Laurent Ricci (since September 2009) from Microsoft Development Center Copenhagen. This PhD project has been conducted in the context of the Industrial PhD programme from the Danish Ministry of Science, Technology and Innovation.

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. A part of the scientific work in this thesis is based on our published work in [75, 78, 79] with my supervisors as co-authors. Another part is based on our work that has been done in collaboration with Tomi Männistö from Aalto University, Finland, including published work [77] and a journal paper in preparation [76].

Lyngby, May 2011

Matthieu Quéva

Acknowledgements

First, my thanks go to Christian Probst and Jens Clausen, my supervisors at DTU, as well as Per Vikkelsøe and Laurent Ricci, who was kind enough to take on the supervision of the project after Per left the company. I extend those thanks to Lars Hvam, who helped me see the industry-side of configuration.

I would also like to thank Tomi Männistö for his contribution to the modelling part of the dissertation as well as Mikko Raatikainen and Juha Tiihonen for their help and warm welcome during my stay at Aalto University.

I wish to thank current and former members of the Language-Based Technology group at DTU: Alejandro Hernandez, Carroline Ramil, Ender Yuksel, Eva Bing, Fan Yang, Flemming Nielson, Fuyuan Zhang, Han Gao, Hanne Riis Nielson, Henrik Pilegaard, Jose Quaresma, Lijun Zhang, Marian Adler, Michael Smith, Michal Terepata, Nataliya Skrypnyuk, Piotr Filipiuk, Sebastian Nanz, Sebastian Mödersheim, Ye Zhang for creating a friendly and stimulating working environment.

I am also indebted to all my colleagues at Microsoft, including the ones that work in the Product Configuration group: Alexey Ovsyannikov, Andre Lamego, Brian Elgaard Bennett, Dennis Conrad, Lars Frandsen, Najimi Sebghatullah and Sverre Thune, that helped me integrate within the company and gave me inspiration for my research.

Finally, special thanks to Julie Chambon and my family, who supported me when I went through tough times during these three years, and without whom it would probably never have been possible to finish this project. A last thank you to all my friends in Denmark and elsewhere, who keep entertaining me in their own special way every day of my life.

Contents

Summary i

Resumé iii

Preface v

Acknowledgements vii

1 Introduction 1
- 1.1 Mass Customisation . . . 2
- 1.2 Product configuration . . . 4
- 1.3 Outline of the Thesis and Overview of the Contribution . . . 5

I Modelling in Product Configuration 7

2 Setting the Scene 9
- 2.1 Background . . . 10
- 2.2 Case Study . . . 11
- 2.3 Modelling Requirements for Product Configuration . . . 14

3 General Purpose Modelling Languages 21
- 3.1 UML and OCL . . . 22
- 3.2 SysML . . . 32
- 3.3 EXPRESS/STEP (ISO 10303) . . . 38
- 3.4 Comparison and Conclusion . . . 43

II Heterogeneous Products 47

4 Research Problem 49
4.1 Background . . . 50
4.2 Research Questions . . . 55

5 Conceptual Modelling Framework 59
5.1 Overview of the Approach . . . 59
5.2 Description of the Views . . . 63
5.3 Dependencies Within and Between Views . . . 74
5.4 Feature Views Hierarchy . . . 78
5.5 Discussion . . . 81
5.6 Comparison with related work . . . 85

6 Framework Implementation 89
6.1 The ProCoLa Modelling Language . . . 89
6.2 Formalism Definition . . . 95
6.3 Formalising ProCoLa . . . 109
6.4 Analysing ProCoLa Models . . . 122
6.5 Summary . . . 136

III Constraint Solving 139

7 Constraint Satisfaction Problems and Configuration 141
7.1 Classic CSP . . . 142
7.2 Extensions to CSP . . . 145
7.3 CSP with Non-Binary Constraints . . . 147

8 Non-binary Dynamic CSP with Simple Tabular Reduction 153
8.1 The DnSTR Algorithm . . . 154
8.2 Analysis . . . 162
8.3 Experimental Results . . . 165

9 CSP Formalisation of the Configuration Framework 167
9.1 CSP Formalism Chosen . . . 167
9.2 CSP Semantics of the Model . . . 168
9.3 Discussion . . . 180

IV Prototype and Evaluation 183

10 Prototype Implementation of the Framework 185
10.1 Language Integration . . . 185
10.2 Tool Support for Modelling . . . 186

10.3 Runtime Implementation 190
10.4 Debugging 191
10.5 Summary and Discussion 193

11 Framework Evaluation **197**
11.1 Benchmark models 198
11.2 Results 200

12 Conclusion **205**
12.1 Further Work 206
12.2 Contributions and Concluding Remarks 207

A Case Study in EXPRESS **209**

B Formalisation of Structure and Realisation Views **213**

Chapter 1

Introduction

There has been an important need for companies to reduce their costs while proposing highly customised products. Today's customers demand indeed products with lower prices, higher quality and faster delivery, but they also want products customised to match their unique needs. To meet these demands, manufacturers have to adapt their business model to *mass customisation* [73], allowing customers to order customised products, often choosing among hundreds of product features and options, for a competitive price.

However, mass customisation can result in an expansion of the specification process, causing iterations to be developed in the process, and information technologies like product configuration systems are essential to implement this new concept.

Product configuration has received a lot of interest for these past years. However, manufactured products are getting more and more complex, and techniques like product configuration need to evolve all the time in order to meet the requirements for managing these products. The main focus of this thesis is to analyse modelling and constraint-based solving methods for product configuration, and design and implement a framework for creating and configuring modern product models.

1.1 Mass Customisation

We start this dissertation by introducing mass customisation and the need for product configuration systems. The main issue with mass customisation is that it can result in an expansion of the specification process, causing iterations to be developed in the process. This process represents the tasks that are done on an individual order before production, and that defines the product to be manufactured (Figure 1.1).

The concept of mass customisation thus differs from previous industrial processes (Figure 1.2) such as:

- *Mass production* consists in producing a large amount of the same standard items, with no option to customise them. This type of process was popularised by Henry Ford, who was describing it by:

  "*Any customer can have a car painted in any colour he wants, as long as it is* ***black****.*"

- *One-of-a-kind production* consists in producing a small amount of items, but with a wide panel of different options to customise these few items. This type of process is usually used when producing large, highly complex and specific products.

- *Small series production* consists in manufacturing customised products in small series. Companies working with this type of process do usually not produce as much as in mass production, and neither are they customising their products as in one-of-a-kind production.

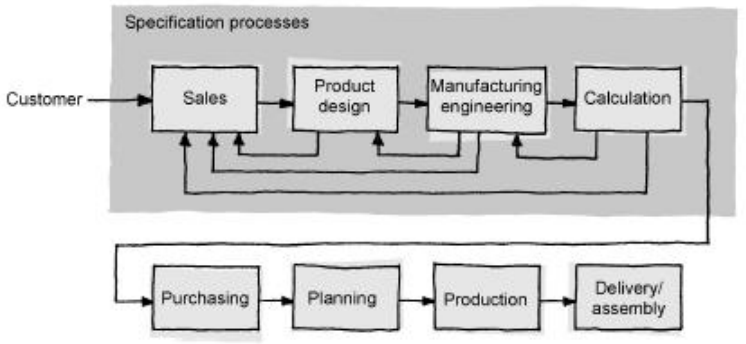


Figure 1.1: Tasks in a typical specification process [51]

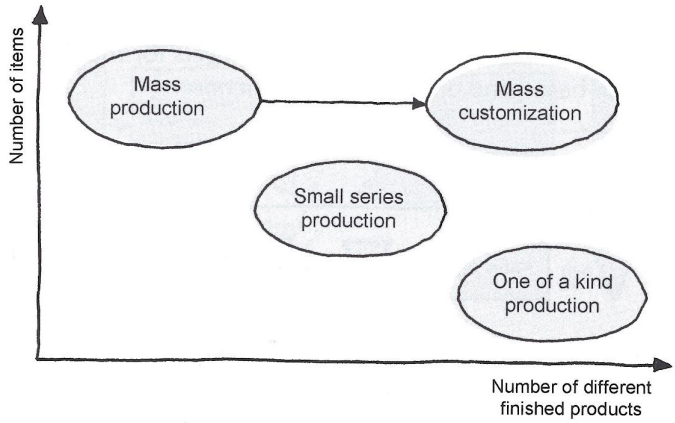


Figure 1.2: Three main types of processes [51]

To successfully implement mass customisation, manufacturers have to overcome three major challenges ([109]):

1. *Lead time*: producing a custom configuration for each product becomes a highly complex and time-consuming task. As the number of parts increases, a simple product can easily end up generating thousands of product variations. This can affect the specification process, increasing the lead time in a significant proportion.

2. *Quality assurance*: producing a significant amount of product variations involving hundreds of configurable parts increases the possibility of making errors in the process. This can create major schedule slips and can lead to costly unplanned iterations in the process.

3. *Expertise*: being able to configure complex products requires a comprehensive product knowledge and expertise from the engineers that are in charge of the configuration process. This results in a need for substantial training, which may even be repeated along the years, as technical changes occur frequently.

These issues can make the implementation of mass customisation very challenging for the companies, or even become barriers. In order to improve this implementation, it is important to develop significant information technology capabilities. One of these technologies is a *product configuration system.*

1.2 Product configuration

The *product configuration problem* is defined by Junker in [85] as characterised by two constituents:

1. A *catalogue* which describes the generic components in terms of their functional and technical properties and the relationship between both.
2. *User requirements* and *user preferences* about the functional characteristics of the desired configuration.

and the configuration task, which consists in finding the following answer:

1. One or more configurations that satisfy all requirements and that optimise the preferences if those requirements are consistent.
2. An explanation of failure in the other case.

Product configuration is applied during the specification processes, as can be seen in Figure 1.3. It can be implemented through the use of a *product configurator*: a software tool that captures the customer's requirements as input and automatically generates customised product designs matching the customer's specific needs, based on predefined design constraints.

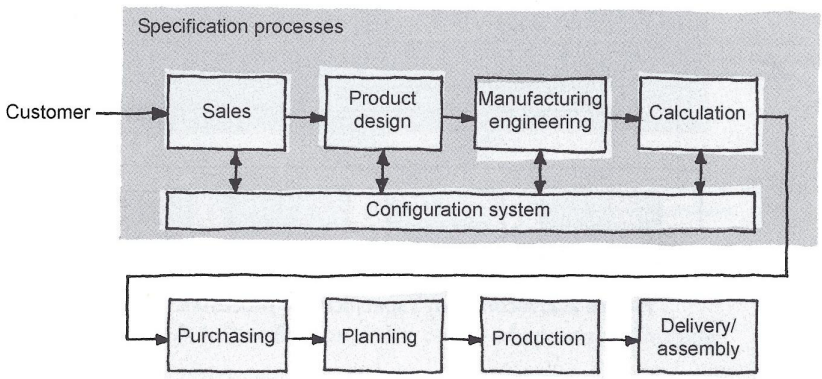


Figure 1.3: Specification process with configuration [51]

There are actually two main issues in the configuration problem:

1. The first one concerns the modelling time, i.e., the time where a design engineer will define a model for the product. At this time, there is a need to find efficient and easy-to-use ways to express the product knowledge, i.e., all the characteristics of the product, from the definition of the attributes, the different subparts, or the constraints applying to the various combinations possible. The more complex the product is, the more important this part is.
2. The second issue concerns how to solve, at configuration time, the constraints expressed at modelling time. At this time, a sales person will assign values to the different attributes of the product, so there is a need for an efficient solving engine, that should be able to solve *in real-time* the constraints defined previously.

1.3 Outline of the Thesis and Overview of the Contribution

We present in this Section the organisation of this dissertation. The thesis is separated into four *parts*, each of them composed by several *chapters*.

In **Part I**, we focus on requirements for the configuration of manufactured products. We introduce an example case study and use it to describe and review several well-known general-purpose modelling languages in the context of configuration.

We consider in **Part II** product families that are not restricted to manufacturing parts, but also contain software and service-related components. Most of the previous approaches in configuration research only consider each specific type of product knowledge separately. However, many products nowadays are heterogeneous, i.e., different design disciplines are taken into account within the same products. We thus define a conceptual framework for modelling heterogeneous product families, and provide clear semantics using a new modelling language called ProCoLa. ProCoLa is then formalised and several model analyses are presented.

Part III focuses on identifying and analysing the relevant constraint programming techniques for solving configuration in our framework. Our main approach is based on Dynamic Constraint Satisfaction Problems to allow efficient addition and removal of the model constraints and user requirements at runtime.

One of the most used types of constraints used in product configuration is a table constraint. We describe the design of a novel algorithm for handling dynamic solving with table constraints. Finally, the interpretation of the ProCoLa language used in our configuration framework is detailed and discussed.

Part IV deals with the practical implementation of the ProCoLa modelling language. We describe the language's integration in a development environment, the analyses implementation, tool support and debugging assistance. An evaluation of the capabilities of the language is also presented through practical experiments.

To conclude the thesis, Chapter 12 sums up the results and gives an outlook on future research. The appendix contains additional material on the ProCoLa language and the case study used throughout this thesis and during evaluation.

Part I

Modelling in Product Configuration

Chapter 2

Setting the Scene

Modelling the product knowledge (i.e., the data representing the characteristics of the products) represents a significant part in the configuration process. It consists in defining the model of a product family that will then be configured by the end-user. Development and maintenance of product knowledge bases are of primary importance, and the representation formalism must be thoroughly considered when choosing a product configuration system. Major vendors of configuration systems already use declarative knowledge modelling [47, 70].

We discuss in this first part the different features that are required from a modelling tool in order to support efficient product knowledge modelling. Deciding on the right modelling language to represent the product knowledge is not an easy task, and many options are available. Therefore, we present in this part several general purpose modelling languages and then evaluate their capabilities in the context of product configuration.

We start by introducing in this chapter relevant background on product knowledge modelling, and describe a case study to illustrate requirements for configuration modelling languages.

2.1 Background

Several approaches for modelling product configuration data have been studied over the years. The original rule-based approach used in the R1/XCON system [10, 64] has the disadvantage that it incurs important maintenance issues, due to rules having influence on both directed relationships (i.e., compatibilities, dependencies, etc.) and actions (related to a solution's computation). In contrast, model-based configuration is based on a strict separation between the product knowledge and the problem solving knowledge (i.e., the mechanisms used to ensure the consistency of the customised product). As the solving process is independent from the product knowledge, this separation provides a better robustness, compositionality, and reusability, making model-based systems the prime choice for configuring large and more complex models [88].

Model-based knowledge representation has become the standard way of dealing with product configuration modelling in recent years. A *configurable product family* is composed of components or functions that are connected together [37, 93]. It can have a large number of variants, and is represented as a *product model*. The model also contains *constraints* that limit the number of possible variants, e.g., by restricting the combinations of values allowed for the different attributes of the product. A common method for modelling in product configuration is the type-instance approach [93]: the model defines a number of *types*, that contain attributes and other properties of the components. Attributes express technical properties of each type, such as, e.g., the storage capacity of a hard drive in a computer model; attribute values are usually limited to a set of potential values. Types are organised in a partonomic tree: each type can define a number of subcomponents from other types. The model can be represented as a tree rooted by a component type. Types are then instantiated as *individuals* during the configuration process to store the actual data of each configured component (e.g., values for attributes). Note that several component individuals can be instances of the same component type, e.g., in case the type was involved in several subcomponent relations.

Configuration systems (or *configurators*) are used to support an end-user in finding a configured product that matches his needs, while respecting the modelled constraints. It takes the product model as input and lets the end-user configure the product during a *configuration process*. Such systems often output a specification of the configured product to be manufactured, usually as *bill-of-materials* (or *BOM*) or *operations routes*.

A BOM defines the list of materials that composes an end-item. It represents the product in the manufacturing perspective. BOMs can be hierarchical and define all the sub-assemblies of the product. In this case, they are called *Modular*

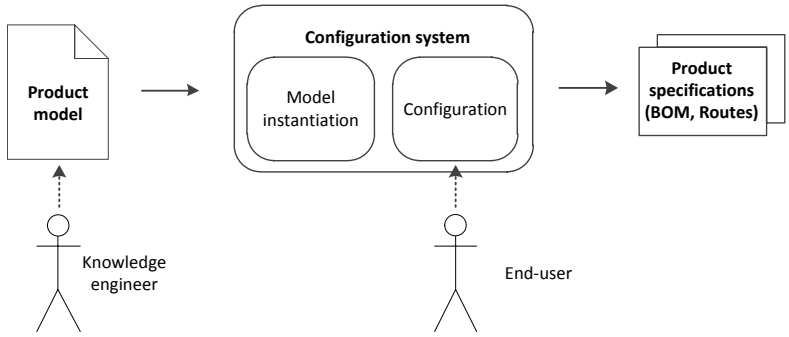


Figure 2.1: Basic architecture of a configuration system

BOM. An example is a BOM for a Personal Computer: this BOM would list the computer and its major sub-assemblies (power supply, mother board, chassis, modem, keyboard, display, etc.), as well as additional materials needed for a complete saleable product – shipping box, user manual, packaging, packaging labels, etc.

Operations (or production) routes describe the list of operations that will have to be performed in order to assemble all the elements of the product. Such production routes can differ depending on the tool used, but often include operators, work load or machine description for example. These routes can also have a hierarchical structure, where operations can have sub-operations, representing the basic units for the production environment. Routes for a Personal Computer could for example include the fixing of the motherboard, or the mounting of the processor.

The creation of both BOMs and production routes can be done in a dynamic way. Indeed, both can differ according to each configuration instance.

2.2 Case Study

In order to illustrate requirements for modelling product configuration knowledge, we designed a product family as a case study for our work. This case study is used as a basis on which to evaluate modelling languages for configuration, and will be used and extended throughout this thesis. The aim of this product family is to present a simple realistic model in order to illustrate the different features required in product configuration.

This section presents the physical model of a Mobile Device product family. The following subsections gives an overview of the product family, and describes the different customisation options and constraints of the model.

2.2.1 Overview of the case study

The Mobile Device model (Figure 2.2) represents a family of portable computing devices such as netbooks, tablets or smartphones. It contains a motherboard and several chips that take care of the processing and the communication capabilities of the device. Some chips are optional, like a GPS receiver that may be added to the product or not. Several components of the same type may also be part of the final product in order to fulfill different functionalities, such as wireless chips for bluetooth, wifi or FM radio. Each device also possesses a screen and a storage drive that are also customisable.

The Mobile Device product family can be customised in the following way:

- A device can have different ports, such as a video port or an Ethernet port. In some cases, an video adapter can be provided with the device to provide additional functionalities.
- Along with the different wireless chips available, other types of chips can be added to the motherboard, such as a compass, an accelerometer or a GPS chip. Phone capabilities can also be provided via a Radio Frequency card. Finally, several options are available for the motherboard's processor.

Figure 2.2: Overview of the case study

- Different types of screens can be chosen for the device, e.g., an OLED screen or an LCD one. The size of the screen is also variable, depending on the type of device chosen. Also, in certain cases, a touch screen (resistive or capacitive) can be chosen, with additional options like oleophobic coating, that can be applied on the screen to avoid fingerprints. In case of a resistive touch screen, a stylus needs to be provided with the device, and the enclosure must have an allocated slot to store it.

- Finally, several types of storage drives are available. A Solid State Drive (Flash or DRAM) or a Hard Disk Drive can be picked, with different memory capacities.

These characteristics of the product family can be represented as types and valued attributes that will be used for configuration. For example, a Motherboard type is related to the main (root) component as a subcomponent, and contains itself several subcomponents defined by types such as a Processor type. The Processor type then has a *frequency* attribute that can have a value of 600 Mhz, 1 GHz or 1.6 GHz.

Asides from those elements, the motherboard provides some chip slots, depending on its size and the chips already on it. Any other chip that may be added to the device, such as the processor or the GPS receiver also needs some room on the motherboard. It is thus necessary that the number of slots available must always be bigger or equal than the number of slots used, in order to avoid a product design impossible to manufacture.

The product model is also subject to constraints that restrict the model elements. These are detailed in the next section.

2.2.2 Constraints

This section introduces several constraints that prevent incompatibilities in the Mobile Device model:

- If a touch screen is chosen for the device, the motherboard should contain specific chips to handle the touch capabilities of the device. Moreover, in the case of a resistive touch screen, a stylus should be added to the device and the enclosure should have a stylus slot. Finally, the size of the stylus should be adapted to the overall size of the device's enclosure: in case of a small enclosure, only a compact stylus should be available.

- In case an internal Ethernet card is added to the motherboard, the enclosure must have an Ethernet port, and vice versa. The type of video port on the device's enclosure should also match the size of the device: if the device is small (e.g., a smartphone), only a special video port (miniUSB) can fit. In that case, it may be possible to choose a miniUSB video adapter with the device.
- The capacity of the storage drive depends on the type of drive chosen: an SSD cannot exceed 64 GB, while for an HDD the capacity is over 64 GB.
- Table 2.1 shows a set of combinations of values relating the size of the enclosure with the size of the screen and the number of available chip slots on the motherboard.

These constraints should always be satisfied to ensure that the final configuration of the product family is consistent with the specifications of the proposed devices.

Enclosure size	Available chip slots	Screen size
115x58	6	3 inches
240x190	10	9 inches
266x178	12	10 inches
295x210	14	12 inches

Table 2.1: Combinations of allowed attributes values depending of the size of the enclosure

2.3 Modelling Requirements for Product Configuration

In this section, we discuss several key requirements for modelling product families in product configuration. These requirements are derived from various sources in the literature as well as discussion with industrial partners. Together with the case study presented earlier, this list will be used in the next chapter to evaluate several existing modelling languages for product configuration. They also form a basis on which to build a modern modelling environment for product configuration. Although some of these requirements are more important than others, it is important to consider all of them when designing a configuration framework.

We first present general requirements, followed by more specific requirements related to representing the structure of the product and for modelling constraints.

When possible, an example is given referring to the case study introduced in the previous section.

2.3.1 General modelling requirements

In this part we present the general requirements for a modelling environment:

1. **Separation between product modelling and configuration process**: The first requirement concerns the structure of the configuration framework itself. As underlined in [4], product knowledge modelling should be clearly separated from the configuration process. Indeed, these two tasks are usually performed by different persons, most of the time with different skills: the modelling task would usually be performed by a design engineer, while the configuration task would be performed by a sales person. Thus this abstraction is needed so that there is no confusion between these two parts.

2. **Easy-to-use**: The persons that will interact with the modelling environment are usually design engineers, often possessing only basic programming skills. The modelling environment should therefore be accessible without advanced training in programming, and support easy development through tools for a fast implementation. Also, the terms used should be based on a widely accepted terminology, e.g., following well-known conceptualizations of configuration [37,93]. Finally, creating a product model is a tedious and error-prone process. Providing a modelling environment with development support is essential to improve the modeller's productivity. Tools should be provided to assist the modeller in defining constraints or managing the model.

3. **Support of object-oriented concepts**: This approach has been favoured by many researchers [7,51,52,61,85]; it is indeed very suitable for product modelling, as product components can naturally be seen as objects.

   Example: *The structure of the Mobile Device product family described in section 2.2 is object-oriented. It follows a type-instance approach: the model defines a number of* component types*, such as a Motherboard type or a Screen and Touch Screen types, that contain the attributes and constraints of the product family. Those types are then instantiated as* individuals *during the configuration process to store the actual data of each configured component. Note that several component individuals can be instances of the same component types: for example, a Wireless Chip component type may have up to three different instances (e.g., for a wifi, a bluetooth and a FM radio chip) during configuration time.*

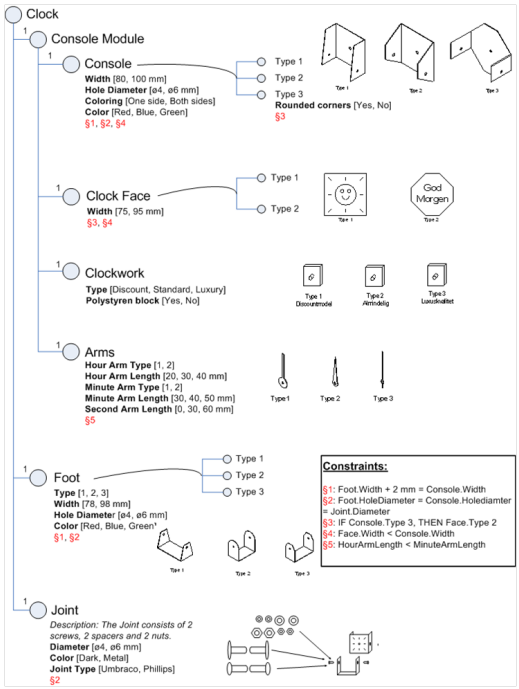


Figure 2.3: PVM view of a Clock model

4. **Graphical representation**: Providing a graphical representation to the user is important for an easy comprehension and a lower maintenance effort [4, 52]. Thus the modelling environment should be able either to provide directly a graphical view, or to have a structure such that the implementation of a Graphical User Interface on top of it is possible. Such a graphical representation should contain information about the different attributes of the products as well as their structure.

   Example: *An example of a graphical representation can be the* Product Variant Master (PVM)*, developed by [51] (Figure 2.3).*

5. **Extensibility**: Companies use many applications around configurators. The system should provide an easy integration of CAD tools, databases, ERP or other systems in the configuration process.

2.3.2 Structure modelling

One part of modelling in configuration deals with representing the structure of the product. Several key features can be highlighted:

1. **(Dynamic) partonomy relations**: Partonomy (or part-of) relations define a subcomponent hierarchy in the product model. The multiplicity of these relations corresponds to the number of subcomponents to consider. Providing support for a variable multiplicity allows a better reuse of component types in the model.

   Example: *In the case of the motherboard that may contain several wireless chips, the Motherboard component type is involved in a partonomy relation with the Wireless type as a subcomponent. The multiplicity is dynamic, as there may be from 0 to 3 instances of the wireless chips during configuration, depending on the choice of the end-user and the different constraints in the system.*

2. **Taxonomy relations**: Taxonomy (or kind-of/specialisation) relations permit the use of generic base components to group features that are common to several subcomponents, which eases modelling and maintenance of the model.

   Example: *The Mobile Device model features a screen that can be specialised as a touch screen. The Touch Screen component type reuses the same attributes as the Screen type, and declares new ones, e.g., its type (resistive or capacitive). Thus, if the attributes or constraints of the Screen type evolves, there is no need to change the Touch Screen type, reducing the problem of maintenance.*

3. **Data types and units**: A product can be complex and contain many different data types. That is why data types such as enumerations, sets, or integers can be needed. It should be possible to declare different units, in order to make the model's creation and maintenance easier.

   Example: *The frequency of the processor, the size of the screen, or the capacity of the storage drive can all be defined as integer numbers. However, they all have different units: the frequency is in Megahertz, the size in inches, and the capacity in Gigabytes, which should be included in the model, to avoid ambiguities when dealing with these different dimensions, and to make the model specification closer to the actual product family.*

4. **Connection ports**: Connection ports represent non-hierarchical relations between components that may be located in different subtrees of the model. Specific attributes can also be added in order to provide configuration possibilities to these relations.

Example: *The Stylus component type in the Mobile Device model connects the Touch Screen and the Enclosure types in a relation that do not involve any parent-child relationship. Furthermore, an attribute (the size of the stylus) is attached to this relation.*

5. **Default values**: Another useful feature is the possibility to declare default values for the attributes of the models, as pointed out in [52] . These values will then be assigned to the attributes when the configuration process is started, and the user would have the possibility to change them during configuration time.

   This permits to improve the configuration experience, avoiding long default set-up of the models during configuration.

   Example: *Any attribute in the case study could be associated a default value. For example, it can be decided that there should be no miniUSB adapter in the configuration by default, although the end-user can decide to add one.*

6. **Hidden/Read-only attributes**: Providing the possibility to specify the visibility of attributes helps the designer to create models easier to maintain [52], as some of the attributes may be used as intermediary data containers, and thus may not be accessible to the end-user. Some other attributes can also be used as read-only, in order to provide the customer with some unmodifiable information.

   Example: *A hidden attribute can be defined in the Motherboard component type that describes the actual number of chips already on the motherboard, depending on the number of touch chips and whether an accelerometer or a compass chip is chosen. This attribute will not be presented to the end-user, but can be used in constraints or to specify the actual number of chip slots used on the motherboard.*

7. **Production attributes**: Industrial product configurators are usually meant to be integrated with production management software, like Enterprise Resource Planning system (ERPs). This includes mapping the configuration output to a bill-of-materials (BOM) and operations routes that can be used in sales and manufacturing.

   Allowing the definition of production attributes that model how theb BOMs and routes will be constructed from the product model's components is an important step towards the automatic generation of production data.

2.3.3 Constraint modelling

Another important aspect of product modelling is the definition of constraints on the model. Requirements for constraint modelling are:

1. **Built-in functions**: A panel of built-in functions and constraints should be made available to the modeller: aside from simple arithmetic and logical constraints, advanced functions (e.g., sum) or constraints (e.g., allEqual) provide better support to the product modeller.

2. **Product catalogues**: More and more real product data is coming from tables representing allowed combinations of attributes/components, for example in product catalogues.

   The ability to declare table constraints directly (instead of more complex formulas) simplifies by far the creation and usability of the model. That is why an easy-to-use modelling tool for these constraints should be available.

   Example: *The table constraint representing the allowed combinations for the screen size and the number of chip slots on the motherboard depending on the enclosure size is a typical product catalogue. Defining such constraints as tables (instead of long series of value assignments) involves less repetitions of attributes, which make them easier to maintain. They can also be taken from various tools such as databases.*

3. **Balance of resources**: Products are often configured according to the resources they produce/consume, such as the energy produced and consumed in a system for example. These resources can then be used in constraints, enabling the configuration tool to ensure the consistency of the system.

   Example:*The number of available chip slots on the motherboard can be define as one of the system's resources. The motherboard is producing this resource depending on the number of chips pre-installed on it, while additional chips may consume this resource.*

4. **Hard/Soft constraints**: *Hard* constraints are constraints that must not be violated, while *soft* constraints may be violated if they are overridden by a user selection or indirectly as a consequence of a constraint with higher priority.

   This kind of distinction provides the designer with the ability to guide the configuration process with specific "recommendations", as he has a solid knowledge of the product, though it is still possible for the customer not to follow them.

> ***Example:*** *Soft constraints could be added to the Mobile Device to act as recommendations. One might add a constraint specifying that a large touch screen (size bigger than 9) should be a resistive screen, although the end-user might overrule this.*

Now that these requirements have been motivated and listed, we will in the next chapter model our case study in three general purpose modelling languages, and discuss the advantages and pitfalls of each.

Chapter 3

General Purpose Modelling Languages

In this chapter, we present and analyse three general purpose modelling languages in the context of Product Knowledge Modelling. A *modelling language* is a language that is used to represent knowledge or information in a structured way. It can be used to express a lot of different systems, from the Enterprise Architecture ([57]), to Software Engineering ([104]), through products architecture, among others. Representing these concepts in a formal way can have several advantages, including:

- Providing formal specifications that can be reused and exchanged, improving team understanding and communication.
- Defining the conceptual design of a system, which allows to avoid mistakes due to the lack of a clear vision of the system's architecture.
- Formalizing a system's structure and requirements, and providing a basis for a clear and well-defined structure for implementation (in case of software development), manufacturing (e.g. for a product), etc.

A modelling language is defined by two fundamental parts: its *syntax* (the rules defining how to write a model) and its *semantic* (how the language should be

interpreted). The syntax of a modelling language consists usually of *abstract* syntax and *concrete* syntax. The concrete syntax represents the visual part of the language that will be interfaced to the user, while the abstract syntax expresses the inner representation of the data from the models. Not all modelling languages are executable, however, and in the case of product configuration, at least a link to an executable language is required in order to interface with a solving engine.

Modelling languages usually fit into two main categories: *graphical* and *textual* modelling languages [48]:

- *Graphical* languages contain **diagrams** with symbols to express the different concepts needed to represent a specific information. The symbols are usually linked together by lines that represent the relationships between them. The concrete syntax of these languages is thus their graphical notation.
- *Textual* languages contain **standardized keywords** as concrete syntax in order to structure the knowledge representation. This representation is usually interpreted by a computer in the abstract syntax.

The languages discussed in this chapter are object-oriented languages. As previously discussed, object-oriented concepts have great advantages when it comes to product configuration models. These languages have been chosen because of their wide use and their well-known ability to model systems. We use the requirements and the case study described in the previous chapter in order to assess the potential of these languages for use in modern product configuration modelling environments, and discuss each advantages and pitfalls.

3.1 UML and OCL

This first section presents the Unified Modelling Language (UML), associated with the Object Constraint Language (OCL).

3.1.1 The Unified Modelling Language

The **Unified Modelling Language (UML)** is an international standard defined in 1997 by the Object Management Group (OMG). It started as version

1.1, and a major revision has followed, with the adoption of the UML 2.0 version in 2003 by the OMG. The current version is 2.3.

UML is a visual specification language for object-oriented modelling. It has been created as a general-purpose modelling language, and includes a graphical notation used to create an abstract model of a system, that is referred to as a UML model.

UML 2.0 contains 13 types of diagrams, that are organized hierarchically (Figure 3.1).

There exists a relatively important amount of concepts used in UML 2.0 for object-oriented design, from structure concepts (e.g., classes, components, packages) to relationships (e.g., aggregations, associations, generalization) through behavioural concepts (e.g., events, messages, methods).

In order to model products for product configuration, only a subset of all these concepts and diagrams is used. Indeed, most of the interest in product modelling for configuration lies in the product structure, its attributes, subcomponents, and the constraints around all of them. The UML *class diagram* is of prime interest as it contains the structural elements that can help us represent the product configuration models.

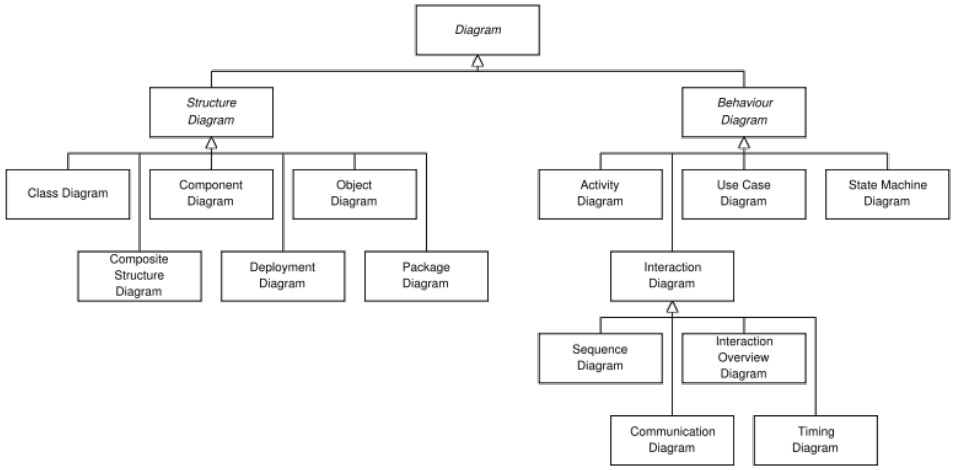


Figure 3.1: UML diagrams [104]

We detail several UML modelling concepts relevant to this chapter:

- UML contains different types of relations between components, including:

  1. An *Association* establishes a semantic relationship between two components. A good way to illustrate this is by comparing it to a marriage: an association is binding a man and a woman. In the case of the marriage, the multiplicity is one-to-one, but in general it can have different multiplicities.

  2. An *Aggregation* is a specialized form of association, and can be either *shared* or *composite*. It represents typical whole-part relationships, where a notion of ownership exists. In the case of a shared Aggregation, all objects have their own life cycle. For example, an object representing a Department can have a shared Aggregation relationship with a Professor object: the Professor belongs to this Department, but if the Department is deleted, the Professor will still exist. On the contrary, for the composite Aggregation (or *Composition*), the life-cycle of the child is linked to the life cycle of the parent: if the parent is deleted, then the child will be, too.

  3. The *Generalisation* is used to model inheritance for data and code reuse: the child element inherits all the properties of its parent, and can define new ones.

- UML 2.0 contains an extension mechanism called *Stereotypes*. A stereotype allows designers to extend UML by creating new model elements from existing ones. The new nodes are then stereotyped, which is reflected graphically by adding a name enclosed by quotes above the name of another element. Each stereotype can contain *tagged values*, which represent values specific to the stereotyped elements.

An example of a very simple model of car can be seen in Figure 3.2. The Car *Class* represents the Car component, and has two *Attributes*: **color**, that can be either *Black* or *White*, and a boolean **hasSunRoof**. It also has four **wheels**, that are modelled by an *Aggregation* relation with the Wheel class, and an engine, thanks to the aggregation with the Engine class. Finally, it has a *constraint* that specifies that "If there is a sun roof, the color is White". Moreover, the Car is the *Generalisation* of two classes: a Standard car and a Cabriolet, which is more constrained, as "it has no sun roof and it has no spare tire".

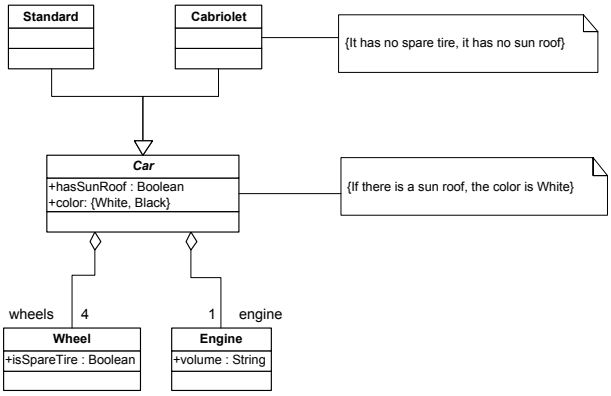


Figure 3.2: A simple car model, with generalization and aggregation relationships

3.1.2 The Object Constraint Language

It should be pointed out that the constraints in Figure 3.2 are expressed in natural language. Another way to express constraints in UML is by using a programming language. What is missing in UML is the ability to describe constraints in a more formal way than natural language or raw code. This is what the **Object Constraint Language (OCL)** is being used for.

OCL is an extension to UML that allows to write standardized constraints. It is actually a textual language that provides constraint and object query expressions that cannot be expressed using notations like diagrams. The aim of OCL is to provide an unambiguous language for constraint specifications, but that can stay accessible to persons with few or no programming skills. OCL is a pure specification language, which means that an OCL expression has no any side effects. Indeed, when an OCL expression is evaluated, it simply returns a value, and does not change anything in the model.

An *OCL statement* is always evaluated in a specific *context*. The context defines the situation in which the statement is valid, e.g., a class. Then the body of the constraint is defined. OCL statements can contain *navigation expressions* such as `c.hasSunRoof`, which, if `c` is a Car, results in fetching the value of its attribute *hasSunRoof*. Finally, OCL constraints can be *invariants* for a specific class, or *pre/post conditions* for a specific operation, though only invariants are used in product configuration, as components do not contain operations. In this work, we only use a subset of the OCL constraints. For a complete overview, refer to [72].

It is now possible to express the constraints of Figure 3.2 using OCL (note that the keyword **self** refers to an object of the class being constrained, though it can be left out in most cases, when the context is clear):

- For the constraint on the Car "If there is a sun roof, the color is White":

```
context Car
inv: hasSunRoof implies color = ''White''
```

- Specifying that a Cabriolet has no sun roof can be done in two ways, depending whether the constraint has to be evaluated in the context of the Car or of the Cabriolet itself:

```
context Cabriolet
inv: hasSunRoof = false

context Car
inv: self.oclIsTypeOf(Cabriolet) implies hasSunRoof =
    false
```

  The OCL function `oclIsTypeOf(Type t)` checks if a given instance is an instance of type *t*.

- Finally, specifying that a Cabriolet has no spare tire:

```
context Car
inv: self.oclIsTypeOf(Cabriolet)
     implies wheels -> forall(w:Wheel | w.isSpareTire =
         false)
```

  The operator `->` is used to call an operation on a collection, in the following way:

```
collection->operation(arguments)
```

  Also, the `forall` construct permits to test a boolean expression on all elements of a collection. The declaration of the elements' type (`w:Wheel`) can be left out when unambiguous.

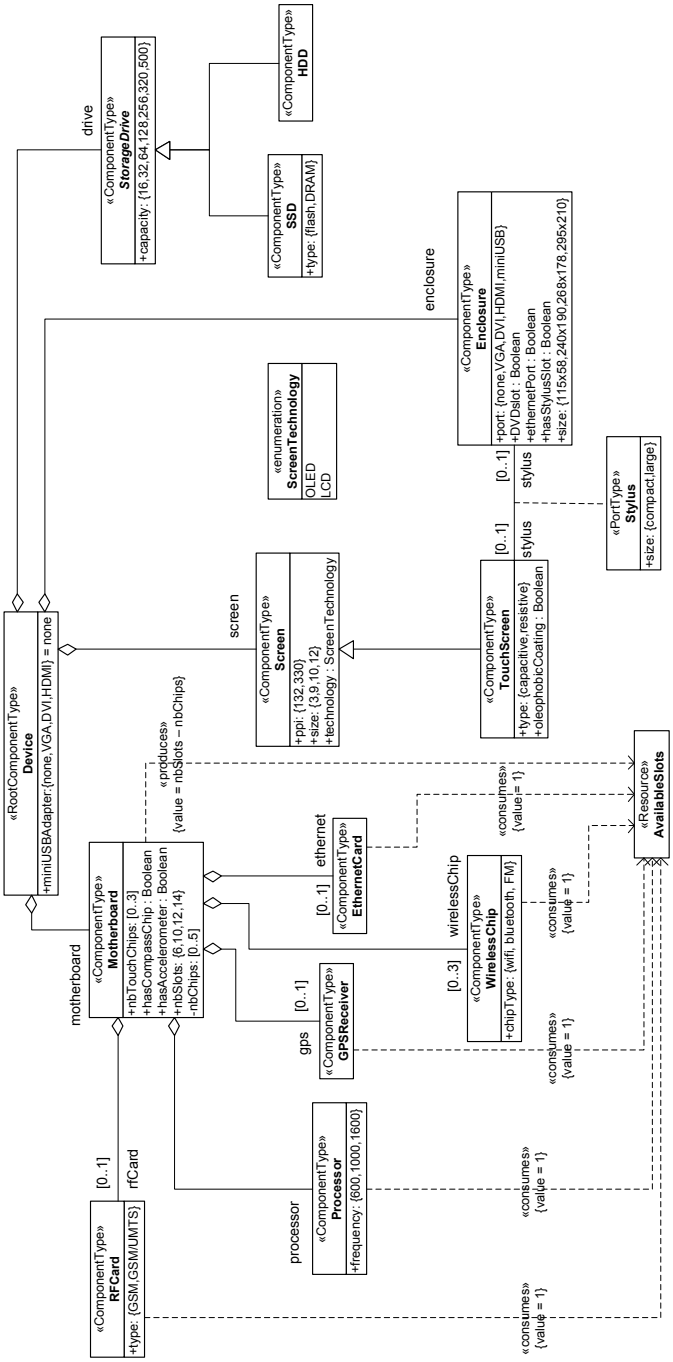


Figure 3.3: UML model for the Mobile Device case study

3.1.3 Implementation of the case study in UML/OCL

This part presents the implementation of the Mobile Device case study introduced in Chapter 2 in UML/OCL. The UML diagrams have been extended here using stereotypes in order to reflect the product configuration concepts, following Felfernig et al. [40, 41]. The stereotypes *ComponentType*, *RootComponentType*, *Port*, and *Resource* have been defined from the UML metamodel element *Class*, while the *consumes* and *produces* are from the UML metamodel element *Dependency*. The stereotype *PortType* has also been defined on the *Class* element.

Figure 3.3 shows the Mobile Device model implemented in UML 2.0. Several points are worth being highlighted:

- It is not possible to specify units like GB (Gigabytes) or MHz (Mega Hertz) for the **capacity** attribute of the StorageDrive component type or the **frequency** attribute of the Processor for example. Indeed, only basic data types are available. Thus these attributes are declared as *integers*.
- It is possible to declare the StorageDrive class as *abstract*. This means that an instance of the Storage Drive type will need to be specialised into one of its subtypes (SSD or HDD) in a final component instance.
- Each partonomy relation has been represented as a UML (shared) *Aggregation*, as it is often the case in product configuration that the child (or subcomponent) is independent from its parent and could be reused in other partonomy relations. The multiplicity of each aggregation represents the multiplicity of the partonomy relation, with expressions like "0..3" meaning that 0 to 3 instances of the component types can be present in the final configuration. Note that a default multiplicity of 1 is implicit when it is not specified.
- Taxonomy (or kind-of) relations are represented by UML *Generalisation*. The TouchScreen component type is for example related to the Screen type by a UML Generalisation, inheriting all the attributes and operations of the Screen type.
- Each attribute is represented by its visibility, its name and its type:
  - A *public* attribute (represented by a "+") is an attribute that is visible by all, while a *private* attribute (represented by a "-") is hidden (e.g., the attribute **nbChips** of the Motherboard type).
  - Different predefined data type are available in UML, including all the most classic ones (*Integer*, *Boolean* ...). Moreover, it is possible to

define a specific *enumeration* each time the domain of an attribute is composed by predefined values (e.g., the attribute **technology** of the Screen type uses the ScreenTechnology enumeration). However, it is possible to display those enumerations inline in the attribute definitions when only used once (e.g., the attribute **type** of the RFCard type), although it has to be written in the attribute name, which does not fit the right UML specifications.

  - It is also possible to define *locked* parameters in UML 2.0 using the **{read-only}** keyword after the attribute's definition.
  - An initial value can be provided to act as the default value of an attribute, e.g., the miniUSBAdapter attribute of the Device type is initially set to "none".

- The *connection port* between the touch screen and the enclosure is here represented using a UML *Association* with a multiplicity of 0..1 on each sides. This denotes that it is possible to have a Stylus instance associated to both instances (Touch Screen and Enclosure) during configuration, although it is not mandatory. A *port type*, called Stylus, contains information on the size of the stylus used in the final product.

- Moreover, the number of available slots in the motherboard has been modelled using a UML *Class AvailableSlots* annotated with the *Resource* stereotype. Then, a component produces some of this resource (the motherboard) while others need some of it (the chips).

Using OCL, it is possible to specify some rules in order to describe the correct usage of the stereotypes [40]. For example:

```
-- Comment: the component classes does not have
    operations
context ComponentType
inv: self.allOperations->size = 0
```

In this example, the OCL *allOperations* attribute represents the collection of all operations defined in a class. This constraint specifies that no UML Operations should be defined in a UML Class with the *ComponentType* stereotype.

Following the UML implementation, it is still necessary to define, using OCL, some of the constraints that cannot be represented graphically:

- The following constraints represent the need for touch-enabling chips for touch screen devices, the presence of a stylus for resistive screen and its impact on the device's enclosure:

```
context Device
inv: (screen.oclIsKindOf(TouchScreen)) = (motherboard.
    nbTouchChips > 0)

context TouchScreen
inv: (screenType = 'resistive') = (stylus -> size() = 1)

context Enclosure
inv: (hasStylusSlot) = (stylus -> size() = 1)
      and (size = '115x58') = (stylus.size = 'compact')
```

- These other constraints ensure that an Ethernet port is present if an Ethernet card is added to the device, and govern whether a miniUSB adapter is available.

```
context Device inv:
(motherboard.ethernetCard -> size() = 1) = (enclosure.
    ethernetPort)
and (enclosure.port <> 'miniUSB') implies (
    miniUSBAdapter = 'none')

context Enclosure
inv: (size = '115x58') = (port = 'miniUSB')
```

- The constraints on the capacity of the storage drive are defined as follows:

```
context SSD
inv: capacity <= 64

context HDD
inv: capacity > 64
```

- Product catalogues: no expression for table constraints has been implemented in UML/OCL. This does not make impossible the implementation of product catalogues (i.e., table) constraints, but it makes it much more tedious to write, and thus to maintain, as it has to be converted into a logical expression. The constraint from Table 2.1 can thus be written in OCL as:

```
context Device inv:
(motherboard.nbSlots = 6 and (enclosure.size= '115x58'
    and screen.size = 3) or
(motherboard.nbSlots = 10 and (enclosure.size= '240x190'
     and screen.size = 9) or
(motherboard.nbSlots = 12 and (enclosure.size= '266x178'
     and screen.size = 10) or
(motherboard.nbSlots = 14 and (enclosure.size= '295x210'
     and screen.size = 12)
```

3.1.4 Discussion

This part summarizes the insights that have been provided by the experience in modelling the case study using UML 2.0 and OCL.

UML provides a direct graphical view of object-oriented structures such as software architecture or product architecture in the case of this dissertation. This permits to have a **clear view** of how the components are related to each other. Moreover, as UML is widely applied in industrial software development as a standard model design, its concepts are relatively well known, which makes it an **accessible modelling tool**. The UML diagram library contains an important amount of object-oriented diagrams, and the **use of stereotypes** permits to extend the existing concepts to configuration-specific concepts.

The integration of the Object Constraint Language (OCL) allows the designer to define complex constraints in a relatively accessible language, that does not require advanced programming skills either. OCL contains a wide variety of constraints, which would be sufficient for most configuration problems.

However, there are some issue with UML and OCL. First of all, the UML language is not aimed specifically at general product modelling, and even less for product configuration, and it can be thus **difficult** for knowledge engineers **to adapt** it to product configuration: the UML language itself had indeed to be extended through the use of stereotypes to fit the configuration problem. All the concepts are aimed at software engineering, which is illustrated by the fact that it is not possible to directly declare **new units** other than the ones used in programming languages (Integer, Boolean...). The lack of support for product configuration specific constraints is also a problem: indeed, constraints defining product catalogues or soft constraints do not have a specific implementation in UML. Although a series of logical constraints can represent product catalogues, a specific support for table constraints would improve the modelling experience and add maintainability to the model.

There is also no specific concept in UML that allows to represent production attributes such as BOM and operation routes. One solution could be to use UML Classes with new stereotypes, and map them to the types describing the product family structure. OCL constraints may then be used to ensure that the right item or operation is present depending on the values of the component type's attributes.

Finally, one of the biggest drawbacks in using UML and OCL is the problem of interpretation of the language. It is indeed necessary to be able to interpret the models into a declarative representation of the configuration knowledge, in

order to apply solving algorithms. Felfernig et al. [41] have provided UML/OCL interpretation in first order logic in order to be able to use it in a configurator. Again, this type of automatic interpretation requires a strictly defined UML profile for product modelling in configuration. Research has also been going on in this area [1, 2], but more remains to be done in order to provide a strong tool support and debugging facilities to the modeller in the context of product configuration with UML/OCL. A few tools to check the validity of UML/OCL models exist (e.g., USE [45]), but they usually do not permit to find solution to the configuration problem.

3.2 SysML

The Systems Modelling Language (SysML) is a recent modelling language specified by the OMG. It is actually a UML profile, and thus inherits the characteristics of UML. The aim of SysML is to represent systems and product architectures, as well as their behaviour and functionalities, where UML was used for software engineering. The relationship between UML and SysML can be seen in Figure 3.4.

The development team of SysML aimed on the one hand at limiting the concepts semantically too close from software engineering, and on the other hand at simplifying UML original notations by limiting the number of diagrams available, in order to make it easier to use. Figure 3.5 shows the SysML diagrams.

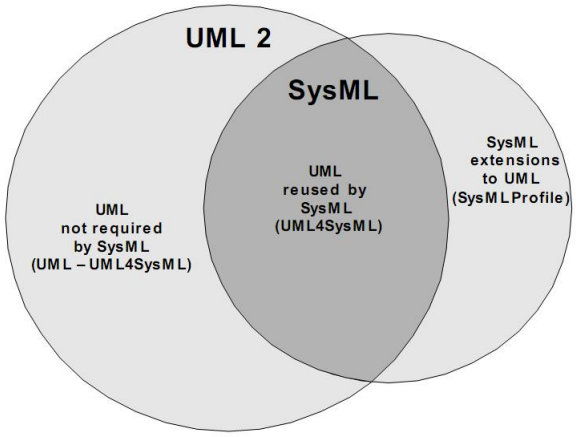


Figure 3.4: Overview of SysML/UML interrelationship [96]

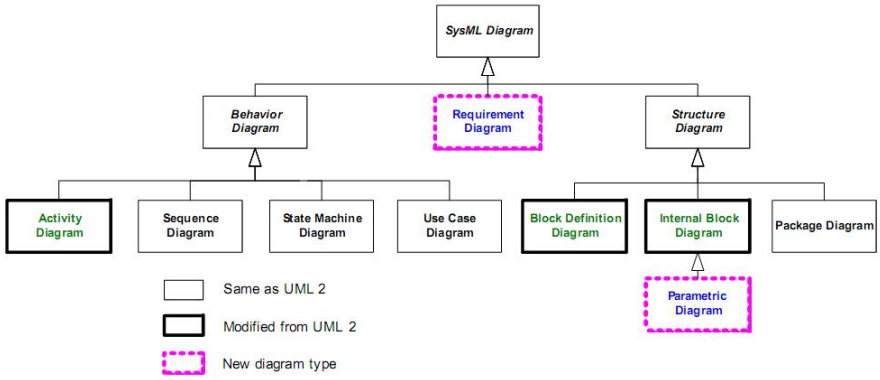


Figure 3.5: SysML diagrams [96]

Some diagrams, like the UML *Class Diagram*, renamed *Block Diagram*, have just been modified through the concept of *Block*, that allows to express any structural element of a system.

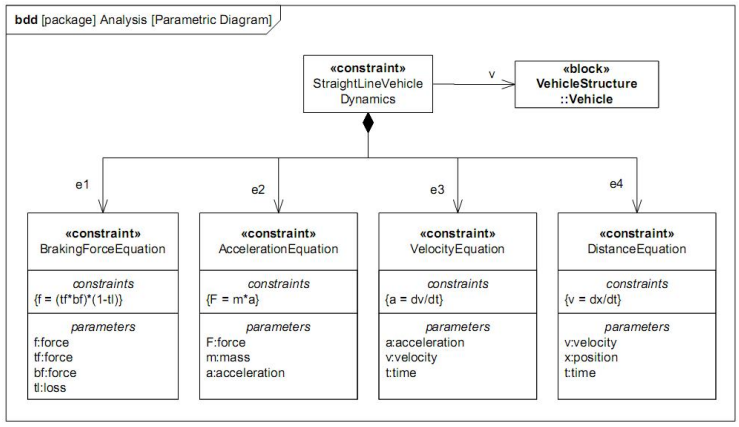
Two new diagrams are also present in SysML:

- The *Requirement Diagram* is used to specify the needs of the system. These requirements can be mapped together and to the different components of the system.

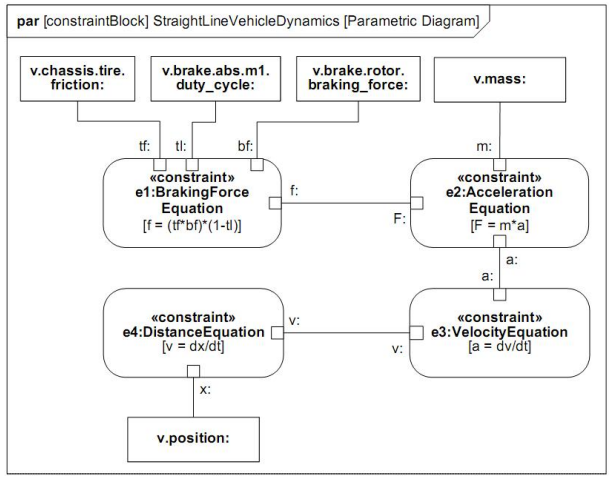
- The *Parametric Diagram* is the second new diagram. It is used to specify mathematical expressions between the different elements of the model.

These new diagrams aim at making the system and product modelling more complete. The Parametric Diagram is particularly interesting as it permits to declare complex constraints on systems, using mathematical expressions or any other constraint mechanism already available before. Figure 3.6(a) represents a Block Definition Diagram for defining constraints on a special car model, while Figure 3.6(b) uses the equations in a Parametric Diagram to constrain value properties of the model.

These new concepts directly implemented in SysML bring interesting improvements for product modelling, especially concerning product configuration. Another interesting feature is the possibility to declare one's own dimension and units, that can thus be used in the model.



(a) Block Diagram



(b) Parametric Diagram

Figure 3.6: Block and Parametric Diagrams for HSUV vehicle dynamics [96]

3.2.1 Implementation of the case study

The implementation of the case study in SysML is decomposed in several different diagrams. Indeed, SysML allows the definition of all the constraints using Parametric Diagrams, which can be practical when combined with other diagrams such as Requirement Diagrams, if the whole product specification is to be done in SysML. Figure 3.8 shows the Block Definition Diagram for the Mobile Device product family.

A first point is that this block diagram looks like the one built in UML (with blocks, aggregations, packages...). Modelling using SysML is very similar than doing so with UML, except that almost no user-defined stereotype is needed. Indeed, only the *resource* stereotype on resources have been added on a UML Class. Units are shown in Figure 3.7. The advantage of declaring units lies in the more acurate modelling experience, as well as the fact that it helps maintaining the model, by providing clearer specification of the product.

As described before, it is possible to describe parametrized constraints using Parametric Diagrams in SysML. Figure 3.9 shows the definition of the constraints that are used in the model, while Figure 3.10 represents the Parametric Diagrams for the Device root block.

The goal of these Parametric Diagrams is to map the inner attribute of a block to the constraints. This allows to reuse constraints in a more structured way than with raw OCL declaration. However, it must be pointed out that the constraints can still be written as formulas or OCL statements.

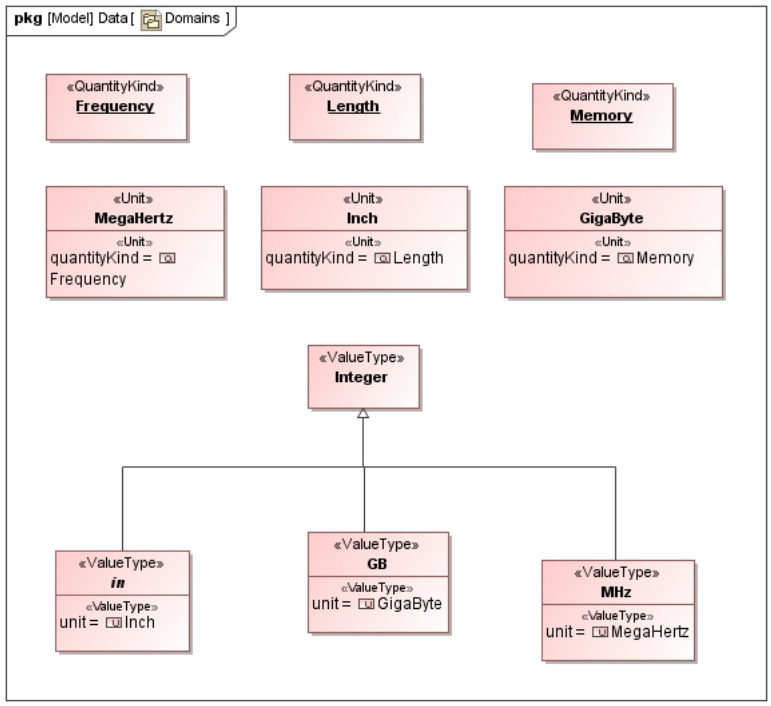


Figure 3.7: Units definition with SysML

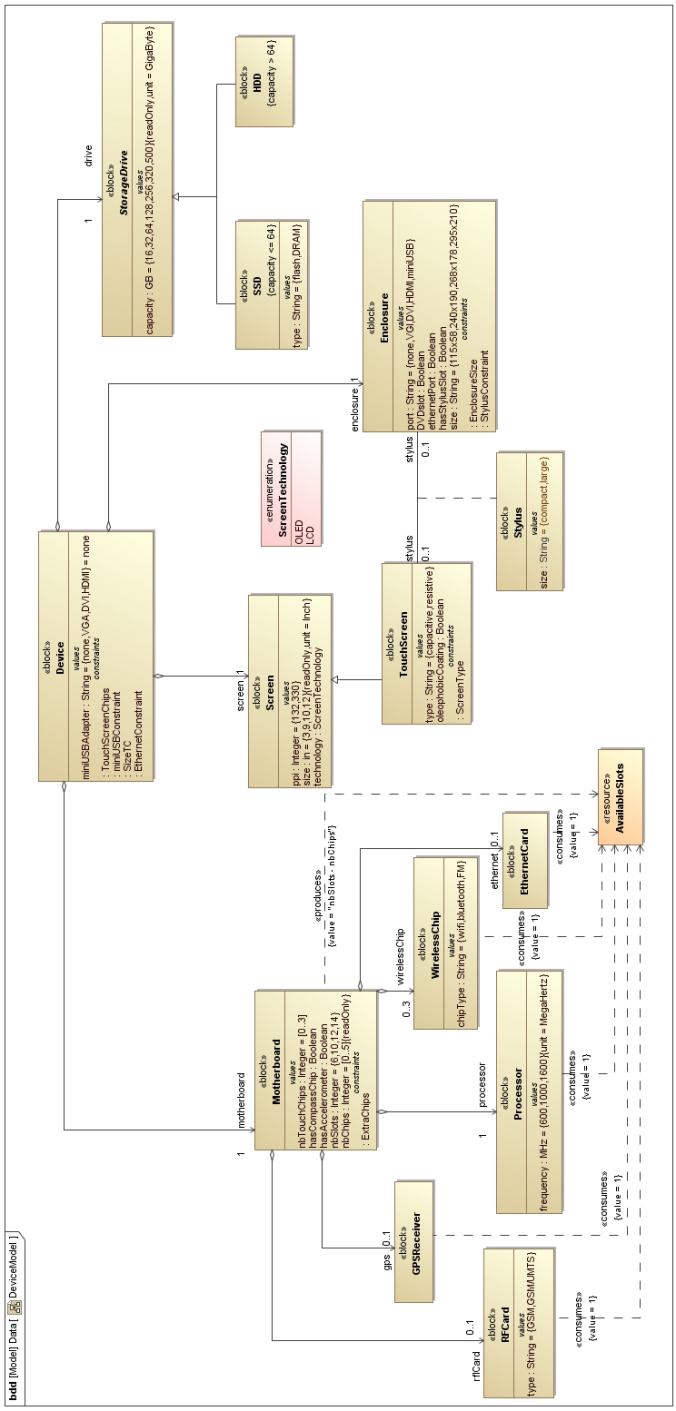


Figure 3.8: SysML Block Definition Diagram for the Mobile Device case study

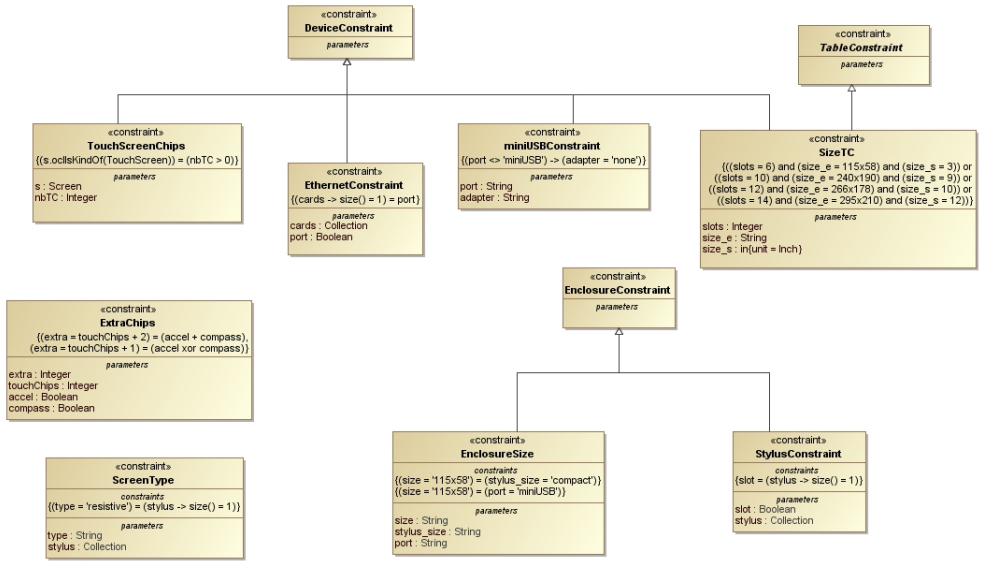


Figure 3.9: Block Definition Diagrams defining the constraints used

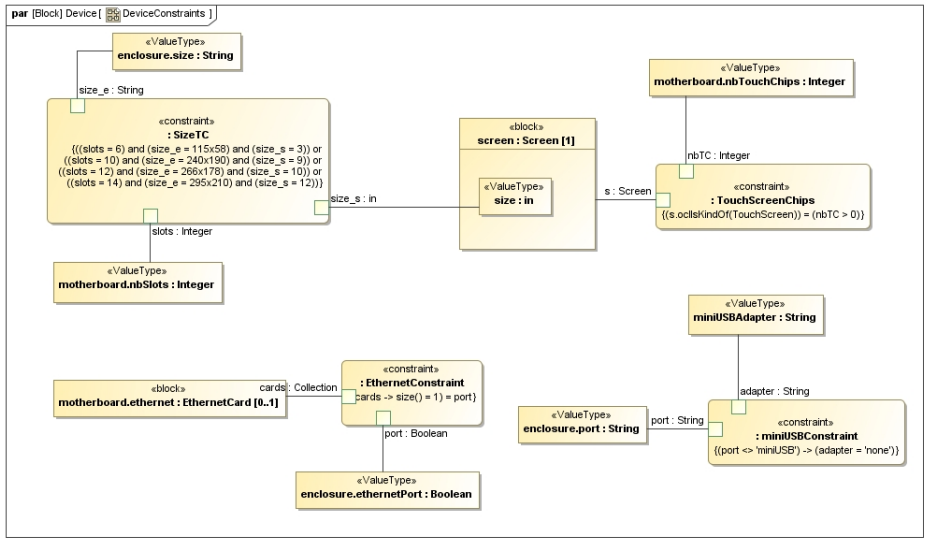


Figure 3.10: Parametric Diagrams for the Device block

3.2.2 Discussion

SysML represents an interesting and powerful extension of UML when it comes to modelling systems and products. It is indeed possible for a knowledge engineer to design a product model for configuration in SysML almost without adding any new user-defined concepts. The power of SysML also lies in its simplicity compared to UML (fewer diagrams), and the possibility to design a complete product model, including requirements, that can be added to the structural model that is needed for product configuration, though no specific feature is introduced to define a mapping to BOM or routes information.

In a product configuration-only point of view, several interesting features are added compared to UML. The possibility to introduce **constraints in diagrams**, and to map a model's internal attributes to their use in the constraints, brings a **much clearer view** for knowledge engineers not used to raw programming language file, though OCL is still needed in complex constraints.

Thanks to this graphical view, it is also possible to group constraints linked together (see example in Figure 3.6(a)), and to declare them in a parametric way, so they can be generalized and reused. SysML also permits the declaration of **user-defined units**, making the model closer to the physical product, and allowing a much easier maintenance of the system, especially in the case several different engineers are using it.

However, there are still things missing in SysML. **Table constraints** are still not available, though it is still possible to implement them using OCL, as in UML. Some concepts, like *resources* produced/consumed, had also to be added using stereotypes, though less than in UML. It is also not possible to declare which block is the *root element* without using **stereotypes**.

Finally, the **interpretation** of the language stays an open issue: as SysML is a graphical language, it needs to be efficiently converted so that a constraint engine can be used, as well as to support tools to assist the knowledge engineer during the model's creation.

3.3 EXPRESS/STEP (ISO 10303)

This section introduces the *International Standard ISO 10303*, which is referenced as *STEP* (STandard for the Exchange of Product data).

STEP was first released in 1994, and is published as a series of Parts:

- Part 1 provides an overview;
- Parts 11, etc., specify description methods (the EXPRESS family of information modelling languages);
- Parts 21, etc., specify implementation methods (data exchange mechanisms);
- Parts 31, etc., specify conformance testing methodology and framework;
- Parts 41, etc., specify integrated generic information models;
- Parts 101, etc., specify integrated application resource models;
- Parts 201, etc., specify Application Protocols (specific models targeted for product data exchange);
- Parts 301, etc., specify Abstract Test Suites (corresponding to the Application Protocol series);
- Parts 501, etc., specify Application Interpreted Constructs (interpreted models common to two or more Application Protocols).

The goal of STEP is to allow the exchange of data describing a product between Computer Aided systems (CAD, CAM, etc). STEP is based on Integrated Generic Resources, that are refined for different industrial areas as Application Protocols (AP). Those AP are first defined independently of STEP, according to the concepts of the specific industrial area they refer to, and are then implemented in STEP using the EXPRESS language.

EXPRESS [36] is thus a data modelling language standardized as the Part 11 of STEP. It consists of two different representations: textual, or graphical (called EXPRESS-G). However, EXPRESS-G is not able to represent all details that can be formulated in the textual form, on which we will concentrate in this part.

3.3.1 Implementation of the case study

It is possible to use the concepts behind EXPRESS in order to define product configuration models. The whole implementation of the case study in EXPRESS can be seen in Appendix A. This part describes relevant parts of the model.

EXPRESS supports object-oriented concepts: component types are defined in EXPRESS as *entities*, and they are composed by *attributes* that can be of basic types or entities themselves, representing partonomy relations in the model:

```
ENTITY Motherboard;
  rfCard: SET[0:1] OF RFCard;
  processor: Processor;
  gps: SET[0:1] OF GPSReceiver;
  wirelessChip: SET[0:3] OF WirelessChip;
  ethernet: EthernetCard;
  nbTouchChips: INTEGER;
  nbSlots: INTEGER;
  hasCompassChip: BOOLEAN;
  hasAccelerometer: BOOLEAN;
  ...
END_ENTITY;
```

As can be seen in the definition of the Device structure, the entities can be aggregated as *sets* (among other *lists*, *bags*, ...) with dynamic multiplicity, (`[0:3]` means 0-to-3 multiplicity).

Taxonomy relations are also available through the use of *inheritance* and *abstract* classes:

```
(* Abstract base component for storage drives *)
ENTITY StorageDrive ABSTRACT SUPERTYPE;
  capacity: GB;
END_ENTITY;

ENTITY SSD SUBTYPE OF (StorageDrive);
 ...
END_ENTITY;
```

These concepts permit to model the partonomy and taxonomy structures of component types in the model, though there is no way to define which component is the *root*.

It is also possible to declare *named types* and *units*, which help to clarify the meaning and the context of the variable of these types.

```
TYPE GB = INTEGER;
WHERE
  SELF >= 0;
END_TYPE;

TYPE SSD_TYPE = ENUMERATION OF (flash,DRAM);
END_TYPE;
```

Constraints can be declared both locally and globally in EXPRESS, though only local constraints are used in the implementation of the Mobile Device product family. These local constraints are written within the entities using the *WHERE* keyboard:

```
ENTITY Device;
  motherboard: Motherboard;
  screen: Screen;
  drive: StorageDrive;
  enclosure: Enclosure;
  miniUSBAdapter: MINIUSBADAPTER_TYPE;
WHERE
  ((motherboard.nbSlots = 6) AND (enclosure.size = 115x58)
      AND (screen.size = 3)) OR
  ((motherboard.nbSlots = 10) AND (enclosure.size = 240x190)
       AND (screen.size = 9)) OR
  ((motherboard.nbSlots = 12) AND (enclosure.size = 266x178)
       AND (screen.size = 10)) OR
  ((motherboard.nbSlots = 14) AND (enclosure.size = 295x210)
       AND (screen.size = 12));

         (TYPEOF(screen) = TouchScreen) = (motherboard.
            nbTouchChips > 0);
  ...
END_ENTITY;
```

As can be seen in those examples, one of the issues with writing constraints in EXPRESS concerns the support of *product catalogues*: as other languages shown in this Chapter, the only formulation available for product catalogs is using a heavy logical disjunction.

EXPRESS also lacks built-in functions like *sum*, or built-in navigation within collections (such as *forAll* or *forEach* functions). It is however possible for the user to declare his own functions that can be used in the entities, for example to calculate the actual number of slots made available by the motherboard:

```
FUNCTION SumChips(mb:Motherboard): INTEGER;
  LOCAL
    result: INTEGER := mb.nbTouchChips;
  END_LOCAL;
  IF (mb.hasAccelerometer)
          result := result + 1;
  END_IF;
  IF (mb.hasCompassChip)
          result := result + 1;
  END_IF;
  RETURN(result);
```

```
END_FUNCTION;

ENTITY Motherboard;
  ...
DERIVE
  nbChips : INTEGER:= SumChips(SELF);
END_ENTITY;
```

Being able to declare functions extends the possibilities in the model, but it requires some programming skills. Thus it should be reserved for complex functions, while functions like *sum* should be supported out-of-the-box.

Finally, other points need to be noticed:

- EXPRESS does not provide any support for *connection ports*: the one in the Mobile Device case study is defined in the Enclosure entity and declared as INVERSE reference in the TouchScreen. It however brings a relation of parent-child that should not be present in this kind of relations.
- EXPRESS does not provide any way to declare *resource* consumption and production.
- It is not possible to declare *read-only* attributes. On the other hand, *derived attributes* must be used as *hidden* attributes, though their value has to be directly associated to a function.
- Finally, EXPRESS does not support the declaration of *soft constraints*.

3.3.2 Discussion

EXPRESS is an interesting language for product modelling. It supports object-oriented concepts and a complex multiple inheritance mechanism, which makes it suitable for the modelling of most of the product configuration problem. This is complemented by the possibility to use a full procedural programming language to define functions and constraints on data instances. It also includes nice features such as units and constant declaration or dynamic multiplicity. Another point is the possibility to produce a graphical representation of the model defined in EXPRESS using EXPRESS-G. This can be a real advantage when modelling, and so is worth being pointed out.

However, the EXPRESS language seems too general and is not suitable for knowledge engineers that are not expert in the language itself. The definition

of functions **requires advanced programming skills**, and writing a complex model without these functions can be very difficult or impossible, as a lot of functions are not built-in (*min/max*, *sum*, *forAll*, ...). Other product configuration specific **features are also missing**, such as the declaration of a root element, read-only and default attributes, support for resources, soft and table constraints or connection ports definitions.

Moreover, the use of EXPRESS for product configuration modelling is restricted. The major problem comes from the STEP standard itself. EXPRESS is meant to be used to describe application protocols as part of the standard, and should thus not include company-specific data, such as company-specific configuration models. Männistö et al. describe in [63] a way to extend STEP in order to permit the definition of such models in STEP, where the AP would be used to declare the area-specific part of the model, while company-specific extensions would be added as instances of the AP concepts. However, this could reduce the freedom of the modeller, as he would have to comply to a basic structure for its model, according to the corresponding AP. Other issues arise as well, such as the definition of constraints in the model, as a STEP schema is supposed to describe only valid instances [63].

3.4 Comparison and Conclusion

Table 3.1 retains the main insights retrieved all along this chapter for the different modelling languages.

This table shows the differences between the languages, and first of all highlights the capabilities of the graphical languages, in both the clear view they provide and the completeness of their specifications. The difference of capabilities is greatly due to the fact that defining these languages with a lot of features is easy while interpreting them is more difficult. This is shown by the research done in the interpretation of UML-based model for product configuration [1, 2].

However, a language like UML is too general for configuration and its concepts too specific to software engineering. That is why SysML has been introduced, and it brings some clarity on product-related concepts, adding interesting features that can be used for product configuration modelling.

The advantage of textual languages is obviously that they can be directly interpreted as such. They also usually make possible to create a graphical representation on top of them, like EXPRESS-G. EXPRESS provides a full support of all object-oriented featured discussed in the requirements section.

Language	Type	Requirements	
		Features	General use
UML/OCL	Graphical + textual constraints	- Full OO support - Dynamic multiplicity - Extensible through stereotypes - Complete constraint language - Missing: units, table constraints	- Clear view of the structure - Need adaptation by the user using stereotypes - Structure and constraints are separated
SysML	Graphical + textual constraints	- Inherits features from UML - Mapping between blocks/constraints - Parametric constraints - Units - Some stereotypes still needed - Missing: table constraints	- Most of product oriented syntax already in place - Support for other product specifications apart from product configuration ones
EXPRESS	Textual + graphical representation	- Full OO support - Dynamic multiplicity - Units and constants declaration - User-defined functions - Lack of built-in functions - Missing: read-only and default attributes, table constraints, connection ports, resources,	- Possible graphical representation - Needs programming skills

Table 3.1: Comparison of the modelling languages

However, the definition of the constraints on the entities is made difficult by the lack of built-in functions or navigation within the aggregation collections. The possibility to declare functions using a full procedural programming language permits to do almost everything, but it then requires more advanced programming skills. Moreover, the use of EXPRESS for configuration is restricted because of its belonging to the STEP standard (see section 3.2.2).

None of the three languages compared here includes support for mapping structural component types and production attributes (BOM and operation routes). Although this feature may be specific to the underlying ERP or PDM software in charge of production management, it may be very useful in some cases, e.g., for standalone configuration system.

Using a general purpose modelling language for such a specific problem as product configuration has advantages, including the fact that most of these languages are well-known and have many interesting features. However, these languages are quite complex and very general, and it is not always an easy task to adapt them to product configuration and its specific concepts.

In this part, we have shown that, although these general languages have some interesting capabilities for modelling configurable product families, they are missing some important concepts, and languages like UML and SysML have limitations when it comes to dealing with modelling configuration-specific constraints or providing tool support. The next parts will thus introduce a new configuration framework supported by a configuration-specific modelling language that make use of the lessons learned from this study.

Part II

Heterogeneous Products

Chapter 4

Research Problem

This chapter introduces the Research Problem motivating our work in this part, with the following question: *how to uniformly support the configuration and management of heterogeneous product families?* What we refer to as a *heterogeneous product family* is a family of products integrating separate design disciplines interacting with each others. Such design disciplines involved in configurable product families can be, e.g., physical product design, software design, or service design. In the rest of this thesis, we refer to these design disciplines as the *dimensions* of a product family. Modelling configuration in such product families yields various issues, due to the diversity of the product knowledge necessary to address these different dimensions. Indeed, the model of a heterogeneous product family can be very complex, and involves several types of users with different skills and objectives. Finally, the different dimensions in such a product family are not independent, and it is primordial to take the interactions between dimensions into account.

We first introduce the literature background relevant to this part, and then we derive several Research Questions from our motivation problem.

4.1 Background

This section describes different areas of research on which this work is based, beside product configuration (Section 2.1). We start by providing an overview of related work on software product lines. Thereafter, we describe research on service configuration.

4.1.1 Software Product Lines

In this subsection, we briefly describe software product lines and some research on how to model them.

Software product lines (SPL), also known as software product families, is a set of software systems sharing a common set of features that "satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [24]. The concept of SPL has been proposed to improve software development and control the complexity and variability of software products. An SPL is based on a common software architecture and a common set of components implemented during a first phase called *development* [20]. This phase makes the core of the software system, which is then adapted to software products in a second phase called *deployment*. The deployment phase is aimed at deriving an individual product based on the software product line architecture that fits the requirement of a specific market or customer.

Several methods have been used to model efficiently software product lines, in particular two types of approaches, based on the modelling of the system's features or its architecture. The rest of this section gives an overview of some of the feature modelling approaches and modelling in software architecture.

4.1.1.1 Feature modelling

Feature modelling approaches are based on the concept of *features*, which has had several definitions over the years. The first feature modelling language, FODA (Feature-Oriented Domain Analysis) [53], defines a feature as "a prominent or distinctive user-visible aspect, quality, or characteristic of a system". Some other definitions have been given, including "an increment in program functionality" [12] or "a system property that is relevant to some stakeholder" [30].

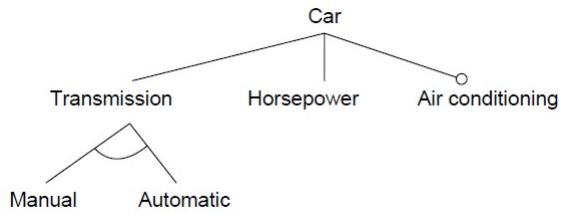


Figure 4.1: Example of a FODA model [53]

FODA is based on *feature models* that describe the different features of the software product line and the variability in it. An example of a FODA model can be seen in Figure 4.1. A feature model is a tree starting at a *root feature*. Each feature can have other features as *subfeatures*. In Figure 4.1, the root feature is *Car*, which has *Transmission*, *Horsepower* and *Air conditioning* as subfeatures. Moreover, *Manual* and *Automatic* are subfeatures of *Transmission*.

Variability is addressed using different relations between features and subfeatures, including *mandatory* subfeatures (must be selected if its parent is), *optional* subfeatures (may or may not be selected), or *alternative* subfeatures. An alternative subfeature consists of a set of features where one and only one feature must be selected if the parent is; *Manual* and *Automatic* are alternative subfeatures of *Transmission* in Figure 4.1. Feature models can be *configured* to represent individual products. In order to obtain a *valid configuration*, the subset of selected features must obey the rules defined in the feature model, i.e the different rules concerning which subfeatures are allowed to be selected or not.

An important number of extensions have been made to FODA. New feature modelling methods includes feature models formed as *directed acyclic graphs* [29, 54], where subfeatures can be shared by more than one parent. Subfeatures have also been given a *feature cardinality* [23] to specify how many times the subfeature may or must appear in a valid configuration. *Feature attributes* have also been added to feature models: a feature may have at most one attribute [30] or several [27]. Another improvement to feature models includes *feature groups* [28], which consists of a number of features (the *group cardinality*) that must be selected. In order to avoid confusion, subfeatures not members of any feature group are referred to as *solitary features*. Finally, Forfamel [7] describes a feature modelling approach synthesizing most of the previous approaches, along with type-instance semantics and subfeature names for more flexibility in establishing the different roles of the shared subfeatures.

4.1.1.2 Software architecture

Another type of approach considers software product lines from an architectural point of view. *Architecture description languages (ADLs)* have been proposed to describe the SPLs in terms of their structure, including their components, interfaces, or communication protocols. Despite the large number of existing ADLs, only a few have been developed to handle variability in SPLs.

Koala [105] relies on a model with *components* and *interfaces*. Components are the basic "encapsulated" blocks of software, and interfaces and can be used to provide or require different functions to/from different components. As with product configuration, partonomy and taxonomy structures can also be created. A component can contain one or more interfaces, specified using *interface types* described as "small sets of semantically related functions". Each interface thus has a type, a name, and a direction (required or provided), and interfaces with matching directions can be *connected* together. *Parameters* can be used in Koala as variability mechanisms. Those parameters influence the functionality of a system: it may for example affect which interfaces are connected. Interfaces can also be declared optional, i.e., the parent component may not have the interface in the final configuration. Figure 4.2 shows a Parking product modelled with Koala.

Koalish [8] extends Koala by adding more variability features such as attribute *domains* and *constraints*. It specifies the attributes and constraints of Koala components in a textual representation, as shown in Figure 4.3.

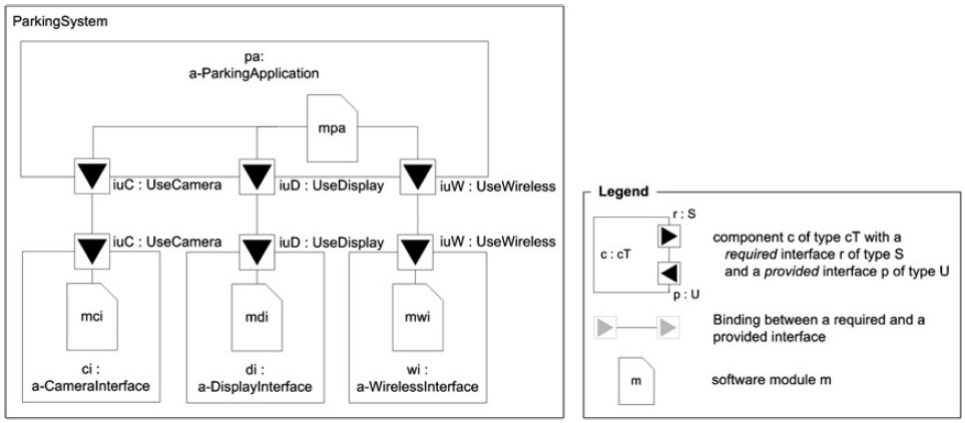


Figure 4.2: Parking model in Koala [91]

```
component ParkingSystem {
  contains
    component (a-ParkingApplication_WindowsCE,a-ParkingApplication_Symbian) pa;
    component DisplayInterface di;
    component (a-CameraInterface_iPaq,a-CameraInterface_Symbol,a-CameraInterface_P900) ci;
    component (a-WirelessInterface) wi[0-1];
  constraints
    ci instance of a-CameraInterface_iPaq => pa instance of a-ParkingApplication_WindowsCE;
    ci instance of a-CameraInterface_Symbol => pa instance of a-ParkingApplication_WindowsCE;
    ci instance of a-CameraInterface_P900 => pa instance of a-ParkingApplication_Symbian;
}
```

Figure 4.3: Koalish integration of variability in the Parking model from Figure 4.2 [91]

Other architecture description languages have been proposed, e.g., [42], although few contain concepts for variability. An exception is xADL 2.0 [31], which permits to define customised ADLs using the Extended Markup Language (XML). Basic modelling elements in xADL 2.0 are *component types*, *interface types*, and *connector types*. Those elements are organised in a partonomy and can be *optional* (it is not required to be included), *variant* (it represents a choice between two or more elements), or *optional variant*. Boolean expressions are used to define whether to include an optional element or which variant to select.

4.1.1.3 Multi-layer models

Several approaches have tried to bring together the two modelling methods described above [13, 18, 90]. In particular, Czarnecki et al. [26] illustrate a multi-level modelling approach with four layers: *feature configuration*, *analysis models*, *architecture and design models*‘, and *code*. Boolean conditions or OCL constraints can be used to relate elements in the different layers. Asikainen et al. created Kumbang [6] by merging two previous languages, Forfamel (feature modelling) and Koalish (Koala-based architecture description language). In Kumbang, the two different views are related using *implementation constraints*, defined in a specific constraint language.

Other methods have extended feature models to different levels. Czarnecki et al. [29] propose a specialisation of feature models using specialisation steps that includes refining feature and group cardinalities, removing/selecting a subfeature from a group, assigning an attribute value, or cloning a solitary feature. The aim of this approach is to perform staged configuration of feature models, i.e., to allow the specification of family members in several stages, where each stage eliminates some configuration choices. Reiser and Weber [82] also propose a multi-level approach. *Reference feature models* serve as a template and guideline for new models, called *referring models*. Reiser and Weber define several *deviations* representing the difference between a reference model and its referring model. One major difference between this approach and the one from

Czarnecki et al. is that in Reiser and Weber's work, feature models are not organised according to a specialisation hierarchy: in multi-level feature trees a valid product of a referring feature model does not necessarily have to be a valid product of the corresponding reference feature model. This aims at tackling for example the situation when changes are included locally in referring feature models before being pushed up the feature tree at a later time. Finally, Zaid et al. [111] propose a new modelling technique called *Feature Assembly Modelling* (FAM), based on the concept of *perspectives*. In order to improve the designing of feature models, this method advocates the separation of concerns approach via proposing different perspectives, e.g., *System* perspective, *Users* perspective, or *Functional* perspective. Aside from composition and specialisation relations, classic dependencies relations (e.g., requires, excludes, incompatible) are used to connect the different features, whether or not they belong to the same perspective.

4.1.2 Service configuration

Services are usually defined as products with a significant service dimension [56]. *Configurable services* represent services that can be customised from a set of predefined options, in order to fit the needs of individual customers. Research on configurable services and how to model them for configuration is a relatively recent topic, with limited work available. Several authors have been discussing the configuration of different types of services, including IT services [19], insurance [108], or maintenance services [65]. Other researchers [3, 49, 98, 107] proposed more detailed conceptualisations for modelling services.

Akkermans et al. [3] present a model using three perspectives: *value*, *offering*, and *process*. The value perspective represents what the customer needs, the offering perspective what the supplier is providing, while the process perspective describes the realisation of the service. Heiskala et al. [49] propose a conceptual model with four viewpoints, called worlds (Figure 4.4): the *needs world*, representing the customer's needs; the *service solutions world*, for the service's specifications; the *process world*, related to the service delivery; and finally the *object-of-services world*, that is used to describe the service recipient and the environment in which the service will be supplied.

Although the first three worlds are similar to the perspectives from the work in [3], the last world can be beneficial as the characteristics of maintained equipment or the service's environment may affect the service definition and should thus be included in the model. Each world is organised according to different types: the need type from the needs world denotes a benefit that a customer is looking for in the service; the service element type from the service solution

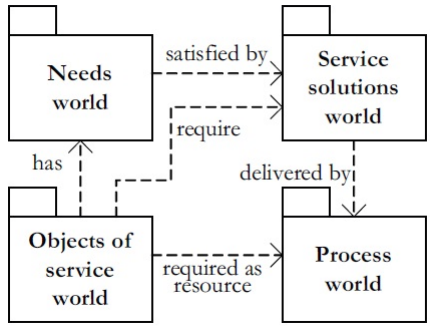


Figure 4.4: Overview of Four-Worlds model from [49]

world is used to describe the agreement about what is to be delivered by the service; the service object type from the objects-of-service that describes the recipient of the service, or the environment relevant to that recipient; and the process module types that represent tasks carried out as part of the service delivery process. These types can have valued attributes, and are organised in the different worlds using partonomy and taxonomy relations, while constraints specify the dependencies between the types that must hold to obtain a valid configuration.

4.2 Research Questions

Based on the requirements from Part I, we specify in this section the main research problem in more detail with the following Research Questions:

RQ1 *What needs of the model's user should be supported?*

RQ2 *What modelling constructs support addressing the heterogeneity?*

RQ3 *How to integrate the different dimensions of heterogeneity in the models?*

RQ4 *How to support the management of product families over time and for different market situations?*

In the rest of this part, we extend the motivation example presented in Section 2.2. We present in Figure 4.5 a simplified version of the Mobile Device case study.

It represents a configurable family of products consisting of:

- A set of physical elements, including a keyboard, a screen, a battery or a motherboard for example. This part represents all the components of the products that are manufactured, and that will be translated into BOMs and operations routes.
- The configurable software running on the devices. This includes the different applications (e.g., phone, photo, or map applications) that are to be deployed on the devices, as well as the libraries that will support them. Deployment of such elements involves installing software packages in a specific sequence.
- The services associated with the devices. The product family can be sold with different services, including phone and data subscriptions, different types of repair schemes, or synchronisation services.

Already this rather small case study illustrates how complex the modelling of a heterogeneous product family can be. Indeed, the engineers responsible for modelling the variability in the physical parts of the system often possess knowledge different from the ones responsible for managing the software configuration model, or creating the service model. As can be seen in Sections 2.1 and 4.1, although the modelling approaches often use the same basis (types, partonomy, etc.), the high-level concepts behind each type of modelling are different, and thus require different mindsets. One can therefore assume that the task of managing the variability of the hardware, software and service parts for large products is delegated to separate groups of knowledge engineers.

Configuring such a product family can be quite complex, due to the amount of technical details represented in each different aspect of the products. Those details are often not easily accessible to sales persons and end-users, who prefer viewing the features (or functions) of product families, as described in [93]. Defining the feature set of the product family may be enough in some cases. However, we identified several scenarios that illustrate different situations where this feature set may need refinement:

- *Market differentiation*: The company selling the products proposes different feature sets for different markets. In our example, different markets, e.g., Europe and United States, requires different data signals to be handled by the phones, as well as different regulations. The possible combinations of features may just be restricted on those different markets.
- *Feature set evolution*: The product family's feature set is evolving with time. Devices may not arrive fully featured on the market, due to time

Figure 4.5: Overview of the extension of the Mobile Device case study. The product family represents mobile devices and is configured according to three dimensions, hardware, software and services, and needs to be adapted to different scenarios, e.g., tailored to distributors and markets.

constraints or strategic decisions. A refined feature set may then be needed for a specific time, with additional constraints that may disappear (or be modified) in future evolution of the product family.

- *Distributors tailoring*: The producing company is distributing the products to different intermediary vendors. Products like the Mobile Device may not be distributed directly by the manufacturer. This producer may propose a feature set to vendors that can adapt it in order to forbid specific combinations, or to create a more simple feature set for the end-customer. For example, the example products may be sold by distributors by letting the customer choose between different feature packages, limiting the choices in configuration.

- *Market analysis*: The end-users can also be considered beforehand (instead of the product family). A market study identifies the different needs of the customers (or needs that the company wants to introduce in the market) which are used to identify different feature sets to satisfy these needs, aiming at creating a product family to fit those. In contrast to the first three scenarios, this scenario considers the market needs as the basis for designing the product family.

These scenarios provide a more concrete characterisation of how the functionalities of the product family may need to evolve depending on its use and distribution, as introduced in Research Question 4.

Chapter 5

Conceptual Modelling Framework

In this chapter we present a framework for modelling heterogeneous product families. This framework synthesizes, unifies, and extends different approaches to modelling configuration in the different design disciplines, e.g., physical products, software, or services. We first give an overview of the approach, and then detail the different conceptual elements and the dependencies between them. The framework and related work are discussed at the end of the chapter.

5.1 Overview of the Approach

Our approach is based on the concept of *modelling views*. Those views are used to model different aspects of the product family, according to the different roles of the modellers. The main assumption is that each product family considered consists of different dimensions, and that all these dimensions need (and benefit from) configuration. An overview of the basic concepts can be seen in Figure 5.1.

Configuration models are often created and maintained by knowledge engineers based on various details provided by domain experts. However, heterogeneous

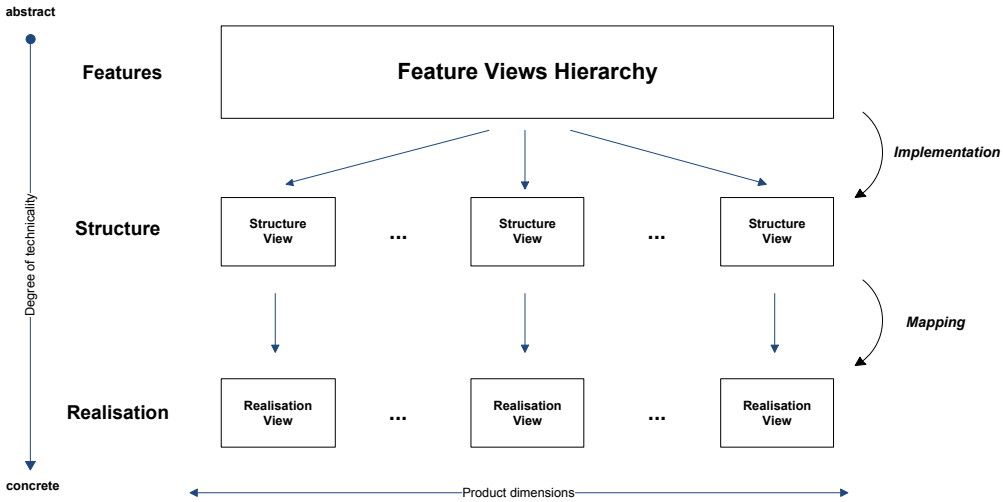


Figure 5.1: Overview of the conceptual framework. Realisation and structure views are specific to each dimension of the product family, while feature views are organised in a feature views hierarchy (Figure 5.2) and represent the whole product family.

product families with multiple dimensions may require different kinds of domain experts with different roles and sets of skills, according to the degree of technicality or the dimension considered. That is why our framework is based on different types of views: *feature views*, *structure views* and *realisation views*.

Feature views provide a view of a product family from a high level of abstraction. Technical details of a product family are not suitable for all types of users: feature views are therefore used to express the different functionalities of the product family. These views are targeted at sales persons or end-users that need to have an understanding of *what* the product can do, instead of *how* they can do it. The aim is indeed to differentiate between what functionalities the end-user needs from the product family versus how those needs can technically be achieved. In our conceptual approach, feature views are not separated according to the different dimensions of the product family. The relations between the concepts described in feature views are related to the product as a whole, and as such should not be dimension-specific. Products can indeed be characterised by the features (or functions) they provide, independently from the way they are structured.

Structure views define the different design components that implement the described features of the product family, and the relations between them. Structure-

based approaches for configuration are widely used [93], as the compositional structure of the product families is often used to represent the product data knowledge. The structure views communicate the aspects of the architecture of interest to those involved in designing the system. They provide more concrete models of a product family, as they represent the specifications of the components of the system. Structure views are thus mainly aimed at design or maintenance, and are for example targeted at product design engineers, software architects, or service contractors.

Realisation views offer a detailed technical view of how the products are realised. Compared to structure views, whose purpose is to represent the design of a specific dimension of the product family, realisation views are aimed at describing the elements necessary for the concrete realisation of the system for that dimension. They are thus targeted at highly specialised engineers, e.g., product engineers, software developers, or service deliverers, and represent the lowest abstraction level in our conceptual configuration framework. Each realisation view is associated with a dimension, which defines its proper meaning: physical products use this view to represent manufacturing data, while software involves the solution deployment, and services the delivery process.

While feature views are related to the whole product, different structure and realisation views must be considered for each dimension, providing a *two dimensional modelling* of the product family, according to the degree of technicality of the view (structure versus realisation view) and the domain of expertise (depending on the dimension considered). This also means that structure and realisation views from different dimensions do not influence each other. This is motivated by the fact that each type of view is handled by engineers with different roles and skills, e.g., the architecture of the software should be independent from the structure of the physical product, although some elements may be combined to provide specific features in the feature views.

Feature views are related to structure views using *implementation*. This interaction is primordial as it models the dependencies between the feature view and the different structure views. The implementation of a specific feature in the model has indeed to be provided by the structural capabilities of the product family, whether it comes from a specific dimension or a combination of factors from different dimensions. Moreover, structure views and realisation views are related using *mappings*. Those mechanisms are described in more details in the next sections.

In order to address with our approach the different scenarios discussed in the previous chapter, feature views are organised in a specialisation hierarchy called a *feature views hierarchy* (Figure 5.2). Each model defines a *base feature view*, which will contain all the features available for the modelled product family,

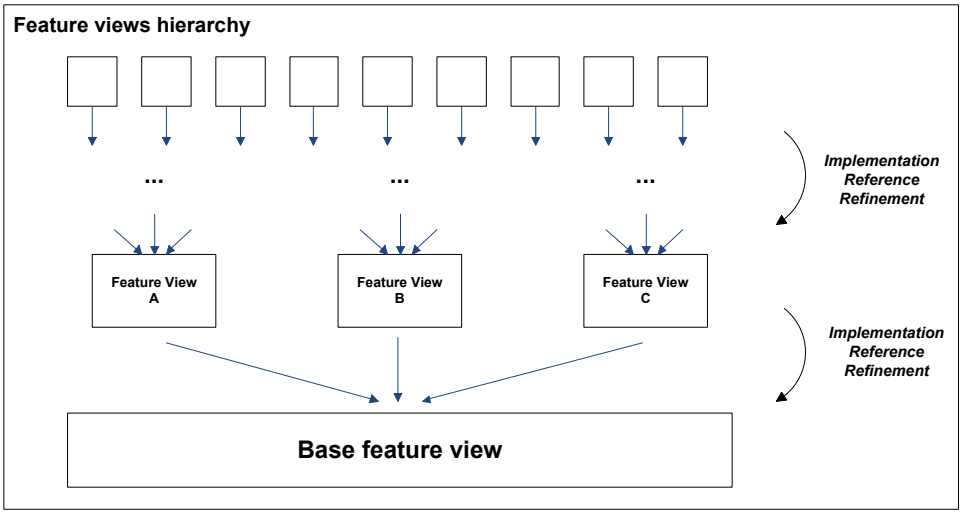


Figure 5.2: Feature views hierarchy

and should be implemented by the structural views. This view can then be refined in specialised feature views, in order to model specific sets of products. In contrast to the structural and realisation views organisation, the feature views hierarchy is a dynamic hierarchy, in the sense that feature views can be added or removed in that hierarchy in order to fit specific needs. The scenarios defined previously can be supported by the feature views hierarchy by using a fully featured base view and specialised views for different markets (*Markets differentiation* scenario), for each evolution of the product family (*Feature set evolution* scenario), or for distributors' own feature sets (*Distributors tailoring* scenario). The last scenario (*Market analysis* scenario) can also be supported in a top-down approach: higher level feature views can be built according to a market study; those views can then be merged into one base feature view, from which the company can see whether all necessary features can be implemented by its current offering, or whether some changes need to be done. Feature views in the feature views hierarchy are related by *specialisation operations* (refinement or reference) or by implementation.

In the next section, the different views are described in more details, illustrated by our case study example. Section 5.3 then addresses the different types of dependencies and constraints used in the framework and their semantics. Finally, we detail in Section 5.4 the mechanisms used to relate feature views and types in the feature views hierarchy. We refer to the case study extended in Section 4.2 for illustrative examples.

5.2 Description of the Views

Modelling views fit different modelling purposes, and are therefore split into *feature views*, *structure views*, and *realisation views*. Each type of view is characterised by a set of concepts with a specific organisation. Modelling views can also interact with other views in various ways, depending on their types.

Each view follows a type-instance approach. Types represent classes of distinguishable entities in the modelled product family, and serve as a description of its individuals, or *instances*. Each of the following subsections describes one type of view in detail, defining the associated concepts using UML meta models and how they are used when modelling the different product dimensions.

5.2.1 Feature Views

Feature views provide a view of a the functionalities of a product family. The UML meta model defining the concepts used in feature views can be seen in Figure 5.3, while Figure 5.4 shows the feature types of the base feature view of our case study. UML is used to represent the case study, although no constraints are shown here. Note also that the model of the Device Mobile product family presented in the rest of this chapter are subparts of a larger model available used in Chapter 11 and available at [74].

A feature view is composed of *feature types*. Individuals instantiated by those types represent features that a product family can provide. Adapting the definition from [93], we define a feature as "*an increment in the system functionality that the product provides to the customer, the user of the product, or the environment in which the product instance will be situated*".

Feature types define a partonomy structure: each type can define one or several *subfeature* relationships with other feature types. A subfeature is a decomposition of a feature type, and provides a part of the functionality that its parent is defined for. Subfeature relations have a cardinality, defining the number of feature instances that are involved in each relation. Several feature types can thus be grouped together to express the fact that they are providing similar functionalities or are part of the functionalities of a larger feature type, in an incremental way. Subfeature relations can also be used to model optional feature or feature group as in [29], expressing a choice over different subfeatures. This also permits a better model maintainability, as the model becomes more modular, and the parent feature type (i.e., the "head" of a subfeature relation) can be reused as the subfeature of different feature types, limiting the number

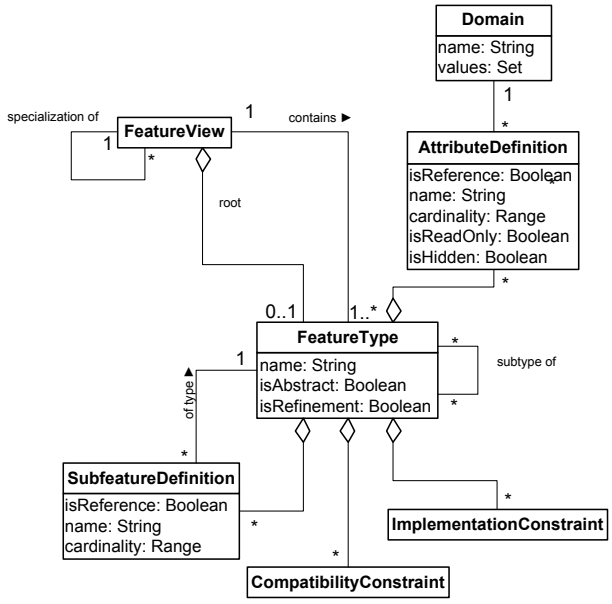


Figure 5.3: UML meta model for feature views

of relations declared without limiting the expressiveness of the model.

Example 1 *Consider the base feature view of our case study (Figure 5.4). Mobile Devices can propose features such as device localisation, telephony capabilities or different types of input and display. We thus define feature types such as* Localisation *or* Input*. The* Phone *feature type is a subfeature of* Communication*, and may not be used during configuration, as its cardinality is 0 or 1.*

The feature types in a feature view are also organised in a taxonomy: each type can be related to other types via a *kind-of* relation. In such a relation, there is a *subtype* and a *supertype*. Individuals instantiated from subtypes also contain the properties of their supertypes and their constraints. Types can have multiple subtypes and multiple supertypes, provided that the names given to each supertype's properties do not collide with the properties of the others supertypes. This means that the instance of the subtype inherits from its supertypes, e.g., when it provides functionalities derived from all of them. Finally, a type may be declared *abstract*, in which case instances of this type must be assigned one of its concrete subtypes during configuration.

Attributes are valued properties of a feature instance. They represent the characteristics of a feature type, i.e., some variable information that must be chosen

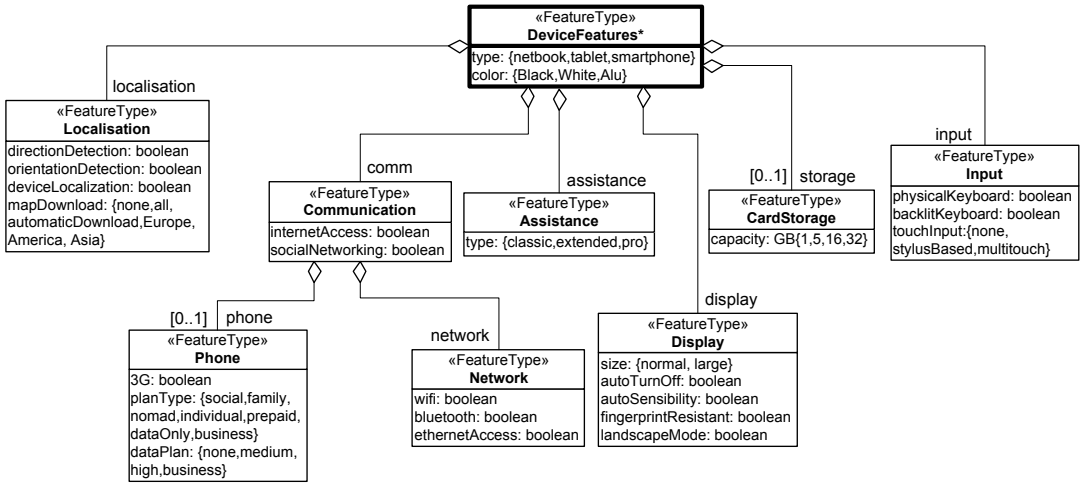


Figure 5.4: Types structure of the base feature view for the Mobile Device product family, presented in UML notation. UML Aggregations are used to represent subfeature relationships, including names and cardinalities. The root feature type is designated by a star(*) and thick lines. Compatibility constraints are not shown here.

a value during configuration. Each feature type can define attributes using *attribute definitions*. Those definitions define the name of the attribute, and the number of elements it refers to, using a *cardinality*. Each attribute can take different values, which are specified in a named *domain*. An attribute can also be defined as *read-only* or *hidden*. If an attribute is read-only, it will not be modifiable during configuration by an end-user. In case it is hidden, it is not even visible. These options are useful when providing information to the end-user or declaring attributes for computational purposes and ease of modelling. Attributes and subfeatures are from now on referred as the *properties* of a feature type.

Example 2 *The feature type* Display *has an attribute* size*, which represents whether the display of the mobile device should be of normal or large size. The* Input *type also contains attributes, for example, to specify whether the input should be touch-based, and whether the device should have a backlit keyboard.*

Types can define *compatibility constraints* and *implementation constraints*. These constraints permit to model the dependencies within feature views and with other views as well. Dependencies and constraints will be described in more details in Section 5.3.

Semantics. In a valid configuration, there must exist at most one *root* feature type, which would be the root of the partonomy tree. However, due to the presence of a feature views hierarchy and the potential specialisation operations between the feature views (see Section 5.4), it is possible that no root is defined, in which case the root feature type is assumed to be the root feature type of the immediate parent feature view in the hierarchy. A special case is the base feature view, which must declare a root feature type, as it is at the root of the feature views hierarchy. Moreover, there should be no cycle in the partonomy and taxonomy, as those relations are hierarchical and form a tree. The number of instances of feature types defined as subfeatures should satisfy the cardinality of the subfeature relations they are declared in. Finally, a valid instance of a feature type contains all the attributes of its type and its supertype; each attribute must have a value that has its declared type, and which is chosen within its declared domain.

5.2.2 Structure Views

Structure views define the different design components that realise the described features of the product family. The UML meta model defining the concepts used in structure views can be seen in Figure 5.5, while Figure 5.6 shows the three structure views for our case study, corresponding to the physical, software, and service dimensions, in UML representation. It is worth pointing out that, although the UML representation of the physical structure view is very similar to the UML model seen in Chapter 3 (Figure 3.3), the model shown here is merely using UML symbols to represent the concepts defined in our framework, while the model from Chapter 3 used the actual UML concepts to represent the product family.

The basic building blocks in structure views are *structure types*, which can be either *component types* or *association types*. Each structure type is associated to a structure view.

Instances of component types represent structural elements of the system, and can be composed of *subcomponents*, thanks to partonomy relations. Each structure view contains one *root component type*, which becomes the component type from which the partonomy starts. Subcomponent relations are also associated a cardinality, that may be static (a specific number of subcomponents is specified) or dynamic (a range of possible cardinalities is declared). Note that the component types involved in a subcomponent relationship must be associated with the same structure view: this prevents that the partonomy of a view extends to another structure view describing another dimension. Also, as in feature views, no cycle can exist in the partonomy.

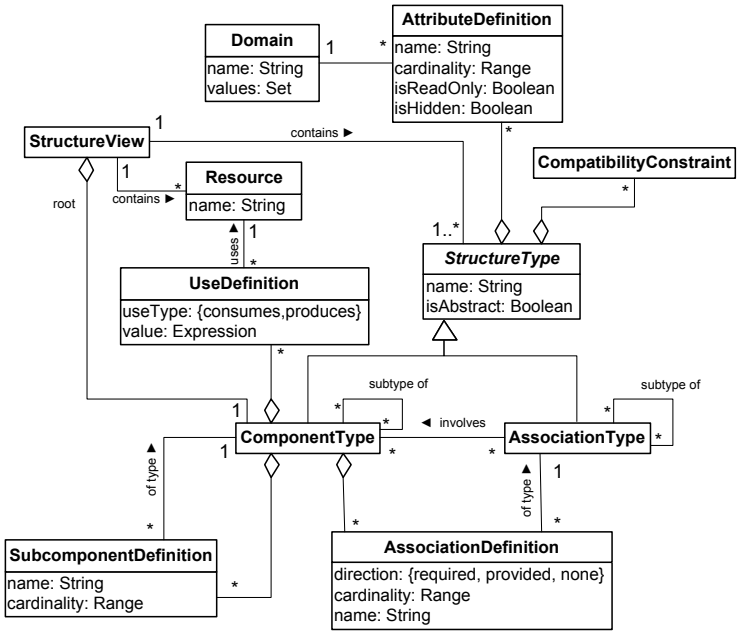


Figure 5.5: UML meta model for structure views

Association types represent types that can be involved in an *association* relation. Associations are non-hierarchical relationships between components. A component type can define an association of a specific association type. This association type can be bound to a specific number of component types that are allowed to use it. Association relations also define the direction of the relation: a component type defining an association can either *provide* a connection point, *require* one, or simply be involved in a *non-directional* relationship. As for subcomponent definitions, cardinalities can be used to define the number of individuals potentially involved in the association definition. Component types defining the same type of association can be *connected* if the directions match (one is required and the other one is provided, or both have no direction).

A taxonomy structure can also be created, as structure types can be *subtypes* of other structure types, provided that they are of the same nature (component or association types). As with feature types, the subtype of a structure type contains all the latter's properties, and may define additional ones.

The different concepts used in structure views have a specific meaning according to which dimension each view refers to. For example, a *physical structure view* represents the physical structure of the product family. Component types

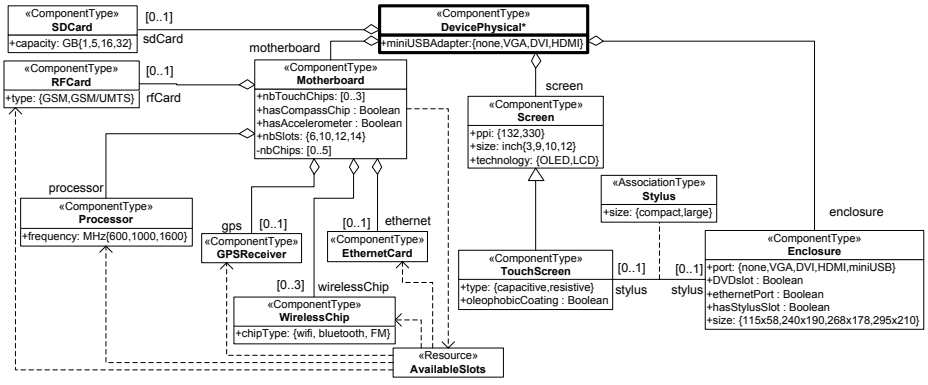
are entities whose individuals are physical components involved in the physical design, while associations are used to model non-directional physical links between two components. A *software structure view* describes the architecture of the software system involved in the product family. Instances of component types represent software components, and associations can be defined to model interfaces, whether they provide software functions or require some. Also, a *service structure view* describes the specifications of the service to be delivered. Component types are service element types, and describe contractual agreements of what to be delivered, similar to what is modelled in the service solutions world of Heiskala et al. [49].

Example 3 *The Mobile Device product family (Figure 5.6) is composed of devices that include for example a screen and a motherboard. Subcomponent definitions are represented using UML Aggregations, and generalisation are represented using UML Generalisations. The root component types are designated by a star(*) and thick lines. In the physical structure view, association definitions are shown using UML Associations. Resource use definitions are shown as dashed arrows, their use types symbolised by the directions of the arrows. In the software structure view, association definitions are shown using UML* Interface *(provided) and UML* Socket *(required), depending on the direction of the association.*

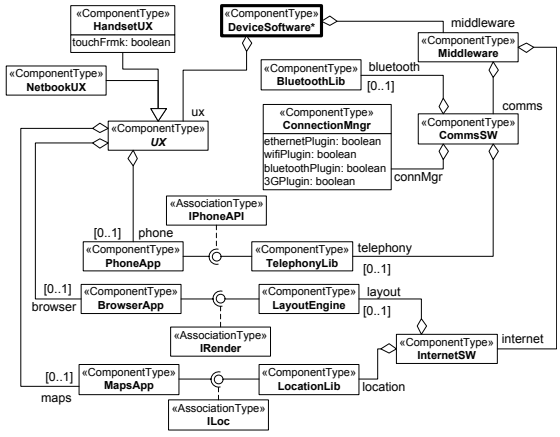
The WirelessChip *component type represents different chips that can be installed on the motherboard: it is thus involved in a subcomponent relations with the* Motherboard *component type, with a cardinality of 0 to 3, i.e., up to 3 instances of WirelessChip can be present in a valid configuration. The* Stylus *association type models a stylus: it has no specific parent in the partonomy, but is associated with the* TouchScreen *type (which may need a stylus) and the* Enclosure *of the device (which may have a specific slot to store the stylus).*

The software running on the devices is composed by several layers, including two subject to configuration: the middleware layer and the User Experience layer (UX). The UX layer describes the different applications that can be installed, while the middleware layer relates to the libraries behind those applications. Other layers, such as the software kernel, are not configurable, and thus do not appear in the configuration model. The UX *component type is abstract, and has to be specialised as a HandsetUX or NetbookUX during configuration time. Also, the* ILoc *interface is modelled as an association type between the library* LocationLib *that provides it and the* MapsApp *application that requires it.*

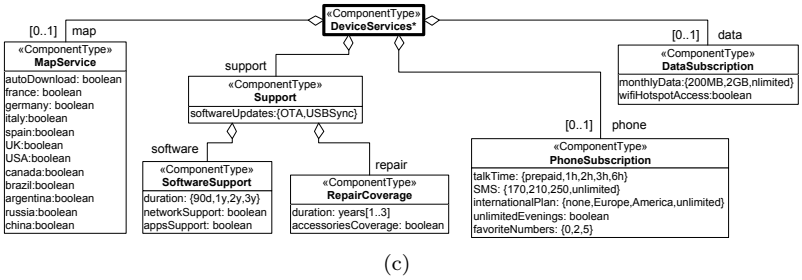
Finally, the devices come with different services, including phone and data subscriptions, after-sales support, and map downloading services. The service structure view thus contains component types such as PhoneSubscription*, that may appear in the service contract if the device is sold with a phone subscription.*



(a)



(b)



(c)

Figure 5.6: Structure views of the Mobile Device product family. (a) Physical structure view. (b) Software structure view. (c) Services structure view.

Structure types can have attributes and compatibility constraints, in the same way as feature types. Those attributes represent valued characteristics of the instances, and the compatibility constraints describe the dependencies within the structure view (detailed in Section 5.3). Attributes, subcomponent, and association relations will from now on be referred to as the *properties* of a structure type.

Example 4 *The* SoftwareSupport *component type in the service structure view (Figure 5.6(c)) has a* duration *attribute (which ranges from 90 days to 3 years), while the* Stylus *association type in the physical structure view may have a compact or large value for its* size.

Finally, *resources* can be declared in structure views, and their production and consumption by component types is declared by *use definitions*. Resources can model any quantifiable entity that needs to be (at least) balanced in the system.

The value of a resource use definition is specified with the same kind of expression as compatibility constraints (Section 5.3) and must evaluate to a quantity.

Example 5 *Resources make the most sense in physical structure views. The* AvailableSlot *resource in the Mobile Device product family represents the number of chip slots available on the motherboard: it is produced by the motherboard itself, depending on the size made available within the enclosure, and is consumed by each chip installed on the motherboard.*

Semantics. In a valid configuration, there must exist exactly one root component type. No cycle cannot appear in the partonomy and taxonomy trees. The number of instances of component types defined as subfeatures should satisfy the cardinality of the subfeature relations they are declared in.

Moreover, each association instance present in the configuration should be connected to component types matching the direction of their association definitions (if a definition is provided).

Resources must also be balanced: the difference between all the productions from the different type instances and all the consumptions must be greater or equal to 0 in a valid configuration. Finally, as for feature views, a valid instance of a structure type contains all the attributes of its type and its supertype; each attribute must have a value that has its declared type, and that is chosen within the declared domain.

5.2.3 Realisation Views

How each product family design is mapped to production data depends often on the backend of the configuration tool used (i.e., whether the tool is standalone, or integrated in an ERP system, etc).

However, we define here a model for realisation views aiming at providing the base for a common conceptualisation of the realisation phase in configuration. The UML meta model defining the concepts used in realisation views can be seen in Figure 5.7.

The building blocks of a realisation view are *realisation types*. There are three possible realisation types: *item types*, *operation types*, and *resource types*. Each realisation view is associated with a structure view (representing the same dimension), and this interaction is responsible for the presence of structure types in the meta model, although they do not belong to the realisation view.

Item types represent the production components used to realise products. This can be a BOM item for manufactured parts, a software package when dealing with software, or an object to be produced when delivering a service (e.g., a contract or a bill). Contrary to structure views, realisation views are not starting with a single root type. Instead, the structure and realisation views are associated via a mapping between the structure types and the corresponding item and operation types necessary to realise the product.

A *mapping condition* is specified to determine whether an item (or operation)

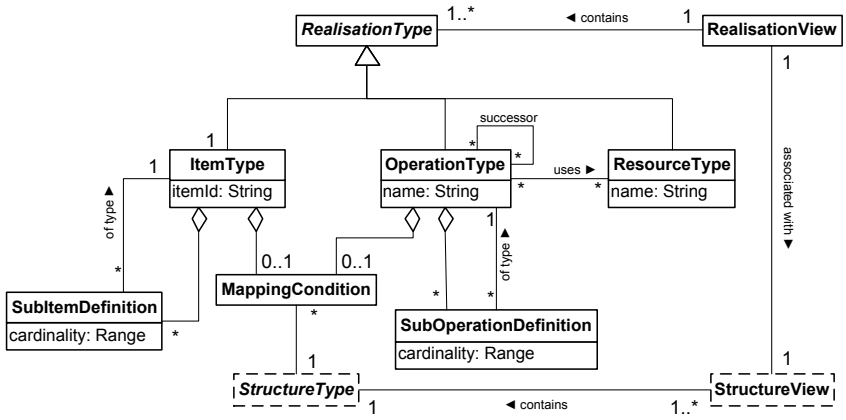


Figure 5.7: UML meta model for realisation views

type is to be produced or not. Mapping conditions are written as constraints, and will be described in Section 5.3. Since structure views provide a higher level of abstraction of the product family, there are often much more item types than structure types. Each item type mapped to a structure type thus defines its own tree of *subitem* types, providing a more detailed breakdown of the actual production components.

Example 6 *Figure 5.8 shows a part of the realisation view for the physical dimension of our case study example. It represents the realisation of the Screen and TouchScreen structure types. The Screen type can be mapped to different items, depending on the mapping conditions specified in the relationships. These conditions may involve elements from the structure view, such as attributes values or any other properties.*

For example, if the individual is an instance of the TouchScreen *type, the BOM may be composed by the item #8920, which is in turn composed of two items #235 and two items #239.*

Operation types are used to specify a set of operations needed during the production of configured products. As for item types, a mapping is done with structure types, and *suboperations* can be defined.

Different production operations must often be performed in a specific order. Operation types can thus be linked by *successor relations*. In case a successor relationship is defined between two operation types, the operation individuals instantiated must respect the order in which they are carried out if they are both involved in the realisation process. The operation types correspond for example to sequences of manufacturing operations for physical parts, deployment actions (with dependencies) for software, and delivery processes for services.

Example 7 *The* Coating *operation (for applying coating to the screen) is specific to instances of the TouchScreen type. Also, the different operations are ordered, in such a way that the thermal tempering must be done before the coating (if needed), and before the assembling. This is defined using the successor relation between the* ThermalTempering *and the* Assemble *types.*

Finally, resource types can be defined to represent resources used in the operations. Those resource types may describe a machine, an operator, some information, or anything that may be necessary to complete the operations.

Example 8 *The* Coating *operation type uses the* CoatingInjector *resource type, as an injector is need during the coating of a touch screen.*

Semantics. In a valid configuration, for each item or operation type, the following applies:

1. If the type is mapped to a structure type via a mapping condition, an instance of the realisation type should be present for each instance of the structure type for which the mapping condition is true.
2. If not, an instance (or more, depending on the cardinalities involved) of the realisation type should be present if and only if it is involved in a subitem (or suboperation) relation and its parent is present.

Moreover, an instance of a resource type should be present for each instance of operation type that uses the resource type. Finally, the list of operation instances should be ordered in according to the successor relation defined.

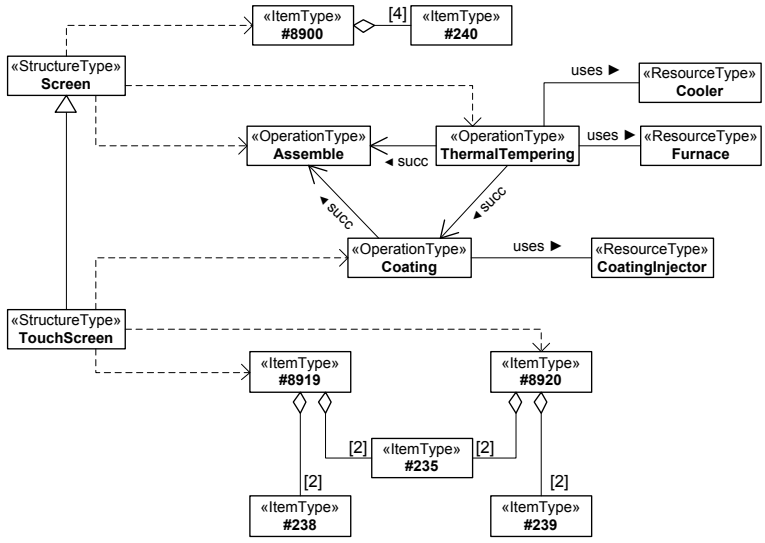


Figure 5.8: Part of the physical realisation view for the running example, presented in UML notation. Dashed arrows show the mapping between structure types and item/operation types (associated conditions are not shown here). Subitem relations are represented using UML Aggregations. The Screen and TouchScreen structure types are shown here to illustrate the mapping relations, although they are not part of the realisation view.

5.3 Dependencies Within and Between Views

Attributes and other properties of types in feature and structure views handle the variability in the product families. It is however necessary to restrict the combination of values available in order to obtain a model that fits the actual product family. Moreover, modelling heterogeneous product families is more than modelling its different dimensions. These dimensions interact with each other, and modelling those dependencies is essential in order to have a complete and consistent modelling framework. Constraints can be used to specify dependencies within or between views when other modelling mechanisms are not sufficient to capture them.

A constraint is a Boolean condition relating different elements of the model. We define different types of constraints in our conceptual framework: *compatibility constraints*, which aim at ensuring the consistency of a structure or feature view independent from each other; *mapping constraints*, which define whether individuals from the realisation views should be included or not in the final configuration output; and *implementation constraints*, which are used to describe the relations between the base feature view and the structure views, or between two feature views related in the feature views hierarchy.

This section describes the different types of constraints. Examples of constraints are given in a constraint language for illustrative purposes: this constraint language will be given in more detail in the next chapter.

5.3.1 Compatibility Constraints

Compatibility constraints are fundamental in the modelling of the feature views and the structure views, independently from the dimension considered, as they remove possible inconsistencies between different properties of the instances during configuration.

A compatibility constraint is specific to a particular view, and can only involve properties of this view. As it models the compatibility of the different elements in the system, such a constraint must be evaluated to true in a valid configuration of the product family. A compatibility constraint belongs to a specific type: this type is called the *context* of the constraint.

Example 9 *Consider the physical structure view in Figure 5.6(a). In the context of the* DevicePhysical *type, the following constraint specifies that the enclosure of a device must contain a slot for an Ethernet port if and only if the*

motherboard contains an Ethernet card.

$$enclosure.ethernetPort = true \Leftrightarrow Count(motherboard.ethernetCard) = 1$$

Properties of types are referenced using qualified names by navigating from the context using a dot-notation. The constraint is an equivalence, specifying that the ethernetPort *attribute defined in the* enclosure *subpart of instances of* DevicePhysical *must be true if and only if the cardinality of the subpart definition named* ethernetCard *in the* motherboard *subpart of* DevicePhysical *is equal to 1.*

Constraints may contain references to properties that are not always present in the product instance being configured. This may occur if a constraint accesses a subpart whose cardinality is not fixed, or an attribute from a subtype that may not be chosen. In that case, we say that the term referring to this property (attribute, subpart, subfeature, ...) may be *inactive*.

Example 10 *Consider the base feature view in Figure 5.4. The following constraint, declared in the* Communication *feature type, specifies a phone feature with 3G implies that the device has internet access:*

$$phone.3G = true \Rightarrow internetAccess = true$$

The term phone.3G *may be inactive during configuration, as the cardinality of the* phone *subfeature in* Communication *may be equal to 0. In such a case, the above constraint would be directly evaluated to true.*

Semantics. The evaluation of a constraint occurs during configuration, when types are instantiated to individuals. Each instance of the context type in which the constraint is declared must satisfy it. Also, each compatibility constraint containing at least one inactive term is evaluated to *true* during configuration.

5.3.2 Implementation Constraints

The types defined in the base feature view represent all the functional characteristics of the system, and the relations between those abstract concepts and the more concrete ones, the structural elements of the product family, have to be defined. Feature views use the concept of *implementation*. The implementation

of a feature type is done using *implementation constraints*. Implementation constraints are essential to our framework, as they model the interactions between the base feature view and the structure views, as well as between views in the feature views hierarchy (Section 5.4). These constraints involve the properties of the feature type they are declared in, as well as properties of the types from the different *structure* views.

Implementation constraints model the interdependencies between a *child* view and one or more *parent* views. These constraints may involve the base feature view (child) and the structure views (parents), or a feature view and its parent in the feature views hierarchy.

Implementation constraints are always defined in the child view. The main challenge with these constraints is to express what is the impact of features from the child view on the elements of the parent view(s), e.g., what structural elements can implement the desired feature. In order to reflect this, implementation constraints are composed of two expressions C and P involved in an implication, an equivalence, or an equality constraint:

$$C \Rightarrow P$$
$$C \Leftrightarrow P$$
$$C = P$$

The expression C represents the features to be implemented by the constraint. In a similar way as in the compatibility constraints, it takes the form of a relation between properties from the child view, using the context of the feature type declaring the constraint.

On the other hand, the expression P represents what is needed in the parent view(s) to implement the features specified by C. Each term in P specifies a local context defined by a dimension (if there are several parent views, from different dimensions), and a type, independently of the global context of the implementation constraint. For example, the term `Physical::TouchScreen.type` refers to the *type* attribute of the *TouchScreen* type in the physical structure view.

Semantics. By default, a relation involving a term defined using a local context with type T is satisfied if there exists one individual of T satisfying it. Existential quantifiers are implicitly used in the semantics of the right hand side expression P, as a feature may exist if there is at least one combination of structural elements implementing it. This permits to model the fact that a feature is actually implemented if a combination of relevant factors are fulfilled.

Example 11 *Consider the following constraint from the base feature view (Figure 5.4), with* Input *as context feature type:*

$$touchInput = multitouch \Leftrightarrow (Physical :: TouchScreen.type = capacitive \\ \wedge\ Software :: HandsetUX.touchFrmk = true)$$

This implementation constraint specifies that a device has a multitouch input if and only if there exists *a capacitive touchscreen and a touch framework is implemented in the software.*

The *present()* function can also be used to assess whether a specific type exists in the parent view.

Example 12 *Consider the following constraint in the* Localisation *feature type:*

$$deviceLocalisation \Leftrightarrow \\ present(Physical :: GPSReceiver) \wedge present(Software :: LocationLib)$$

This constraint ensures that it is necessary to have both a physical GPS receiver and a software location library to guarantee the device localisation feature.

Finally, implementation constraints handle inactive terms in two different ways, depending on which part of the expression they are in. If the expression C contains an inactive term, the constraint should be directly evaluated to true, as for compatibility constraints. On the other hand, due to the use of (implicit) quantifiers in the expression P, inactive term are directly taken into account in the semantics of P.

5.3.3 Mapping Constraints

Mapping constraints are defined in realisation views to specify under which conditions a realisation type should be included in the configuration results. They specify a mapping between structure types and item and operation types, and the latter should only be part of the final configuration if certain conditions are met. Each item and operation type R thus declares a boolean mapping condition that use terms referencing properties from the structure type T they are mapped to. Each individual r of type R is implicitly associated to a boolean term *present(r)* representing whether or not r will be added to the final configuration. During configuration, the mapping conditions $c_{map}(t, r)$ for each

individual t of type T are evaluated and must be equivalent to the value of *present(r)*.

Example 13 *Consider the physical realisation view in Figure 5.8 (on page 73) and the following mapping condition, declared in the* Coating *operation type :*

$$c_{map}(TouchScreen, coating) : oleophobicCoating = true$$

which implicitly refers to the evaluated constraint

$$oleophobicCoating = true \Leftrightarrow present(coating)$$

This condition maps an instance coating (of the Coating *operation type) to the* TouchScreen *structure type and specifies that the coating operation should be done for each touch screen that needs an oleophobic coating.*

A valid configuration thus ensures that the latter constraint is true for each instance of the TouchScreen *type, i.e., an instance of the* Coating *operation type is present if and only if the attribute* oleophobicCoating *of* touchScreen *is* true.

Semantics. Just like compatibility constraints, mapping constraints have to deal with inactive terms. This can happen when a mapping condition contains terms accessing subparts with dynamic cardinalities, in which case it should be ensured that an inactive term cannot provoke the inclusion of an item or operation type in the product realisation. Hence, a mapping condition containing at least one inactive term is evaluated to *false*: in order to be satisfied, the evaluated constraint (i.e., the implicit equivalence constraint) would then ensure that the realisation type is not present in the final configuration.

5.4 Feature Views Hierarchy

As explained in the previous sections, different feature views can be defined, in order to address the management and evolution of the product family (Research Question 4) and the scenarios discussed in Section 4.2. The feature views hierarchy starts from a base feature view, defining all the features implemented

by the product family components. This base feature view may then be specialised, as different versions or evolutions of the product family may require special restrictions to the set of available features (*Market differentiation* and *Feature set evolution* scenarios), or even more abstract feature views in order to be presented to final customers (*Distributors tailoring* and *Market analysis* scenarios).

The feature views hierarchy defines a specialisation tree, rooted by the base feature view. A feature view $\mathcal{F}'$ is the child of another feature view $\mathcal{F}$ if $\mathcal{F}'$ is a *specialisation* of $\mathcal{F}$. The specialisation of a feature view is done through different concepts:

- **Refinement and reference**: Feature views can refine feature types from their parent view. A refined feature type can transform the original type by:
  - Defining new attributes or subfeatures: New attributes can be declared in the refined type, as well as new subfeature relationships (with a new feature type).
  - Refining referenced attribute definitions: Attribute definitions can be refined. In that case, the refined feature type is *referencing* the attribute definition, and can modify it by restricting its cardinality, its domain, or its visibility (with the following hierarchy: "visible" > "read-only" > "hidden").
  - Refining referenced subfeature definitions: As for attributes, existing subfeature definitions can be referenced and restricted. A subset of the cardinality can be used, or the target feature type can be changed to one of its subtypes.
  - Changing the type from concrete to abstract: In case the feature type has subtypes, it can be defined as abstract to force the use of its subtypes.
  - Adding compatibility constraints: Asides from the transformations above, new compatibility constraints can be added to the refined feature type, involving any attributes from the current feature view (if the type is new or refined) or from the parent view (if the type is not refined).

  Figure 5.9 shows an example of feature types refinement. Type F_1 is refined: the attribute a_1 in F_1 is declared as hidden, and a new attribute a_5 is declared. The $feat_3$ subfeature definition is also refined, as the cardinality is restricted to [1..2]. Note that the feature type F_3 is not part of the refined feature view, and is just shown here for illustrative purpose.

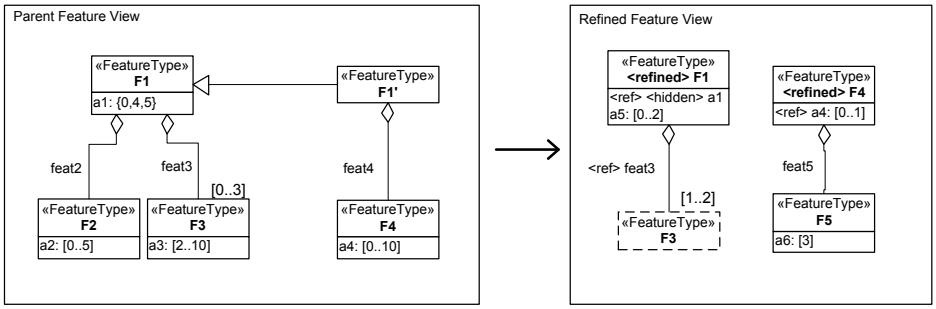


Figure 5.9: Feature view refinement. Refined types are characterised by the <**refined**> tag, while referenced definitions are tagged with <**ref**>. The type $F3$ in the Refined Feature View is shown with a dashed outline, as only the $feat3$ definition is part of the view, while the feature type is not.

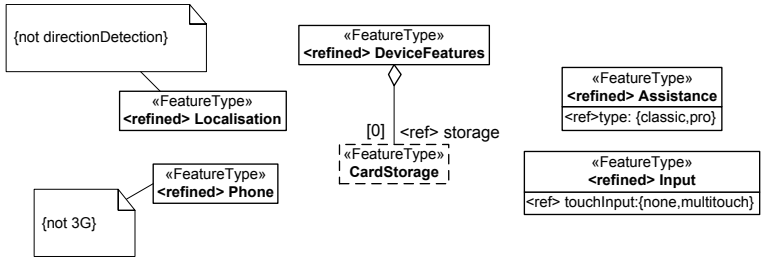


Figure 5.10: Refined feature view for the case study. This feature view uses feature types from the base feature view: it refines their attributes and subfeature definitions, and add new constraints.

Also, even though F_1' is not directly modified, the type F_4 is also refined: the domain of a_4 is reduced and a new subfeature is defined.

Example 14 *The base feature view of the running example can be specialised in different ways. Figure 5.10 shows a feature view that refines feature types from the base feature view. This refined feature view fulfils the Feature set evolution scenario: several choices in the variability of the product family are not available, due to a lack of financial resources for example. It is preferable to keep the base feature view intact, in case it is used for other scenarios or if the restrictions may disappear in the future.*

In this figure, constraints are added to the Localisation *and* Phone *types, while the* Assistance *and* Input *types are refined by restricting the domain of one of their attributes. Finally, the subfeature definition* storage *of* DeviceFeatures *is refined as well, its cardinality set to 0.*

- **Implementation**: Aside from refining types from the parent view, a specialised feature view $\mathcal{F}'$ can declare new feature types and attributes, for example to define more abstract feature groups and properties. As for the base feature view, the types in $\mathcal{F}'$ must use implementation constraints to associate their properties to the feature view $\mathcal{F}$, parent of $\mathcal{F}'$.

  Example 15 *Figure 5.11 presents another feature view, specialised from the base feature view. This view declares new feature types, and divides the functionalities of the product family in packages. The new types and attributes are connected to the base feature view properties using implementation constraints, e.g., in the* HomePackage *feature type:*

  $$multitouch \Rightarrow \\ (ParentView :: Input.touchInput = "multitouch" \\ \wedge \neg ParentView :: Input.physicalKeyboard)$$

  This constraint specifies that if multitouch *is selected, there should be at least one instance of the* Input *feature type in the parent view (i.e., the base feature view) that has its* touchInput *attribute set to "multitouch" and its* physicalKeyboard *attribute set to "false".*

Semantics. A specialised feature view is representing the features of the product family as well, but in a more constrained way. During the configuration process, one of the feature views in the hierarchy is chosen as the feature view that will be focused on. This feature view is used for configuring the product family, aside from the structure and realisation views. In a valid configuration, instances of refined types should contain the unreferenced properties of the parent view's corresponding types as well as the referenced properties defined in the refined view. Constraints of the chosen feature view should be satisfied, including implementation constraints to the parent view. The root instance of the feature view should be an instance of the root feature type of the chosen refined view, or the one from its parent view if no one is declared.

5.5 Discussion

The different views provide a modelling framework as a contribution to address the Research Questions (RQs) exposed in Section 4.2. Figure 5.12 shows the Mobile Device product family example split into hardware, software, and services components. The clear separation of concerns in the structural and realisation data for each dimension is motivated by RQ1 (*What needs of the model's user*

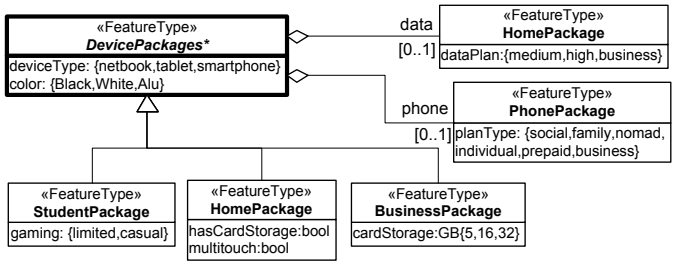


Figure 5.11: Another example of refined feature view for the Mobile Device product family. This minimal feature view uses new feature types to propose the product family to the end-user as a choice of "packages". Implementation constraints ensure the link with the base view.

should be supported?) and previous work on modelling each dimension (Section 2.1 and 4.1).

Each view is targeted at a different audience: the structural model of the software is handled by a software architect, while a production engineer may be more adequate to handle bill-of-materials and manufacturing operations. Moreover, we argue that structural and realisation views from each dimension should be considered independently from each others, and unified in the feature models they contribute to implement, defined in feature views. In Figure 5.12, the sales persons working on the device features models the types of input that the end-user may be interested in. How this feature is implemented is dependent on several structural elements from different parts of the system: the touch screen hardware and a touch framework component in the user experience software. Those two elements can however be chosen independently from each others, but will only provide the feature if they are both present in the final product.

The UML meta models (Figures 5.3, 5.5 and 5.7) provide a good basis in order to address the problem of modelling the different dimensions of an heterogeneous product family, as raised in RQ2 (*What modelling constructs support addressing the heterogeneity?*). Uniform modelling constructs and the different types of inter-view constraints defined in the framework also contribute to the issue posed in RQ3 (*How to integrate the different dimensions of heterogeneity in the models?*): the implementation and mapping constraints permit to model the interdependencies between the views, allowing a tight integration of the different dimensions of the product family.

These inter-view constraints are the key to the modelling framework, as they permit to model the product family as a whole, instead of configuring each dimension independently. The implementation constraints relate the views and

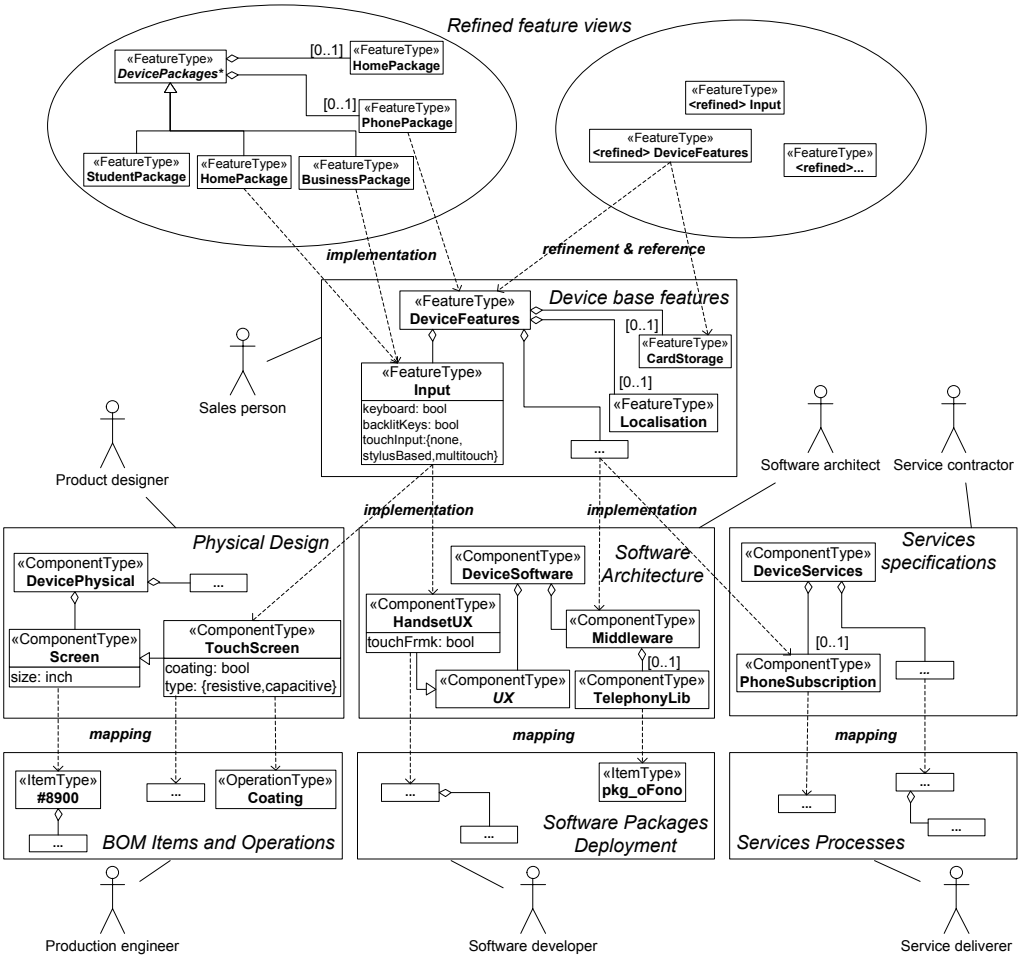


Figure 5.12: Overall view of the interdependencies in the running example between the different modelling views, depending on the three dimensions (physical, software, and services). Refined feature views are circled, while other views are surrounded by a rectangle. For the sake of readability, only two refined feature views in the hierarchy are shown.

propagate choices from one view to another during configuration, making the model globally consistent and reducing the number of errors and the time spent in configuring the different parts of the product family. Modelling implementation constraints requires communication between the different stakeholders. The sales person responsible for the touch input feature may inquire the product designer in order to assess what hardware components are needed for the requested feature. On the other hand, product designers and production engineers need to confer on which items are available to realise the structural design of the hardware.

Our modelling approach also extends the concept of feature model to a feature views hierarchy, as a contribution to RQ4 (*How to support the management of product families over time and for different market situations?*). Refinement of the base feature views permits to create specific views tailored to even more particular needs. The different scenarios defined in Section 4.2 can be modelled using refined feature views:

- The *Market differentiation* scenario results in creating a specialised feature view where domains of feature types' attributes are narrowed in order to match the restriction to the attached market.
- Refining feature types and adding temporary constraints in the feature model can be used to limit the current capabilities of the device family, e.g., to what currently can be produced. A more appropriate refined feature view may be used instead in order to make up for the *Feature set evolution*.
- *Distributors tailoring* can be achieved by allowing distributors to create their own personalised feature view, as in Figure 5.11. The newly created feature types are then linked to the lower-level feature views using implementation constraints.
- In a *Market analysis* scenario, several feature views can be created in order to match the product feature sets to introduce in the market. These views may then be joined into one base feature view, by gathering common elements or creating more abstract features that can be specialised to fit the original views, via refinement or implementation.

The feature views hierarchy enables a unification of the product family management and evolution at the feature level, independently from the heterogeneity of the family, while each dimension may have its own separate mechanism for coping with this issue (e.g., product data management). We also argue that a child feature view in the feature views hierarchy is indeed a specialisation of its parent view. The refinement and reference mechanisms restrict the model by

adding constraints explicitly or implicitly (e.g., removing values in an attribute's domain is equivalent to constraining that attribute so that it cannot take those values anymore). However, the case of creating new feature types and attributes could lead to larger feature sets and make the model less strict. Yet, each new feature type and attribute has to be related to feature types and attributes from the parent view using implementation constraints, and thus these new feature types and attributes are just another way to represent the parent's features, with the possibility to add even more constraints.

Using modelling frameworks such as the one presented in this chapter is often a cost-benefit problem. Having multiple views can make the model larger and thus introduce extra work for the knowledge engineers responsible for it. A single integrated model has indeed the advantage that the different data about a part of the system may be centralised in one particular type, while in our case they may be split into separate views depending on the dimensions considered. However, we argue that a compartmentalisation of the modelling framework is primordial when one is dealing with model involving heterogeneous product family like the Mobile Device case study. The different stakeholders can participate in the model creation and maintenance according to their area of expertise, while dependencies between the different views are handled by exchange and communication. Using refined views to model specific cases like the scenarios defined at an earlier stage also permits to adapt the feature set of a product family according to the requirements of the different stakeholders. Finally, the separation of the model into different views permits a more modular organisation of the configuration model: the use of common concepts independently from each dimension makes knowledge sharing easier, and more dimensions may be added to the product model in the future if configuration models from other design disciplines can semantically use the same basis of modelling concepts.

5.6 Comparison with related work

The conceptual basis of our modelling framework for heterogeneous product families draws from a number of sources, mainly from research in product configuration, software variability, and service configuration. One of the major source of inspiration is the configuration ontology by [93]. Concepts such as component types, resources, partonomy, and taxonomy derive from this well-known ontology. Moreover, our association type concept is semantically close to connection ports. Our framework is also described in three levels: a *meta level* where the concepts are defined, the *model level* where the configuration views and types are described using the concepts from the meta model, and an *instance level*, where configuration types are instantiated for configuration.

Aside from [93], this approach has also been considered by Cechticky et al. [23] or Asikainen et al. [6].

Our concept of features is closely related to the one from feature modelling approaches. However, many of these approaches differ from our work, as most of the techniques we use come from product configuration technology, and few have adapted them to software product lines [6, 33]. Classic feature modelling approaches, such as [53], do not use a type-instance approach, and define a configuration as a subset of the features appearing in the feature model. On the other hand, a configuration in our framework consists of a set of individuals instantiated from the types of our configuration model. This has some important consequences, in particular on the subfeature relationships: if a subfeature is shared by several features, defining a configuration without a type-instance approach will result into duplicating the subfeature instead of having two separate instances. The concept of feature cardinality defined in [23] raises the same issue, as it is the same feature that occurs several times as a subfeature. Czarnecki et al. [29] use context-free grammar to formalise feature models and provide string representations in order to allow the repetitions of features.

Another difference between our approach on feature views and feature modelling in software product lines comes from the complexity of the mechanisms used. Classic feature models, as well as extensions like cardinality-based models [29] do usually not consider taxonomy structure, or limit the variability in the model to the different relations between features, while we provide the possibility to use concrete compatibility constraints.

Multi-view models in feature modelling have also been studied. Czarnecki et al. [26] sketch a model where different levels of customisation are modelled (including feature and design view). Reiser and Weber [82] and work from Zaid et al. [111] propose feature models with different perspectives, although they are all centered on software variability and feature modelling techniques only, and the lack of specialisation hierarchy may make the task of implementing the unification with different structured views difficult. Männistö et al. [62] define a specialisation tree for product structures called an *element model hierarchy*. In this work, several configuration models (similar to our structure views) can be created for a single product family, by using *specialisation operations*. One of the obvious differences with our current work is that we use refinement on feature views in order to solve specific scenarios (see Section 4.2), while Männistö et al. concentrate on organising multiple structural models into a hierarchy for better management. Another difference lies in the specialisation operations: asides from type and attribute refinement (including adding new compatibility constraints), we permit the creation of new feature types and attributes, linked to the parent feature view using *implementation constraints*, which allows to handle a broader range of scenarios (in particular the *Distributors tailoring* scenario).

The four worlds from Heiskala et al. [49] can be compared with the modelling views of our framework when related to service configuration: the *needs world* concerns the customer's needs (in an abstract way), and is thus close to our feature views, which describes the abstract features that the customer may require; the *service solution world* denotes the set of elements used to establish the service's specifications, as the structure views; the *process world* describes how the service will be delivered, or realised, as in our realisation views. Note that there is nothing in our conceptual approach that is similar to the *object-of-services world* from [49], which specifies the service recipients or the environment relevant to those recipient. From a modelling point of view, all these worlds are based on the same meta model, using different types and attributes, as well as taxonomy and partonomy structures, as in our approach. However, dependencies between types of different worlds are simply modelled using classical constraints, while we use implementation and mapping constraints. Moreover, one important conceptual difference between our work and the four-worlds approach is that our framework is centered on the configured product, and thus the services described in the services dimension are seen from the configured product's point of view, while the external environment is not considered. One could then say that the only "object-of-services" in our approach is the configured product itself. One potential solution to take the environment into account could be the definition of another type of view, that could be associated to a refined feature view, and where the different sales channels would define externally controlled elements (such as, in the running example, access to company specific services or credentials, data transfer from an old device, etc.).

Kumbang [6] is the closest to our work on the software variability side, including their type-instance approach. We consider our work to be an extension of Kumbang, as we use implementation constraints to unify structure views from the different dimensions (including manufactured products and services), and we also model realisation data. Thus the main contribution of our work is to provide conceptual and practical mechanisms to bring the different dimensions together and unify them under feature models. The two views in Kumbang are also solved separately from each other: both points of view are subject to their own consistency rules, and implementation constraints may not modify the consistency of a configuration. On the other hand, implementation constraints in our framework integrate the different views, so that the model must be consistent as a whole.

Chapter 6

Framework Implementation

The concepts developed in the previous chapter are the basis for modelling and solving the configuration of heterogeneous product families. In this chapter, we define a textual modelling language called ProCoLa that supports our conceptual approach. This language aims at providing a practical and well-defined way to represent configuration models, as well as permitting a direct interpretation of the models in order to assist the end-user with the actual configuration of the product families.

We then propose a formal definition of our conceptual framework for modelling variability in heterogeneous product families. This formalism is used to provide strict semantics to the framework and the ProCoLa modelling language, by means of a type system and rules of well-formedness. Finally, we define graph-based static analyses of the configuration models using ProCoLa and the formalism defined, in order to assist the modeller in his task of maintaining product models.

6.1 The ProCoLa Modelling Language

The *ProCoLa* modelling language is a textual object oriented language created to fit the concepts developed in Chapter 5. Defining such a language permits to

FeatureView	:=	`featureView` Id_f [`specialisationOf` Id_s] [`units:`(*UnitDecl*)*] [`constants:`(*ConstDecl*)*] `featuresDeclaration:` (*FTypeDecl*)*
UnitDecl	:=	Id_u `:` *TypeDecl*`;`
ConstDecl	:=	Id_c [`:`Id_u]`:= c;`
FTypeDecl	:=	[`[Root]`] [`refined`] [`abstract`] `featureType` Id_t [`subtypeOf` Id_1 (`,` Id_i)*] [{ [`description:`[`c;`]] [`attributes:`(*AttrDecl*)*] [`subFeatures:`(*SubFDecl*)*] [`constraints:`(*ConstrDecl*)*] [`implementation:`(*ImplDecl*)*] }]`;`
AttrDecl	:=	[`ref`] *Visibility* Id_a [`[`*C*`]`]`:` *TypeDecl*`;`
Visibility	:=	`readonly` | `hidden` | ϵ
SubFDecl	:=	[`ref`] Id_s [`[`C_1 [`..` C_2]`]`]`:` Id_t`;`
ConstrDecl	:=	{ *CSymExp* }`;` | [Id_c`:`]{ [`description:`[`c;`]] *ConstrVal*}`;`
ConstrVal	:=	[`type:hard;`] `value:`{ *CSymExp* }`;`
	|	`type:table[`*CSymExp* (`,`*CSymExp*)*`]; value:`{*TableExpr*}`;`
ImplDecl	:=	{*ImplExp*}`;` | [Id_i`:`]{ [`description:`[`c;`]] `value:`*ImplExp*}`;`
TypeDecl	:=	*Type* [*DomReduc*]
DomReduc	:=	{ C_1 (`,` C_2)* } | `[`C_1`..`C_2`]` | `[- inf..`C_2`]` | `[`C_1`..inf]`
Type	:=	Id_u | `integer` | `boolean` | `enum`
C	:=	`c` | Id_c

Table 6.1: ProCoLa feature view syntax

support the conceptual approach in several ways:

- It defines configuration models in a format that can be shared easily without the need for any special tool except a text editor.
- It has a precise and unambiguous syntax, compared to other modelling languages such as UML, where profiles need to be defined and attached to classes. Moreover, the profile mechanism has itself been criticised for ambiguity, e.g., with respect to whether only classes or all metaclasses may be extended, and problems in its semantics [50].
- It provides a machine-readable code for interpretation and analysis of the models.
- It has the potential for an easy implementation of tool support and custom User Interfaces.

StructureView	:=	`dimension` *Dimension*`;` *StructureDecl*
Dimension	:=	`Physical` | `Software` | `Services`
StructureDecl	:=	[`units:`(*UnitDecl*)*] [`constants:`(*ConstDecl*)*] [`resources:`(*ResourceDecl*)*] `structureDeclaration:`(*STypeDecl*)*
UnitDecl	:=	Id_u`:` *TypeDecl*`;`
ConstDecl	:=	Id_c [`:`Id_u]`:=` *C*`;`
ResourceDecl	:=	Id_r`:`Id_u`;` | Id_r`:integer;`
STypeDecl	:=	*CTypeDecl* | *ATypeDecl*
CTypeDecl	:=	[`[Root]`] [`abstract`] `componentType` Id_t [`subtypeOf` Id_1 (`,` Id_i)*] [`{` [`description:`[`c;`]] [`attributes:`(*AttrDecl*)*] [`subParts:`(*SubPDecl*)*] [`associations:`(*AssocDecl*)*] [`produces:`(*ResourceUse*)*] [`consumes:`(*ResourceUse*)*] [`constraints:`(*ConstrDecl*)*] `}`]`;`
ATypeDecl	:=	[`abstract`] `associationType` Id_a [`subtypeOf` Id_1 (`,` Id_i)*] [`{` [`description:`[`c;`]] [`attributes:`(*AttrDecl*)*] [`constraints:`(*ConstrDecl*)*] `}`]`;`
AttrDecl	:=	*Visibility* Id_a [`[`*C*`]`]`:` *TypeDecl*`;`
Visibility	:=	`readonly` | `hidden` | ϵ
SubPDecl	:=	Id_s [`[`C_1 [`..` C_2]`]`]`:` Id_t`;`
AssocDecl	:=	*Direction* Id_a [`[`C_1 [`..` C_2]`]`]`:` Id_t`;`
Direction	:=	`provides` | `requires` | ϵ
ResourceUse	:=	Id_r`:=` *CSymExp*
ConstrDecl	:=	`{` *CSymExp* `};` | [Id_c`:`]`{` [`description:`[`c;`]] *ConstrVal*`};`
ConstrVal	:=	[`type:hard;`] `value:{` *CSymExp* `};`
	|	`type:table[`*CSymExp* (`,`*CSymExp*)*`]; value:{`*TableExp*`};`
TypeDecl	:=	*Type* [*DomReduc*]
DomReduc	:=	`{` C_1 (`,` C_2)* `}` | `[`C_1`..`C_2`]` | `[- inf..`C_2`]` | `[`C_1`..inf]`
Type	:=	Id_u | `integer` | `boolean` | `enum`
C	:=	`c` | Id_c

Table 6.2: ProCoLa structure view syntax

The ProCoLa language defines configuration models in three different types of files, representing the three different types of views introduced in Chapter 5. Tables 6.1, 6.2, and 6.3 present the syntax of ProCoLa for the feature, structure, and realisation views, respectively. The syntax is shown as a context-free grammar using regular expressions, in order to avoid the cumbersome derivations for lists. Expressions are presented separately in Table 6.4.

RealisationView	:=	`dimension` *Dimension*`;` `itemsDeclaration:`(*ITypeDecl*)* `operationsDeclaration:`(*ORTypeDecl*)*
Dimension	:=	`Physical` | `Software` | `Services`
ITypeDecl	:=	`itemType` *Id*$_i$ [{ [`mapping:`(*MappingDecl*)*] [`subItems:`(*SubEltDecl*)*] }]`;`
ORTypeDecl	:=	*OTypeDecl* | *RTypeDecl*
OTypeDecl	:=	`operationType` *Id*$_o$ [{ [`mapping:`(*MappinDecl*)*] [`successors:`(*Id*`;`)*] [`uses:`(*UseDecl*)*] [`subOperations:`(*SubEltDecl*)*] }]`;`
RTypeDecl	:=	`resourceType` *Id*$_r$`;`
MappingDecl	:=	*Id*$_t$`:` { *CSymExp* }`;`
SubEltDecl	:=	*Id*$_s$ [`[c]`]`:` *Id*$_t$`;`
UseDecl	:=	*Id*$_u$`:` *Id*$_r$`;`

Table 6.3: ProCoLa realisation view syntax

TableExp	:=	`[` [*C*$_1$ (`,`*C*$_2$)*] (`,` [*C*$_i$ (`,`*C*$_j$)*])* `]`
	|	`MSSQL(` *C*$_{conn}$ `)` `::` *C*$_{table}$`;`
	|	`Excel` `::` *C*$_{file}$`;`
ImplExp	:=	{ *CSymExp*$_L$ } `<->` { *ISymExp*$_R$ }
	|	{ *CSymExp*$_L$ } `->` { *ISymExp*$_R$ }
	|	{ *CSymExp*$_L$ `==` *ISymExp*$_R$ }
CSymExp	:=	*Var* | *C* | *CSymExp Op CSymExp* | `not` *CSymExp*
	|	`(`*CSymExp*`)` | *Var* `is` *Id*$_t$ | *Func*`(`*Var*$_1$ (`,`*Var*$_2$)*`)`
ISymExp	:=	*IVar* | *C* | *ISymExp Op ISymExp*
	|	`not` *ISymExp* | `(`*ISymExp*`)` | *IVar* `is` *Id*$_t$
	|	*Func*`(`*IVar*$_1$ (`,`*IVar*$_2$)*`)` | `Present(`*Id*$_t$`)`
Op	:=	`->` | `<->` | `and` | `or` | `xor` | `=` | `!=` | `<` | `<=` | `>` | `>=`
	|	`+` | `-` | `*` | `mod`
Func	:=	`Count` | `Sum` | `Max` | `Min`
IVar	:=	*Id*$_t$`::` *Var*
Var	:=	*Id*$_v$ | *Id*$_v$`.` *Var*

Table 6.4: ProCoLa expressions syntax

The keywords used in ProCoLa are closely related to the concepts defined in our modelling framework: the syntax for feature views is based on the declaration of feature types, while the one for structure views permits to declare component types and association types, and the one for realisation views permits to declare item, operation and resource types. Several sections can be declared in each view. Feature and structure views may declare units and constants, and a resource declaration section may also be added for structure views. Both feature and structure views have a mandatory section where types are declared (with a `featuresDeclaration` header for feature types and `structureDeclaration` for structure types). Realisation views have two mandatory sections: one with an `itemsDeclaration` header for declaring item types and one with the header `operationsDeclaration` for operation and resource types.

Units can be used in feature and structure views in place of basic types in order to provide more concrete information on the attributes defined. Units are declared using an identifier, a basic type and optionally a domain reduction.

Example 16 *A unit* inch *can be defined in a* `units` *section by*

```
units:
  inch: integer[0..inf];
```

This specifies that the domain of each attribute with an inch unit will only contain positive integers.

Constants represent identifiers that are assigned constant values, and can be used in many places where an actual constant value can be used. The derivation C can be a constant identifier Id_c or an explicit constant value `c`, which represents a truth value (`true` or `false`), an integer, or a string.

Several types of expressions can be used in the different views. Compatibility constraints (defined in the `constraints` sections of feature, component, and association types) can use symbolic expressions (*CSymExp*) or table expressions for declaring table constraints. Implementation constraints in feature views (*ImplExp*) also use symbolic expressions, although the right hand side symbolic expressions (*ISymExp*) are slightly different than the ones for compatibility constraints in that they use variables with explicit context (*IVar*) and can contain function calls with the *Present(...)* function. Mapping conditions in realisation views also use compatibility symbolic expressions (*CSymExp*). Table expressions can be explicit, where all allowed tuples are defined in the ProCoLa file, or can refer to a Microsoft SQL Server database or an Excel spreadsheet. More information on the last two options is provided in Chapter 10.

Example 17 *The following table constraint is declared in the ProCoLa model of our case study's physical structure view to define the relations between the size of the device's enclosure, the size of its screen and the number of available chip slots on the motherboard:*

```
TableSize: {
type:table[enclosure.size,screen.size,motherboard.nbSlots];
value: {
   ["115x58" , 3 , 6 ],
   ["240x190", 9 , 10],
   ["268x178", 10, 12],
   ["295x210", 12, 14]
 };
};
```

Example 18 *As a more concrete example, here is the ProCoLa syntax for the DeviceFeature feature type from our case study (Figure 5.4 on page 65).*

```
featuresDeclaration:
 [Root] featureType DeviceFeatures [
  attributes:
   deviceType: enum{"netbook", "tablet", "smartphone"};
   color: enum{"Black", "White", "Alu"};

  subFeatures:
   localisation: Localisation;
   communication: Communication;
   assistance: Assistance;
   input: Input;
   cardStorage[0..1]: CardStorage;

  constraints:
   { (deviceType = "smartphone" and input.physicalKeyboard)
       -> input.backlitKeyboard };

  implementation:
   { (not deviceType = "netbook") and input.touchInput = "
       multitouch"} <-> { Physical::TouchScreen.screenType =
       "capacitive" };
   { color == Physical::Enclosure.color };];
```

The feature type declaration defines the type as a root feature type, and specifies its attributes, subfeatures, and constraints. For sake of brevity, only a handful of constraints are presented here, in their inline form (i.e., without constraint identifier or constraint type declaration). The compatibility constraint specifies that if the device is a smartphone and it has a physical keyboard, then the backlit

keyboard must be selected. The two implementation constraints link the multi-touch capability and the color of the device to its physical structure. The full ProCoLa model of the extended case study is available in [74].

Defining the language syntax is only the first step in the implementation of the modelling framework. The next section defines a formalism for the framework that will be used to check and analyse ProCoLa models, as well as provide a basis for defining the concrete semantics of the language.

6.2 Formalism Definition

We provide in this section some definitions aimed at establishing a formalism for our conceptual framework for heterogeneous product families. The definitions are provided in a top-down approach, starting with the ones on the level of configuration models, and entering into details onwards. Examples related to our case study are given when relevant. A type system and well-formedness rules for this formalism are defined in Section 6.3.

6.2.1 Configuration Model

As pointed out previously, the broadest concept in our framework is the one of a configuration model:

Definition 1 (Configuration model) *A* configuration model *is a tuple* $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_{\mathrm{M}} \rangle$*, where* $\mathcal{V}^f$ *is a set of feature views,* $\mathcal{V}^s$ *is a set of structure views, and* $\mathcal{V}^r$ *is a set of realisation views of the model. The function* Sp_{M} *is the* specialisation function *between feature views.*

A configuration model thus contains the different views of the models, and relates the feature views using the specialisation function, defined by:

Definition 2 (Specialisation function, base view) *The* specialisation function $Sp_{\mathrm{M}} \subseteq \mathcal{V}^f \times \mathcal{V}^f$ *is a bijective relation such that* $(\mathrm{F}, \mathrm{F}') \in Sp_{\mathrm{M}}$ *if* F *is a specialisation of* F' *in the feature views hierarchy.*
We then define Sp^*_{M} *as the transitive closure of* Sp_{M}*, so* $(\mathrm{F}, \mathrm{F}') \in Sp^*_{\mathrm{M}}$ *if* F *is a descendant of* F' *in the feature views hierarchy.*
We finally define a base view $base_{\mathrm{M}} \in \mathcal{V}^f$ *as being a view which is not a specialisation of any other view, i.e.,* $\mathrm{F} \in base_{\mathrm{M}} \Leftrightarrow \nexists \mathrm{F}'$ *s.t.* $(\mathrm{F}, \mathrm{F}') \in Sp_{\mathrm{M}}$.

Example 19 *Consider our case study of the Mobile Device product family described in Chapter 5. The configuration model can be defined using the following sets of views:*

$$\begin{aligned} \mathcal{V}^f &= \{\mathrm{F}_{base}, \mathrm{F}_{packages}, \ldots\} \\ \mathcal{V}^s &= \{\mathrm{S}_{phys}, \mathrm{S}_{soft}, \mathrm{S}_{serv}\} \\ \mathcal{V}^r &= \{\mathrm{R}_{phys}, \mathrm{R}_{soft}, \mathrm{R}_{serv}\} \end{aligned}$$

F_{base} *represents the base feature views, while* $\mathrm{F}_{packages}$ *is one of the refined feature views. They are thus related by the refinement function:*

$$Sp_{\mathrm{M}}(\mathrm{F}_{packages}) = \mathrm{F}_{base}$$

$\mathcal{V}^s$ *contains the physical, software, and service structure views, while* $\mathcal{V}^r$ *contains the physical, software, and service realisation views.*

The next sections go into more detail by defining the feature views, structure views, and realisation views. Finally, the constraint language is defined in Section 6.2.5.

6.2.2 Feature Views

Feature views declare feature types and the relations between them, as well as dependencies to other feature and structure views:

Definition 3 (Feature view) *A* feature view $\mathrm{F} \in \mathcal{V}^f$ *is defined by a tuple* $\mathrm{F} = \langle \mathcal{T}_\mathrm{F}, \mathrm{T}_\mathrm{F}^{root}, \mathcal{S}_\mathrm{F}, I_\mathrm{F}, \mathcal{D}_\mathrm{F}, \prec_\mathrm{F}, \mathcal{C}_\mathrm{F}^c, \mathcal{C}_\mathrm{F}^i, \mathrm{M} \rangle$*, where*

- $\mathcal{T}_\mathrm{F}$ *is a set of feature types, and a set* $\mathcal{T}_\mathrm{F}^{Ab} \subset \mathcal{T}_\mathrm{F}$ *is defined to represent the set of abstract types in* F*,*
- $\mathrm{T}_\mathrm{F}^{root} \in (\mathcal{T}_\mathrm{F} \cup \mathcal{T}_{\mathrm{F_P}})$ *is the root of the feature view, with* $\mathrm{F_P}$ *a potential parent view of* F $(\mathrm{F} = Sp_{\mathrm{M}}(\mathrm{F_P}))$*,*
- $\mathcal{S}_\mathrm{F}$ *is a set of subfeature definitions,*
- I_F *is the feature taxonomy relation between feature types,*
- $\mathcal{D}_\mathrm{F}$ *is a set of attribute definitions,*
- $\prec_\mathrm{F}$ *is the reference relation between attribute and subfeature definitions,*
- $\mathcal{C}_\mathrm{F}^c$ *is a set of compatibility constraints and*

- $\mathcal{C}^i_{\mathrm{F}}$ *is a set of implementation constraints.*
- *The model* M *is repeated for practical purposes, although it is redundant.*

Feature types are thus divided into two groups: abstract and concrete types. The root feature type can belong to another feature view, parent of F, in case F is a specialisation of another view and does not define such a root type. The root type is then "inherited" from the parent view.

Example 20 *Let's consider the set of feature types in the base feature view of our case study* F_{base}*:*

$$\mathcal{T}_{\mathrm{F}_{base}} = \{\text{DeviceFeatures}, \text{Localisation}, \text{Communication}, \text{Assistance}, \ldots\}$$

The root type of the base feature view is the DeviceFeatures type, which is formalised by

$$\mathrm{T}^{root}_{\mathrm{F}_{base}} = \text{DeviceFeatures}$$

Feature types can be refined from one feature view to another:

Definition 4 (Refined type) *A type* T *is* refined *from* $\mathrm{F_P} \in \mathcal{V}^f$ *to* $\mathrm{F} \in \mathcal{V}^f$ *if and only if* $\mathrm{T} \in \mathcal{T}_{\mathrm{F}} \cup \mathcal{T}_{\mathrm{F_P}}$ *and* $(\mathrm{F}, \mathrm{F_P}) \in Sp^*_{\mathrm{M}}$.

A type T *is said* directly refined *from* F' *to* F *if and only if:*

- T *is refined from* F' *to* F
- $\nexists \mathrm{F}''$ *s.t.* T *is refined from* F'' *to* F *and* $(\mathrm{F}'', \mathrm{F}') \in Sp^*_{\mathrm{M}}$

i.e., if there is no intermediary feature view F'' *from where* T *is refined.*

This means that the feature type T is refined if it belongs to the two views F and $\mathrm{F_P}$ and if these two views are related in the feature views hierarchy (F is a specialisation of $\mathrm{F_P}$).

Subfeature definitions relate feature types to form the partonomy tree in the view:

Definition 5 (Subfeature definition) *A* subfeature definition *in* $\mathcal{S}_{\mathrm{F}}$ *is a tuple* $\langle name, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2 \rangle$ *where name is the string name of the subfeature,* $\mathrm{T}_s \in \mathcal{T}_{\mathrm{F}}$ *is the source type and* $\mathrm{T}_t \in \mathcal{T}_{\mathrm{F}}$ *the target type. The cardinality is given by a minimum* r_1 *and a maximum* r_2*, denoted as* $[r_1, r_2]$.

The name of a subfeature definition is used to identify each relations so it can be referred to in the different constraints of the model.

Example 21 *Subfeature definitions for* F_{base} *are contained in* $\mathcal{S}_{\mathrm{F}_{base}}$*. The subfeature definition* $\sigma_{phone} \in \mathcal{S}_{\mathrm{F}_{base}}$ *specified by*

$$\sigma_{phone} \quad = \quad \langle \text{“}phone\text{”}, \mathrm{Communication}, \mathrm{Phone}, 0, 1 \rangle$$

describes the definition of the subfeature phone *of type* Phone *within the feature type* Communication*. The cardinality is* $[0, 1]$*, which means that this subfeature may not be included in the final configuration, depending on the user requirements and the constraints defined in the model.*

Feature types are also organised in a taxonomy, thanks to the feature taxonomy relation:

Definition 6 (Feature taxonomy relation) *The* feature taxonomy relation $I_{\mathrm{F}} \subseteq \mathcal{T}_{\mathrm{F}} \times \mathcal{T}_{\mathrm{F}}$ *is an irreflexive relation such that* $(\mathrm{T}_1, \mathrm{T}_2) \in I_{\mathrm{F}}$ *if* T_1 *is a direct subtype of* T_2*.*

We define $\sqsubseteq_{\mathrm{F}}$ *as the reflexive transitive closure of* I_{F}*:*
$\forall \mathrm{T}, \mathrm{T}' \in \mathcal{T}_{\mathrm{F}}$,
$\mathrm{T}' \sqsubseteq_{\mathrm{F}} \mathrm{T} \Leftrightarrow \mathrm{T}' = \mathrm{T} \vee \exists \mathrm{T}_1, \ldots, \mathrm{T}_n$ *s.t.* $(\mathrm{T}', \mathrm{T}_1), (\mathrm{T}_1, \mathrm{T}_2), \ldots, (\mathrm{T}_n, \mathrm{T}) \in I_{\mathrm{F}}$.

This means for two types T and T', we have $\mathrm{T}' \sqsubseteq_{\mathrm{F}} \mathrm{T}$ if T' is equal to T or is one of its subtypes (direct or transitively).

We go on by defining attribute definitions:

Definition 7 (Attribute definition) *An* attribute definition *in* $\mathcal{D}_{\mathrm{F}}$ *is a tuple* $\langle name, \mathrm{T}_s, \mathrm{D}, v, r \rangle$ *where* l *is the string name of the attribute,* $\mathrm{T}_s \in \mathcal{T}_{\mathrm{F}}$ *is the source type and* D *is a non-empty set of literal values called the* domain *of the attribute. The cardinality* $r \in \mathbb{N}$ *denotes the number of individuals of this attribute instantiated during configuration. Finally,* $v \in \{\textsc{Visible}, \textsc{ReadOnly}, \textsc{Hidden}\}$ *represents the visibility of the attribute definition. The set of visibility values is totally ordered:* $\textsc{Visible} > \textsc{ReadOnly} > \textsc{Hidden}$*.*

Domains represent the possible values that each instance of an attribute can take during the configuration process. As for subfeature definitions, the name of an attribute is used for display and identification purposes in constraints.

Example 22 *Attribute definitions for* F_{base} *are contained in* $\mathcal{D}_{\mathrm{F}_{base}}$*. The attribute definition* $\alpha_{color} \in \mathcal{D}_{\mathrm{F}_{base}}$ *specified by*

$$\alpha_{color} \quad = \quad \langle \text{“}color\text{”}, \mathrm{DeviceFeatures}, \mathrm{D}_{color}, \textsc{Visible}, 1 \rangle$$

describes the definition of the attribute color *within the* DeviceFeatures *feature type. The domain* D_{color} *contains three string values representing the potential values the attribute can take:*

$$\mathrm{D}_{color} \quad = \quad \{\text{“}Black\text{”}, \text{“}White\text{”}, \text{“}Alu\text{”}\}$$

This attribute has the default cardinality of 1, so only one instance will appear during configuration.

Refined feature types can reference subfeature and attribute definitions:

Definition 8 (Reference relation) *The* subfeature reference relation $\prec^{s}_{\mathrm{F}} \subseteq (\mathcal{S}_{\mathrm{F}} \times \bigcup_{(\mathrm{F},\mathrm{F}') \in Sp^{*}_{\mathrm{M}}} \mathcal{S}_{\mathrm{F}'})$ *is an irreflexive transitive relation between subfeature definitions defined in the view* F *and subfeature definitions defined in its parent views* F′*. For two subfeature definitions* $\sigma_1 = \langle name_1, \mathrm{T}_{s1}, \mathrm{T}_{t1}, r_{11}, r_{21} \rangle$ *and* $\sigma_2 = \langle name_2, \mathrm{T}_{s2}, \mathrm{T}_{t2}, r_{12}, r_{22} \rangle$*, we have* $\sigma_2 \prec^{s}_{\mathrm{F}} \sigma_1$ *if and only if:*

- $\mathrm{T}_{s1} = \mathrm{T}_{s2} = \mathrm{T}_{\mathrm{ref}}$ *and* $name_1 = name_2$
- $\exists \mathrm{F}_{\mathrm{P}}$ *s.t.* $\mathrm{T}_{\mathrm{ref}}$ *is a feature type refined from* F_{P} *to* F *and* $\sigma_2 \in \mathcal{S}_{\mathrm{F}_{\mathrm{P}}}$

In a similar way, we define the attribute reference relation

$$\prec^{a}_{\mathrm{F}} \subseteq (\mathcal{D}_{\mathrm{F}} \times \bigcup_{(\mathrm{F},\mathrm{F}') \in Sp^{*}_{\mathrm{M}}} \mathcal{D}_{\mathrm{F}'})$$

as an irreflexive transitive relation between attribute definitions defined in the view F *and attribute definitions defined in its parent views* F′*. For two attribute definitions* $\alpha_1 = \langle name_1, \mathrm{T}_1, \mathrm{D}_1, v_1, r_1 \rangle$ *and* $\alpha_2 = \langle name_2, \mathrm{T}_2, \mathrm{D}_2, v_2, r_2 \rangle$*, we have* $\alpha_2 \prec^{a}_{\mathrm{F}} \alpha_1$ *if and only if:*

- $\mathrm{T}_1 = \mathrm{T}_2 = \mathrm{T}_{\mathrm{ref}}$ *and* $name_1 = name_2$
- $\exists \mathrm{F}_{\mathrm{P}}$ *s.t.* $\mathrm{T}_{\mathrm{ref}}$ *is a feature type refined from* F_{P} *to* F *and* $\alpha_2 \in \mathcal{D}_{\mathrm{F}_{\mathrm{P}}}$

Finally, we define the reference relation $\prec_{\mathrm{F}}$ *as the union of the two previously defined functions:*

$$\prec_{\mathrm{F}} = \prec_{\mathrm{F}}^{s} \oplus \prec_{\mathrm{F}}^{a}$$

The subfeature (resp. attribute) reference relations apply to two subfeature (resp. attribute) definitions where one is the refinement of the other. Hence, it means that one belongs to a feature view which is a specialisation of the other. However, this specialisation step may not be atomic, i.e., there may be multiple refinements and references in between these two subfeature (resp. attribute) definitions. We therefore need to specify a direct reference function.

Definition 9 (Direct reference function) *The* direct reference function Ref_{F} *is defined as follows:*

$$\forall \delta_1, \delta_2, \text{ we have } \delta_2 = Ref_{\mathrm{F}}(\delta_1) \text{ if and only if } \nexists \delta_3 \text{ s.t. } \delta_2 \prec_{\mathrm{F}} \delta_3 \prec_{\mathrm{F}} \delta_1$$

i.e., there is no intermediary subfeature or attribute definition δ_3 *to which* δ_2 *is a reference.*

Example 23 *Consider the feature view refinement* F_{ref} *presented in Figure 5.10 on page 80. The* Assistance *feature type is refined from* F_{base} *to* F_{ref}*, and the two attribute definitions*

$$\begin{aligned} \alpha_{type} &= \langle \text{“}type\text{”}, \text{Assistance}, \mathrm{D}_{type}, \textsc{Visible}, 1 \rangle &\in \mathcal{D}_{\mathrm{F}_{base}} \\ \alpha_{type}^{ref} &= \langle \text{“}type\text{”}, \text{Assistance}, \mathrm{D}_{type}^{ref}, \textsc{Visible}, 1 \rangle &\in \mathcal{D}_{\mathrm{F}_{ref}} \end{aligned}$$

are related by

$$\alpha_{type}^{ref} = Ref_{\mathrm{F}}(\alpha_{type})$$

with $\mathrm{D}_{type} = \{\text{“}classic\text{”}, \text{“}extended\text{”}, \text{“}pro\text{”}\}$, $\mathrm{D}_{type}^{ref} = \{\text{“}classic\text{”}, \text{“}pro\text{”}\}$ *and* $\mathrm{D}_{type}^{ref} \subset \mathrm{D}_{type}$*. The* type *attribute is thus refined from the feature view* F_{base} *to the feature view* F_{ref} *as its domain is reduced from one view to the other.*

We can now define compatibility and implementation constraints. Each constraint is assigned a context type that may serve during evaluation of the constraint expressions used. The expressions are not discussed in details here, and will be explicitly defined in Section 6.2.5.

Definition 10 (Compatibility constraint) *A* compatibility constraint *in* $\mathcal{C}^{\mathrm{c}}_{\mathrm{F}}$ *is a pair* $\langle \mathrm{T}, e^c \rangle$ *where* $\mathrm{T} \in \mathcal{T}_{\mathrm{F}}$ *is the context type and* e^c *is a compatibility expression.*

Example 24 *Consider the compatibility constraint* $c_{internet} \in \mathcal{C}^{c}_{\mathrm{F}_{base}}$*:*

$$c_{internet} \quad = \quad \langle \text{Communication}, e^c_{internet} \rangle$$

The constraint is defined in the context of the feature type Communication*. The compatibility expression* $e^c_{internet}$ *describes the following:*

"The device has internet access if and only if it has a Wifi connection, Ethernet access, or a 3G connection."

This expression can be written using the constraint language described in Section 6.2.5.

Definition 11 (Implementation constraint) *An* implementation constraint *in* $\mathcal{C}^{i}_{\mathrm{F}}$ *is a tuple* $\langle \mathrm{T}, e^c_{\mathrm{C}}, e^i_{\mathrm{P}}, \mathrm{Op} \rangle$ *where* $\mathrm{T} \in \mathcal{T}_{\mathrm{F}}$ *is the context type,* e^c_{C} *is a compatibility expression (called the* child *expression),* e^i_{P} *is an implementation expression (called the* parent *expression), and* $\mathrm{Op} \in \{\Rightarrow, \Leftrightarrow, =\}$ *the binding operator.*

Example 25 *Consider the implementation constraint* $c_{3G} \in \mathcal{C}^{i}_{\mathrm{F}_{base}}$*:*

$$c_{3G} \quad = \quad \langle \text{Phone}, e^c_{3G}, e^i_{3G}, \Leftrightarrow \rangle$$

The constraint is defined in the context of the feature type Phone*. The compatibility expression* e^c_{3G} *tests whether the 3G option is selected, while the implementation expression* e^i_{3G} *makes sure that if so, the required physical and software parts are present in the configuration:*

"The phone can use a 3G connection if and only if there exists a RF card supporting GSM/UMTS and the 3G plugin is installed in the software."

How to define these two expressions using the constraint language is described in Section 6.2.5.

This concludes the formal definition for the feature views of our framework. We will now define the elements composing the structure and realisation views.

6.2.3 Structure Views

We present in this section the formalism for structure views.

Definition 12 (Structure view) *A* structure view S *in* $\mathcal{V}^s$ *is defined by the tuple* $\mathrm{S} = \langle \mathcal{T}_\mathrm{S}, \mathrm{T}_\mathrm{S}^{root}, \mathcal{R}_\mathrm{S}, \mathcal{U}_\mathrm{S}, \mathcal{S}_\mathrm{S}, \mathcal{A}_\mathrm{S}, I_\mathrm{S}, \mathcal{D}_\mathrm{S}, \mathcal{C}_\mathrm{S}^c, \mathrm{M} \rangle$ *in* $\mathcal{V}^s$ *where*

- $\mathcal{T}_\mathrm{S} = \mathcal{T}_\mathrm{S}^c \cup \mathcal{T}_\mathrm{S}^a$ *is a set of structure types and contains both component types* $\mathcal{T}_\mathrm{S}^c$ *and association types* $\mathcal{T}_\mathrm{S}^a$*; moreover, we also define the set* $\mathcal{T}_\mathrm{S}^{Ab} \subset \mathcal{T}_\mathrm{S}$ *of abstract types in* S*,*
- $\mathrm{T}_\mathrm{S}^{root} \in \mathcal{T}_\mathrm{S}^c$ *is the root of the structure view,*
- $\mathcal{R}_\mathrm{S}$ *is a set of resources,*
- $\mathcal{U}_\mathrm{S}$ *is a set of resource use definitions,*
- $\mathcal{S}_\mathrm{S}$ *is a set of subcomponent definitions,*
- $\mathcal{A}_\mathrm{S}$ *is a set of association definitions,*
- $I_\mathrm{S} \subseteq \mathcal{T}_\mathrm{S} \times \mathcal{T}_\mathrm{S}$ *is the structure taxonomy relation between structure types,*
- $\mathcal{D}_\mathrm{S}$ *is a set of attribute definitions and*
- $\mathcal{C}_\mathrm{S}^c$ *is a set of compatibility constraints.*
- *As before, the model* M *is repeated for practical purposes.*

Example 26 *Consider now the structure types in* S_{phys}*.*

$$\begin{aligned} \mathcal{T}_{\mathrm{S}_{phys}}^c &= \{\text{DevicePhysical}, \text{Motherboard}, \text{Screen}, \ldots\} \\ \mathcal{T}_{\mathrm{S}_{phys}}^a &= \{\text{Stylus}\} \end{aligned}$$

The DevicePhysical*,* Motherboard*, and* Screen *types are component types, and thus belongs to the* $\mathcal{T}_{\mathrm{S}_{phys}}^c$ *set, while* Stylus *is an association type and is in* $\mathcal{T}_{\mathrm{S}_{phys}}^a$*.*

The root type of the S_{phys} *structure view is the* DevicePhysical *type:*

$$\mathrm{T}_{\mathrm{S}_{phys}}^{root} = \text{DevicePhysical}$$

Finally, the structure view S_{phys} *also defines the* AvailableSlots *resource:*

$$\mathcal{R}_{\mathrm{S}_{phys}} = \{\text{AvailableSlots}\}$$

Structure views have many similarities with feature views, and thus defines similar formal concepts with some nuances:

- A subcomponent definition in $\mathcal{S}_{\mathrm{S}}$ is akin to subfeature definitions, and is defined by a tuple $\langle name, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2 \rangle$ with $(\mathrm{T}_s, \mathrm{T}_t) \in \mathcal{T}_{\mathrm{S}}^c \times \mathcal{T}_{\mathrm{S}}^c$.
- An attribute definition in $\mathcal{D}_{\mathrm{S}}$ is also represented by a tuple $\langle name, \mathrm{T}_s, \mathrm{D}, v, r \rangle$ with $\mathrm{T}_s \in \mathcal{T}_{\mathrm{S}}$.
- The structure taxonomy relation $I_{\mathrm{S}} \subseteq \mathcal{T}_{\mathrm{S}} \times \mathcal{T}_{\mathrm{S}}$ and its reflexive transitive closure $\sqsubseteq_{\mathrm{S}}$ are defined in the same way as teh feature taxonomy relation I_{F} and $\sqsubseteq_{\mathrm{F}}$.
- A compatibility constraint in $\mathcal{C}_{\mathrm{S}}^c$ is also a pair $\langle \mathrm{T}, e^c \rangle$ with $\mathrm{T} \in \mathcal{T}_{\mathrm{S}}$.

Example 27 *The types* Screen *and* TouchScreen *are related in the physical structure view, in that* TouchScreen *is a subtype of* Screen. *We thus have*

$$(\mathrm{TouchScreen}, \mathrm{Screen}) \in I_{\mathrm{S}_{phys}} \text{ and } \mathrm{TouchScreen} \sqsubseteq_{\mathrm{S}_{phys}} \mathrm{Screen}$$

Example 28 *Compatibility constraints defined in structure views are similar to the ones defined in feature views. Consider* $c_{prepaid} \in \mathcal{C}_{\mathrm{S}_{serv}}^c$*:*

$$c_{prepaid} \quad = \quad \langle \mathrm{PhoneSubscription}, e_{prepaid}^c \rangle$$

The compatibility expression $e_{prepaid}^c$ *is used to describe the following condition:*

"If the phone plan is a prepaid plan, the plan does not cover unlimited SMS, unlimited calls during the evening or favourite numbers."

Again, the constraint language (Section 6.2.5) is used to formally define this expression.

Resource use definitions are used to specify how many resources component types use:

Definition 13 (Resource use definition) *A* resource use definition *in* $\mathcal{U}_{\mathrm{S}}$ *is a tuple* $\langle \mathrm{T}, \mathrm{R}, u, e^c \rangle$ *where* $\mathrm{T} \in \mathcal{T}_{\mathrm{S}}^c$ *is the interacting component type and* $\mathrm{R} \in \mathcal{R}_{\mathrm{S}}$ *the resource. The use type* $u \in \{\text{PRODUCES}, \text{CONSUMES}\}$ *provides the type of interaction with the resource, while the use expression* e^c *defines the quantity of the resource used. The context in which the compatibility expression* e^c *is evaluated is the interacting type* T.

Example 29 *The resource use definition* $\upsilon_{slots} \in \mathcal{U}_{\mathrm{S}_{phys}}$ *specified by*

$$\upsilon_{slots} \quad = \quad \langle \text{Motherboard}, \text{AvailableSlots}, \textsc{produces}, e^c \rangle$$

defines that each instance of type Motherboard *produces an amount of* AvailableSlots *equal to the value of the expression* e^c*.*

Association definitions are also used to relate component types with association types, and resemble subcomponent definitions as well.

Definition 14 (Association definition) *An* association definition *in* $\mathcal{A}_{\mathrm{S}}$ *is a tuple* $\langle name, \mathrm{T}_s, \mathrm{T}_t, d, r_1, r_2 \rangle$ *where name is the string name of the association,* $\mathrm{T}_s \in \mathcal{T}_{\mathrm{S}}^c$ *is the source type,* $\mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^a$ *the target association type and* $d \in \{\textsc{none}, \textsc{provides}, \textsc{requires}\}$ *is the direction of the association. The cardinality is given by a minimum* r_1 *and a maximum* r_2*, and is denoted as* $[r_1, r_2]$*.*

Example 30 *The association definition* $\rho_{renderLayout} \in \mathcal{A}_{\mathrm{S}_{serv}}$ *specified by*

$$\rho_{renderLayout} \quad = \quad \langle \text{“}IRender\text{”}, \text{LayoutEngine}, \text{IRender}, \textsc{provides}, 1, 1 \rangle$$

defines the association between the component type LayoutEngine *and the association type* IRender*. The direction* Provides *implies that only a component type defining a* Requires *association with* IRender *can be connected to this component type.*
On the contrary,

$$\rho_{stylusTS} \quad = \quad \langle \text{“}stylus\text{”}, \text{TouchScreen}, \text{Stylus}, \textsc{None}, 0, 1 \rangle$$

defines a bilateral association from the TouchScreen *component type modelling the association between the types representing the device's touch screen and a potential stylus.*

6.2.4 Realisation Views

The third type of view in our modelling framework is realisation views. Realisation views define three different kind of types: item, operation and resource types.

Simple partonomy relations are used to specify realisation subtrees, each rooted by an item or operation type mapped to a specific structure type using a mapping constraints.

Definition 15 (Realisation view) *A* realisation view R *in* $\mathcal{V}^r$ *is defined by a tuple* $\mathrm{R} = \langle \mathrm{S}, \mathcal{T}_\mathrm{R}, \mathcal{S}^i_\mathrm{R}, \mathcal{S}^o_\mathrm{R}, \mathcal{N}_\mathrm{R}, \Omega_\mathrm{R}, \mathcal{C}^m_\mathrm{R}, \mathrm{M} \rangle$ *where*

- $\mathrm{S} \in \mathcal{V}^s$ *is the structure view associated to this realisation view,*
- $\mathcal{T}_\mathrm{R} = \mathcal{T}^i_\mathrm{R} \cup \mathcal{T}^o_\mathrm{R} \cup \mathcal{T}^r_\mathrm{R}$ *is a set of realisation types and contains item types* $\mathcal{T}^i_\mathrm{R}$*, operation types* $\mathcal{T}^o_\mathrm{R}$ *and resource types* $\mathcal{T}^r_\mathrm{R}$*,*
- $\mathcal{S}^i_\mathrm{R}$ *is a set of subitem definitions,*
- $\mathcal{S}^o_\mathrm{R}$ *is a set of suboperation definitions,*
- $\mathcal{N}_\mathrm{R}$ *is a set of successor definition between operation types*
- Ω_R *is a set of resource usage definition between operation and resource types*
- $\mathcal{C}^m_\mathrm{R}$ *is a set of mapping constraints.*
- *Again, the model* M *is repeated for practical purposes.*

Example 31 *The following realisation types belong to the realisation view* R_{phys} *which is partly shown in Figure 5.8 on page 73:*

$$\begin{aligned} \{\#8919, \#239\} &\in \mathcal{T}^i_{\mathrm{R}_{phys}} \\ \{\mathrm{Assemble}, \mathrm{Coating}\} &\in \mathcal{T}^o_{\mathrm{R}_{phys}} \\ \{\mathrm{Cooler}, \mathrm{Furnace}\} &\in \mathcal{T}^r_{\mathrm{R}_{phys}} \end{aligned}$$

$\#8919$ *and* $\#239$ *are item types,* Assemble *and* Coating *are operation types while* Cooler *and* Furnace *are resource types.*

Definition 16 (Subitem definition) *A* subitem definition *in* $\mathcal{S}^i_\mathrm{R}$ *is a tuple* $\langle name, \mathrm{T}_s, \mathrm{T}_t, r \rangle$ *where* $\mathrm{T}_s \in \mathcal{T}^i_\mathrm{R}$ *is the source type and* $\mathrm{T}_t \in \mathcal{T}^i_\mathrm{R}$ *the target type. The cardinality* $r \in \mathbb{N}$ *denotes the number of individuals of this item instantiated during configuration.*

Example 32 *The subitem definition* $\sigma^i_{235} \in \mathcal{S}^i_{\mathrm{R}_{phys}}$ *specified by*

$$\sigma^i_{235} \quad = \quad \langle \text{“}screws\text{”}, \#8919, \#235, 2 \rangle$$

defines the subitem relation between the item types $\#8919$ *and* $\#235$*. The cardinality of 2 means that two items of type* $\#235$ *will be instantiated each time an item of type* $\#8919$ *is instantiated.*

Definition 17 (Suboperation definition) *A* suboperation definition *in* $\mathcal{S}^o_{\mathrm{R}}$ *is a tuple* $\langle name, \mathrm{T}_s, \mathrm{T}_t, r\rangle$ *where* $\mathrm{T}_s \in \mathcal{T}^o_{\mathrm{R}}$ *is the source type and* $\mathrm{T}_t \in \mathcal{T}^o_{\mathrm{R}}$ *the target type. The cardinality* $r \in \mathbb{N}$ *denotes the number of individuals of this operation instantiated during configuration.*

Successor definitions permit to order the instances of operation types present at configuration time:

Definition 18 (Successor definition) *A* successor definition *in* $\mathcal{N}_{\mathrm{R}}$ *is a tuple* $\langle \mathrm{T}_p, \mathrm{T}_s\rangle$ *where* $\mathrm{T}_p \in \mathcal{T}^o_{\mathrm{R}}$ *is the predecessor operation type and* $\mathrm{T}_s \in \mathcal{T}^o_{\mathrm{R}}$ *is the successor operation type.*

Example 33 *The* Assemble *operation type is defined as a successor of the* Coating *operation type, which means that any instance of the* Assemble *type will have to be performed after any instance of* Coating *in the operation route. We thus have* $\eta \in \mathcal{N}_{\mathrm{R}}$ *such that:*

$$\eta \quad = \quad \langle \mathrm{Coating}, \mathrm{Assemble}\rangle$$

Definition 19 (Resource usage definition) *A* resource usage definition *in* Ω_{R} *is a tuple* $\langle name, \mathrm{T}_o, \mathrm{T}_r\rangle$ *where* $\mathrm{T}_o \in \mathcal{T}^o_{\mathrm{R}}$ *is the defining operation type and* $\mathrm{T}_r \in \mathcal{T}^r_{\mathrm{R}}$ *is the type of resource used.*

Definition 20 (Mapping constraint) *A* mapping constraint *in* $\mathcal{C}^m_{\mathrm{R}}$ *is a tuple* $\langle \mathrm{T}_s, \mathrm{T}_r, e^c\rangle$ *where* $\mathrm{T}_s \in \mathcal{T}_{\mathrm{S}}$ *is the origin structure type,* $\mathrm{T}_r \in \mathcal{T}_{\mathrm{R}}$ *is the defining realisation type and* e^c *is a compatibility expression. The context type of* e^c *is the origin type* T_s*.*

Example 34 *Consider a mapping constraint defined in the physical realisation view* R_{phys}*, specifying that*

"The coating operation should only be done for each touch screen that needs an oleophobic coating."

This mapping will be defined as a tuple $c_{coating} \in \mathcal{C}^m_{\mathrm{R}}$*:*

$$c_{coating} \quad = \quad \langle \mathrm{TouchScreen}, \mathrm{Coating}, e^c_{coating}\rangle$$

with $\mathrm{TouchScreen} \in \mathcal{T}_{\mathrm{S}_{phys}}$, $\mathrm{Coating} \in \mathcal{T}_{\mathrm{R}_{phys}}$ *and* $e^c_{coating}$ *being the compatibility expression describing the mapping condition expressed above (written using the constraint language).*

6.2.5 Constraint language

This section defines the constraint language used in our configuration framework. We use the following syntactic categories:

IExp	implementation expressions
CExp	compatibility expressions
Var	expression variables

Constraint expressions are defined according to a specific view $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$ and are specified in conjunction with a context type $T_c \in \mathcal{T}_\mathrm{V}$. The following terms are used:

c	$\in \mathbb{N} \cup \mathbb{B} \cup \mathbb{S}$	integers, booleans and strings
α	$\in \mathcal{D}_\mathrm{V}$	attribute definitions
σ	$\in \mathcal{S}_\mathrm{V}$	subelement definitions
ρ	$\in \mathcal{A}_\mathrm{V}$	association definitions (if $\mathrm{V} \in \mathcal{V}^s$)
T	$\in \mathcal{T}_\mathrm{V}$	concept types

Subelement definitions represent subfeature definitions and concepts types represents feature types if $\mathrm{V} \in \mathcal{V}^f$, or subcomponent definitions and structure types if $\mathrm{V} \in \mathcal{V}^s$. The following operators are used:

op_b	$= op_l \cup op_r \cup op_a$	binary operators
op_l	$\in \{\Rightarrow, \Leftrightarrow, \mathrm{and}, \mathrm{or}, \mathrm{xor}\}$	logical operators
op_r	$\in \{=, \neq, \geq, >, \leq, <\}$	relational operators
op_a	$\in \{+, -, *, \mathrm{mod}\}$	arithmetic operators

The abstract syntax for variables $v \in$ **Var** varies according to the type of view the context type is from. In cases where V is a feature view ($\mathrm{V} \in \mathcal{V}^f$), the syntax of v is given by the following: $v \quad ::= \quad \alpha \mid \sigma \mid \sigma \rightarrow v$

However, if V is a structure view ($\mathrm{V} \in \mathcal{V}^s$), expression variables in compatibility constraints also contain references to association definitions and paths to their attributes: $v \quad ::= \quad \alpha \mid \sigma \mid \sigma \rightarrow v \mid \rho \mid \rho \rightarrow \alpha$

In both cases, expression variables represent qualified paths starting from the context type T_c (if no explicit context type T_i is declared) and following partonomic relations. A more precise characterisation of the final type of each expression variable is given by the type system in Section 6.3.2.

The abstract syntax of the language used for compatibility expressions $e^c \in$ **CExp** is given by the following:

$$\begin{array}{lll} e^c & ::= & e \mid \text{Table}(\,[v_1,\ldots,v_n]\,,[[c_{11},\ldots,c_{n1}]\,,\ldots,[c_{1t},\ldots,c_{nt}]]\,) \\ e & ::= & v \mid c \mid e_1\ op_b\ e_2 \mid \text{not } e \mid (e) \mid v \text{ is T} \mid \text{Count}(v) \mid \\ & & \text{Sum}(\,[v_1,\ldots,v_n]\,) \mid \text{Max}(\,[v_1,\ldots,v_n]\,) \mid \text{Min}(\,[v_1,\ldots,v_n]\,) \end{array}$$

Compatibility expressions can be symbolic expressions e or tables. Symbolic expressions contain variables, constants, binary operation expressions, as well as function calls or *is* expressions, which test for the type of the variable specified.

Implementation expressions are written in a slightly different syntax:

$$\begin{array}{lll} v^i & ::= & \text{T}_i \rightarrow v \\ e^i & ::= & v^i \mid c \mid e^i_1\ op_b\ e^i_2 \mid \text{not } e^i \mid (e^i) \mid v^i \text{ is T} \mid \text{Count}(v^i) \mid \\ & & \text{Sum}(\,\left[v^i_1,\ldots,v^i_n\right]\,) \mid \text{Max}(\,\left[v^i_1,\ldots,v^i_n\right]\,) \mid \text{Min}(\,\left[v^i_1,\ldots,v^i_n\right]\,) \mid \\ & & \text{Present(T)} \end{array}$$

Implementation expressions are just symbolic expressions, but not table constraints, which cannot be used because of the special semantics of these expressions. Implementation expressions are indeed used to specify which combinations of elements from structure views (or parent feature view) are needed to implement a specific combination of features (see Section 5.3). Another important difference lies in the fact that each variable is preceded by an explicit context type T_i that replaces the default context type T_c. This new context type must belong to the structure or feature view implementing the current feature view V. Finally, implementation expressions can also use the $Present(\text{T})$ function to test for the presence of an instance of a specific type from one of the implementing views.

We write $\mathbf{Vars}(e^c)$ (resp. $\mathbf{Vars}(e^i)$) for the set of types and definitions occurring in e^c (resp. e^i), i.e., for all $\delta \in (\mathcal{D}_\text{V} \cup \mathcal{S}_\text{V} \cup \mathcal{A}_\text{V} \cup \mathcal{T}_\text{V})$.

Example 35 *Consider the compatibility expression $e^c_{prepaid}$ from the compatibility constraint $c_{prepaid}$ in Example 28, represented in the constraint language:*

$$e^c_{prepaid} \quad ::= \quad \alpha_{talkTime} = \text{“}prepaid\text{”} \leftrightarrow (\text{not } \alpha_{SMS} \text{ and not } \alpha_{unlimited})$$

where $\alpha_{talkTime}$, α_{SMS}, and $\alpha_{unlimited}$ are attribute definitions for the talkTime*,* SMS*, and* unlimitedEvenings *attributes whose source type is the component type* PhoneSubscription*. We have:*

$$\boldsymbol{Vars}(e^c_{prepaid}) = \{\alpha_{talkTime}, \alpha_{SMS}, \alpha_{unlimited}\}.$$

Consider now the implementation constraint c_{3G} from Example 25. The two expressions e^c_{3G} and e^i_{3G} in the constraint are as follows:

$$\begin{array}{lll} e^c_{3G} & ::= & \alpha_{3G} \\ e^i_{3G} & ::= & \text{RFCard} \rightarrow \alpha_{cardType} = \text{“}GSM/UMTS\text{”} \\ & & \text{and ConnectionMngr} \rightarrow \alpha_{plugin3G} \end{array}$$

The compatibility expression specifies that the feature to implement is the 3G on the phone, as $\alpha_{3G} = \langle \text{“}3G\text{”}, \text{Phone}, \{\text{true}, \text{false}\}, \text{VISIBLE}, 1\rangle$. The implementation constraint involves two variable paths, one starting from the context type $\text{RFCard} \in \mathcal{T}_{\text{S}_{phys}}$ with $\alpha_{cardType} = \langle \text{“}cardType\text{”}, \text{RFCard}, ...\rangle$ and the other one starting from the context type $\text{ConnectionMngr} \in \mathcal{T}_{\text{S}_{soft}}$ with $\alpha_{plugin3G} = \langle \text{“}plugin3G\text{”}, \text{ConnectionMngr}, ...\rangle$. We have here:

$$\boldsymbol{Vars}(e^c_{3G}) = \{\alpha_{3G}\} \text{ and } \boldsymbol{Vars}(e^i_{3G}) = \{\alpha_{cardType}, \alpha_{plugin3G}\}.$$

6.3 Formalising ProCoLa

We now take a look back at the ProCoLa modelling language defined in Section 6.1. In this section we describe the process of translating configuration models written in ProCoLa into the formalism presented in the previous section. This translation is a new step towards providing a clear and complete semantics for ProCoLa: the first semantic checks are performed during the translation from ProCoLa to the formalism. Moreover, we define in Section 6.3.2 a type system for the constraint language used in the formalism, as well as a series of rules in Section 6.3.3 that ensure the well-formedness of the formalised models.

6.3.1 From ProCoLa to the formalism

We present now the operations responsible for the translation process. For the sake of brevity, we only show the translation for feature views. The formalisation rules for structure and realisation views can be found in Appendix B.

Expressions are not covered in this Section either, due to the many similarities between the ProCoLa expression syntax and the constraint language defined in the previous section, which makes the translation trivial.

In order to interpret units and constants, we use two auxiliary mappings defined as follow:

$$
\begin{array}{ll}
u \in \mathit{Units}_{\mathrm{F}} & ::= Id_u \longmapsto D_u \text{ with } Id_u \in \mathbb{S}, D_u \in \mathcal{P}(\mathbb{N} \cup \mathbb{B} \cup \mathbb{S}) \\
c \in \mathit{Const}_{\mathrm{F}} & ::= Id_c \longmapsto \langle c, Id_u \rangle \text{ with } Id_c \in \mathbb{S}, c \in \mathbb{N} \cup \mathbb{B} \cup \mathbb{S}, Id_u \in \mathbb{S} \cup \{\epsilon\}
\end{array}
$$

The units mapping $\mathit{Units}_{\mathrm{F}}$ maps a string, the unit name Id_u, to a set of constant values, the domain of the unit. The constants mapping $\mathit{Const}_{\mathrm{F}}$ maps a string, the constant name Id_c, to a pair composed by the constant value c and a potential unit name Id_u (ϵ is used when no specific unit is assigned to the constant).

Tables 6.5, 6.6, and 6.7 show the translation rules in the context of a feature view $\mathrm{F} = \langle \mathcal{T}_{\mathrm{F}}, \mathrm{T}_{\mathrm{F}}^{root}, \mathcal{S}_{\mathrm{F}}, I_{\mathrm{F}}, \mathcal{D}_{\mathrm{F}}, \mathit{Ref}_{\mathrm{F}}, \mathcal{C}_{\mathrm{F}}^{c}, \mathcal{C}_{\mathrm{F}}^{i}, \mathrm{M} \rangle$, with $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, \mathit{Sp}_{\mathrm{M}} \rangle$. The translation operation $\rightsquigarrow_{\mathrm{F}}$ takes ProCoLa terms and either converts them to a formal element or specifies a boolean condition that must be ensured for the rule to be satisfied.

Rule $[C_1]$ is for example a basic conversion rule: it takes a constant value c in ProCoLa and returns a pair $\langle c, \epsilon \rangle$, where ϵ is used since the constant value has no unit. On the other hand, rule $[C_2]$ takes a constant identifier Id_c and return the pair $\langle c, u \rangle$, assuming that there exists a mapping in $\mathit{Const}_{\mathrm{F}}$ that maps Id_c to a value c and a unit identifier u. This assumption has to be satisfied for the rule to be usable: it thus checks that a corresponding constant has correctly been declared.

Rule $[\mathit{FeatureView}_1]$ from Table 6.7 is an example of a rule that contains a boolean condition that must be ensured: in case F' is a feature view and declared as a parent view of the newly declared view F, the pair $(\mathrm{F}, \mathrm{F}')$ must be added to the specialisation function Sp_{M}.

Moreover, we use ProCoLa terms in brackets and associated with a boolean variable b (e.g., $[\texttt{ref}]_b$) to describe the translation for certain optional terms. For example, rule $[\mathit{AttrDecl}_1]$ relates to attribute declarations and the `ref` keyword. The boolean variable b represents whether or not the `ref` keyword is stated in the attribute declaration. A boolean condition using b is specified with the rule: in case of rule $[\mathit{AttrDecl}_1]$, if ref is present in the declaration, the rule is valid only if there exists an element α_2 for which the attribute α declared is a reference to, i.e., $\alpha \prec_{\mathrm{F}}^{a} \alpha_2$. On the other hand, if `ref` is not in the declaration, there should be no such α_2, in order to avoid collision between two attribute declarations with the same name and belonging to two types with one refining the other.

$$[C_1] \quad : \quad \texttt{c} \rightsquigarrow_{\mathrm{F}} \langle c, \epsilon \rangle$$

$$[C_2] \quad : \quad Id_c \rightsquigarrow_{\mathrm{F}} \langle c, u \rangle \qquad \text{if } (Id_c \longmapsto \langle c, u \rangle) \in Const_{\mathrm{F}}$$

$$[Type_1] \quad : \quad Id_u \rightsquigarrow_{\mathrm{F}} \langle D_u, Id_u \rangle \qquad \text{if } (Id_u \longmapsto D_u) \in Units_{\mathrm{F}}$$

$$[Type_{2-4}] \quad : \quad \texttt{integer} \rightsquigarrow_{\mathrm{F}} \langle \mathbb{N}, \epsilon \rangle \quad \texttt{boolean} \rightsquigarrow_{\mathrm{F}} \langle \mathbb{B}, \epsilon \rangle \quad \texttt{enum} \rightsquigarrow_{\mathrm{F}} \langle \mathbb{S}, \epsilon \rangle$$

$$[DomReduc_1] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{F}} \langle c_i, u \rangle}{\{C_1, ..., C_n\} \rightsquigarrow_{\mathrm{F}} \langle D = \{c_1, ..., c_n\}, u \rangle}$$

$$[DomReduc_2] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{F}} \langle c_i, u \rangle}{\texttt{[}C_1..C_n\texttt{]} \rightsquigarrow_{\mathrm{F}} \langle D = \{n / c_1 \leq n \leq c_n\}, u \rangle}$$

$$[DomReduc_3] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{F}} \langle c, u \rangle}{[\texttt{-inf}..C] \rightsquigarrow_{\mathrm{F}} \langle D = \{n / n \leq c\}, u \rangle}$$

$$[DomReduc_4] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{F}} \langle c, u \rangle}{[C..\texttt{inf}] \rightsquigarrow_{\mathrm{F}} \langle D = \{n / c \leq n\}, u \rangle}$$

$$[TypeDecl] \quad : \quad \frac{Type \rightsquigarrow_{\mathrm{F}} \langle D_1, u \rangle;\ DomReduc \rightsquigarrow_{\mathrm{F}} \langle D_2, u \rangle}{TypeDomReduc \rightsquigarrow_{\mathrm{F}} \langle D_2, u \rangle} \qquad \text{if } D_2 \subseteq D_1$$

$$[ConstrDecl_1] \quad : \quad \frac{CSymExp \rightsquigarrow_{\mathrm{F}} e^c}{\{CSymExp\} \rightsquigarrow_{\mathrm{F}} \langle \mathrm{T}_c, e^c \rangle}$$

$$[ConstrDecl_2] \quad : \quad \frac{ConstrVal \rightsquigarrow_{\mathrm{F}} e^c}{[Id]\texttt{:[description:[c;]]}\ ConstrVal \rightsquigarrow_{\mathrm{F}} \langle \mathrm{T}_c, e^c \rangle}$$

$$[ImplDecl_1] \quad : \quad \frac{ImplExp \rightsquigarrow_{\mathrm{F}} \langle e^c, e^i, \mathrm{Op} \rangle}{\{ImplExp\} \rightsquigarrow_{\mathrm{F}} \langle \mathrm{T}_c, e^c, e^i, \mathrm{Op} \rangle}$$

$$[ImplDecl_2] \quad : \quad \frac{ImplExp \rightsquigarrow_{\mathrm{F}} \langle e^c, e^i, \mathrm{Op} \rangle}{[Id]\texttt{:[description:[c;]]}\ \texttt{value:}ImplExp \rightsquigarrow_{\mathrm{F}} \langle \mathrm{T}_c, e^c, e^i, \mathrm{Op} \rangle}$$

$$[Visbility] \quad : \quad \texttt{readonly} \rightsquigarrow_{\mathrm{F}} \textsc{Readonly} \quad \texttt{hidden} \rightsquigarrow_{\mathrm{F}} \textsc{Hidden} \quad \epsilon \rightsquigarrow_{\mathrm{F}} \textsc{Visible}$$

$$[AttrDecl_1] \quad : \quad [\texttt{ref}]_b\ Id_a\ \ \ldots \rightsquigarrow_{\mathrm{F}} \alpha = \langle Id_a, ... \rangle \qquad \text{if } b \Leftrightarrow \exists \alpha_2 / \alpha \prec^a_{\mathrm{F}} \alpha_2$$

Table 6.5: Formalisation of ProCoLa feature view

$$[AttrDecl_2] \quad : \quad \frac{Visibility \rightsquigarrow_{\mathrm{F}} v;\ TypeDecl \rightsquigarrow_{\mathrm{F}} \langle D, u\rangle}{Visibility\ Id_a\texttt{:}TypeDecl\ldots \rightsquigarrow_{\mathrm{F}} \alpha = \langle Id_a, \mathrm{T}_a, D, v, 1\rangle}$$

$$[AttrDecl_3] \quad : \quad \frac{Visibility \rightsquigarrow_{\mathrm{F}} v;\ TypeDecl \rightsquigarrow_{\mathrm{F}} \langle D, u\rangle;\ C \rightsquigarrow_{\mathrm{F}} \langle r, \epsilon\rangle}{Visibility\ Id_a\texttt{[}C\texttt{]:}TypeDecl\ldots \rightsquigarrow_{\mathrm{F}} \alpha = \langle Id_a, \mathrm{T}_a, D, v, r\rangle}$$

$$[SubFDecl_1] \quad : \quad [\texttt{ref}]_b Id_s\ldots \rightsquigarrow_{\mathrm{F}} \sigma = \langle Id_s, ...\rangle \qquad \text{if } b \Leftrightarrow \exists \sigma_2/\sigma \prec^s_{\mathrm{F}} \sigma_2$$

$$[SubFDecl_2] \quad : \quad Id_s\texttt{:}\mathrm{T}_t\ldots \rightsquigarrow_{\mathrm{F}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, 1, 1\rangle \qquad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{F}}$$

$$[SubFDecl_3] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{F}} \langle r, \epsilon\rangle}{Id_s\texttt{[}C\texttt{]:}\mathrm{T}_t\ldots \rightsquigarrow_{\mathrm{F}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, r, r\rangle} \quad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{F}}$$

$$[SubFDecl_4] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{F}} \langle r_i, \epsilon\rangle}{Id_s\texttt{[}C_1\texttt{..}C_2\texttt{]:}\mathrm{T}_t\ldots \rightsquigarrow_{\mathrm{F}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2\rangle} \text{ if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{F}}$$

$$[FTypeDecl_1] \quad : \quad [\texttt{refined}]_b\texttt{...featureType T...} \rightsquigarrow_{\mathrm{F}} \mathrm{T} \in \mathcal{T}_{\mathrm{F}} \text{ if } b \Leftrightarrow \exists \mathrm{F}_P/\mathrm{T} \in \mathcal{T}_{\mathrm{F}_P} \& (\mathrm{F}, \mathrm{F}_P) \in Sp_{\mathrm{M}}$$

$$[FTypeDecl_2] \quad : \quad [\texttt{Root}]\texttt{...featureType T...} \rightsquigarrow_{\mathrm{F}} \mathrm{T}^{root}_{\mathrm{F}} = \mathrm{T}$$

$$[FTypeDecl_3] \quad : \quad \texttt{...abstract...featureType T...} \rightsquigarrow_{\mathrm{F}} \mathrm{T} \in \mathcal{T}^{Ab}_{\mathrm{F}}$$

$$[FTypeDecl_4] \quad : \quad \texttt{...featureType T subtypeOf } \mathtt{T_1}, ..., \mathtt{T_n}\texttt{...} \rightsquigarrow_{\mathrm{F}} (\mathrm{T}_1, \mathrm{T}), ..., (\mathrm{T}_n, \mathrm{T}) \in I_{\mathrm{F}} \qquad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{F}}$$

$$[FTypeDecl_5] \quad : \quad \frac{A_i \rightsquigarrow_{\mathrm{F}} \alpha_i = \langle ..., \mathrm{T}_{ai}, ...\rangle}{\texttt{...featureType T...attributes : } \mathtt{A_1}, ..., \mathtt{A_n}\texttt{...} \rightsquigarrow_{\mathrm{F}} \alpha_i = \langle ..., \mathrm{T}, ...\rangle \in \mathcal{D}_{\mathrm{F}}}$$

$$[FTypeDecl_6] \quad : \quad \frac{S_i \rightsquigarrow_{\mathrm{F}} \sigma_i = \langle ..., \mathrm{T}_{si}, \mathrm{T}_t, ...\rangle}{\texttt{...featureType T...subfeatures : } \mathtt{S_1}, ..., \mathtt{S_n}\texttt{...} \rightsquigarrow_{\mathrm{F}} \sigma_i = \langle ..., \mathrm{T}, \mathrm{T}_t, ...\rangle \in \mathcal{S}_{\mathrm{F}}}$$

$$[FTypeDecl_7] \quad : \quad \frac{Cons_i \rightsquigarrow_{\mathrm{F}} C_i = \langle ..., \mathrm{T}_{ci}, ...\rangle}{\texttt{...featureType T...constraints : } \mathtt{Cons^c_1}, ..., \mathtt{Cons^c_n}\texttt{...} \rightsquigarrow_{\mathrm{F}} C_i = \langle ..., \mathrm{T}, ...\rangle \in \mathcal{C}_{\mathrm{F}^c}}$$

$$[FTypeDecl_8] \quad : \quad \frac{I_i \rightsquigarrow_{\mathrm{F}} I_i = \langle ..., \mathrm{T}_{ci}, ...\rangle}{\texttt{...featureType T...implementation : } \mathtt{I^i_1}, ..., \mathtt{I^i_n}\texttt{...} \rightsquigarrow_{\mathrm{F}} I_i = \langle ..., \mathrm{T}, ...\rangle \in \mathcal{C}_{\mathrm{F}^i}}$$

Table 6.6: Formalisation of ProCoLa feature view (continued)

$$[ConstDecl_1] \quad : \quad Id_c{:=}\ \texttt{c} \rightsquigarrow_{\mathrm{F}} (Id_c \longmapsto \langle c, \epsilon \rangle) \in Const_{\mathrm{F}}$$

$$[ConstDecl_2] \quad : \quad Id_c : Id_u{:=}\ \texttt{c} \rightsquigarrow_{\mathrm{F}} (Id_c \longmapsto \langle c, Id_u \rangle) \in Const_{\mathrm{F}}$$

$$[UnitDecl] \quad : \quad \frac{TypeDecl \rightsquigarrow_{\mathrm{F}} \langle D_u, \epsilon \rangle}{Id_u : TypeDecl \rightsquigarrow_{\mathrm{F}} (Id_u \longmapsto D_u) \in Units_{\mathrm{F}}}$$

$$[FeatureView_1] \quad : \quad \texttt{featureView F specialisationOf F}'\ldots \rightsquigarrow_{\mathrm{F}} (\mathrm{F}, \mathrm{F}') \in Sp_{\mathrm{M}} \quad \text{if } \mathrm{F}' \in \mathcal{V}^f$$

$$[FeatureView_2] \quad : \quad \texttt{featureView F}\ldots \rightsquigarrow_{\mathrm{F}} \mathrm{F} \in \mathcal{V}^f$$

Table 6.7: Formalisation of ProCoLa feature view (end)

6.3.2 Type System

Now that ProCoLa models can be formalised, we establish a type system for the constraint language defined in Section 6.2.5 and used in the formalism. The translation operations defined above provided several type checks, e.g., for subfeature declarations or feature subtypes. The type system described in this section is yet a new step towards typing the ProCoLa language, by checking variable types in constraint expressions and providing valid types for them.

We first introduce the types and type environments used in the type system, and then present the typing judgements for constraint expressions.

6.3.2.1 Types

The type system is established for a model $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_{\mathrm{M}} \rangle$. Let us first introduce the notion of *types* and *type environments*.

$$\tau \quad \in \mathbf{Type} \quad \text{types}$$

$$\Gamma_{\mathrm{T}} \quad \in \mathbf{TEnv} \quad \text{type environments}$$

We shall assume that types are separated into base types and concepts types:

$$\begin{aligned}
\mathbf{Type} &= \mathbf{Type}_{base} \cup \mathbf{Type}_{concepts} \\
\mathbf{Type}_{base} &= \{\texttt{int}, \texttt{bool}, \texttt{string}\} \\
\mathbf{Type}_{concepts} &= \bigcup_{\mathrm{F} \in \mathcal{V}^f} \mathcal{T}_{\mathrm{F}} \cup \bigcup_{\mathrm{S} \in \mathcal{V}^s} \mathcal{T}_{\mathrm{S}}
\end{aligned}$$

where `int`, `bool`, and `string` are the only three kinds of base types, and the framework concept types (feature types, structure types, and realisation types) are represented by $\mathbf{Type}_{concepts}$. Each constant $c \in \mathbb{N} \cup \mathbb{B} \cup \mathbb{S}$ has a type that we shall denote τ_c; for example, *true* has type $\tau_{\text{true}} = \texttt{bool}$ and 13 has type $\tau_{13} = \texttt{int}$. Each string s has the type $\tau_s = \texttt{string}$. Each binary operator op_b will expect two arguments of types $\tau^1_{op_b}$ and $\tau^2_{op_b}$, respectively, and will return a result of type τ_{op_b}. For example, the relational operation $\geq$ expects two arguments of type `int` and gives a result of type `bool`.

Finally, the domain D of an attribute definition α is said to be *valid* iff all constant literals in D have the same base type τ, in which case D has a type $\tau_{\mathrm{D}} = \tau$.

A type environment is induced by a *context type* $\mathrm{T} \in \mathcal{T}_{\mathrm{V}}$ (with $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$), which specifies in which context it is is valid. It is given by a function

$$\Gamma_{\mathrm{T}} : (\mathcal{D}_{\mathrm{V}} \to \mathbf{Type}_{base}) \oplus (\mathcal{S}_{\mathrm{V}} \cup \mathcal{A}_{\mathrm{V}} \to \mathbf{Type}_{concepts})$$

where $\oplus$ represents the function union operation.

This means that Γ_{T} maps attribute declarations to base types, and subelement declarations to concept types.
$\forall \mathrm{T} \in \mathcal{T}_{\mathrm{V}}, \forall \alpha \in \mathcal{D}_{\mathrm{V}}, \forall \sigma \in \mathcal{S}_{\mathrm{V}}$:

$$\begin{aligned}
\Gamma_{\mathrm{T}}(\alpha) &= \tau_{\mathrm{D}} \quad \text{iff } \alpha = \langle \mathrm{T}', \mathrm{D}, v, r \rangle \text{ and } \mathrm{T} \sqsubseteq_{\mathrm{V}} \mathrm{T}' \\
\Gamma_{\mathrm{T}}(\sigma) &= \mathrm{T}_t \quad \text{iff } \sigma = \langle \mathrm{T}', \mathrm{T}_t, r_1, r_2 \rangle \text{ and } \mathrm{T} \sqsubseteq_{\mathrm{V}} \mathrm{T}'
\end{aligned}$$

Moreover, if V is a structure view ($\mathrm{V} \in \mathcal{V}^s$), association definitions are also mapped to concept types:
$\forall \rho \in \mathcal{A}_{\mathrm{V}}, \Gamma_{\mathrm{T}}(\rho) = \mathrm{T}_t$ iff $\rho = \langle \mathrm{T}', \mathrm{T}_t, d, r_1, r_2 \rangle$ and $\mathrm{T} \sqsubseteq_{\mathrm{V}} \mathrm{T}'$.

6.3.2.2 Typing judgements

For a feature or structure view $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$, the general form of a typing judgement (in the context of $\mathrm{T} \in \mathcal{T}_\mathrm{V}$) $\Gamma_\mathrm{T} \vdash_V e : \tau$ says that the expression e has type τ assuming that any attribute, subelement, or association definition has the type given by Γ_T. The axioms and rules for the judgements are listed in Table 6.8, in the context of a view $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$.

Example 36 *Consider the two constraint expressions e^c_{3G} and e^i_{3G} defined in Example 35 by:*

$$\begin{array}{lcl} e^c_{3G} & ::= & \alpha_{3G} \\ e^i_{3G} & ::= & \mathrm{RFCard} \rightarrow \alpha_{cardType} = \text{“}GSM/UMTS\text{”} \\ & & \text{and } \mathrm{ConnectionMngr} \rightarrow \alpha_{plugin3G} \end{array}$$

As the context type for both expressions is Phone, we have:

$$\Gamma_\mathrm{Phone} \vdash_{F_{base}} \alpha_{3G} : \texttt{bool} \textit{ as } \Gamma_\mathrm{Phone}(\alpha_{3G}) = \texttt{bool}$$

which gives

$$\Gamma_\mathrm{Phone} \vdash_{F_{base}} e^c_{3G} : \texttt{bool}$$

Moreover,

$$\begin{array}{lll} \Gamma_\mathrm{RFCard} & \vdash_{F_{base}} & \alpha_{cardType} : \texttt{string} \\ \Gamma_\mathrm{Phone} & \vdash_{F_{base}} & \mathrm{RFCard} \rightarrow \alpha_{cardType} = \text{“}GSM/UMTS\text{”} : \texttt{bool} \end{array}$$

and

$$\begin{array}{lll} \Gamma_\mathrm{ConnectionMngr} & \vdash_{F_{base}} & \alpha_{plugin3G} : \texttt{bool} \\ \Gamma_\mathrm{Phone} & \vdash_{F_{base}} & \mathrm{ConnectionMngr} \rightarrow \alpha_{plugin3G} : \texttt{bool} \end{array}$$

which finally gives

$$\Gamma_\mathrm{Phone} \vdash_{F_{base}} e^i_{3G} : \texttt{bool}$$

$$[con] \quad : \quad \Gamma_{\mathrm{T}} \vdash_V c : \tau_c$$

$$[att] \quad : \quad \Gamma_{\mathrm{T}} \vdash_V \alpha : \tau_{\mathrm{D}} \qquad \text{if } \Gamma_{\mathrm{T}}(\alpha) = \tau_{\mathrm{D}}$$

$$[sub] \quad : \quad \Gamma_{\mathrm{T}} \vdash_V \sigma : \mathrm{T}_t \qquad \text{if } \Gamma_{\mathrm{T}}(\sigma) = \mathrm{T}_t$$

$$[asso] \quad : \quad \Gamma_{\mathrm{T}} \vdash_V \rho : \mathrm{T}_t \qquad \text{if } \Gamma_{\mathrm{T}}(\rho) = \mathrm{T}_t$$

$$[ipath] \quad : \quad \frac{\Gamma_{\mathrm{T}_1} \vdash_V v : \tau}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{T}_1 \rightarrow v : \tau}$$

$$[spath] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V \sigma : \mathrm{T}' \qquad \Gamma_{\mathrm{T}'} \vdash_V v : \tau}{\Gamma_{\mathrm{T}} \vdash_V \sigma \rightarrow v : \tau}$$

$$[apath] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V \rho : \mathrm{T}' \qquad \Gamma_{\mathrm{T}'} \vdash_V \alpha : \tau}{\Gamma_{\mathrm{T}} \vdash_V \rho \rightarrow \alpha : \tau}$$

$$[op] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V e_1 : \tau^1_{op_b} \qquad \Gamma_{\mathrm{T}} \vdash_V e_2 : \tau^2_{op_b}}{\Gamma_{\mathrm{T}} \vdash_V e_1 \ op_b \ e_2 : \tau_{op_b}}$$

$$[not] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V e : \texttt{bool}}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{not}\ e : \texttt{bool}}$$

$$[par] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V e : \tau}{\Gamma_{\mathrm{T}} \vdash_V (e) : \tau}$$

$$[is] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V v : \tau}{\Gamma_{\mathrm{T}} \vdash_V v \ \mathrm{is}\ \mathrm{T}_1 : \texttt{bool}} \qquad \text{if } \tau \in \mathbf{Type}_{concepts}$$

$$[count] \quad : \quad \frac{\Gamma_{\mathrm{T}} \vdash_V v : \tau}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{Count}(v) : \texttt{int}} \qquad \text{if } \tau \in \mathbf{Type}_{concepts}$$

$$[sum] \quad : \quad \frac{\forall i : \Gamma_{\mathrm{T}} \vdash_V v_i : \texttt{int}}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{Sum}(\,[v_1, \ldots, v_n]\,) : \texttt{int}}$$

$$[max] \quad : \quad \frac{\forall i : \Gamma_{\mathrm{T}} \vdash_V v_i : \texttt{int}}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{Max}(\,[v_1, \ldots, v_n]\,) : \texttt{int}}$$

$$[min] \quad : \quad \frac{\forall i : \Gamma_{\mathrm{T}} \vdash_V v_i : \texttt{int}}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{Min}(\,[v_1, \ldots, v_n]\,) : \texttt{int}}$$

$$[pres] \quad : \quad \Gamma_{\mathrm{T}} \vdash_V \mathrm{Present}(\mathrm{T}) : \texttt{bool}$$

$$[table] \quad : \quad \frac{\forall i : \Gamma_{\mathrm{T}} \vdash_V v_i : \tau_i \qquad \forall i, j, \Gamma_{\mathrm{T}} \vdash_V c_{ij} : \tau_i}{\Gamma_{\mathrm{T}} \vdash_V \mathrm{Table}(\,[v_1, \ldots, v_n]\,, [[c_{11}, \ldots, c_{n1}]\,, \ldots, [c_{1t}, \ldots, c_{nt}]]\,) : \texttt{bool}}$$

Table 6.8: The type system for constraint expressions

6.3.3 Well-formedness

We now define a set of rules that ensure the well-formedness of the whole configuration model. We use in particular the type system established in the previous section to define well-formedness for constraint expressions.

Definition 21 *For a feature or structure view* $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$*, a* compatibility constraint $\langle \mathrm{T}, e^c \rangle \in \mathcal{C}^c_\mathrm{V}$ *is* well-formed *if* $\Gamma_\mathrm{T} \vdash_V e^c : \texttt{bool}$.

A compatibility expression $e^c \in \mathbf{CExp}$ is thus well-formed if it can be evaluated to a boolean by the type environment induced by T.

Definition 22 *For a feature or structure view* $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$*, an* attribute definition $\langle name, \mathrm{T}, \mathrm{D}, v, r \rangle \in \mathcal{D}_\mathrm{V}$ *is* well-formed *iff D is* valid *and* $r > 0$.

For a (feature or structure) taxonomy relation, several rules must be satisfied for it to be well-formed: there can be no loop in the partonomy, and each abstract type must at least have one subtype.

Also, in case of a structure taxonomy relation, only similar structure types can be part of the same taxonomy tree, i.e., an association type cannot be the subtype of a component type (and vice versa).

Definition 23 *For a feature or structure view* $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$*, a feature or structure* taxonomy relation $I_\mathrm{V} \subseteq \mathcal{T}_\mathrm{V} \times \mathcal{T}_\mathrm{V}$ *is* well-formed *iff:*

- $\nexists(\mathrm{T}, \mathrm{T}') \in \mathcal{T}_\mathrm{V}$*, s.t.* $(\mathrm{T} \neq \mathrm{T}') \wedge (\mathrm{T} \sqsubseteq_\mathrm{V} \mathrm{T}') \wedge (\mathrm{T}' \sqsubseteq_\mathrm{V} \mathrm{T})$
- $\forall \mathrm{T} \in \mathcal{T}^{Ab}_\mathrm{V}$*,* $\exists \mathrm{T}' \in (\mathcal{T}_\mathrm{V} \setminus \mathcal{T}^{Ab}_\mathrm{V})$ *s.t.* $(\mathrm{T}', \mathrm{T}) \in I_\mathrm{V}$

Moreover, if $V \in \mathcal{V}^s$*,*

$$\forall(\mathrm{T}, \mathrm{T}') \in I_\mathrm{V}, \ (\mathrm{T} \in \mathcal{T}^c_\mathrm{V} \Leftrightarrow \mathrm{T}' \in \mathcal{T}^c_\mathrm{V}) \wedge (\mathrm{T} \in \mathcal{T}^a_\mathrm{V} \Leftrightarrow \mathrm{T}' \in \mathcal{T}^a_\mathrm{V})$$

Subelement definitions (subcomponent or subfeature) must be have a target type different from the source type (or one of its subtypes) to be well-formed, and the minimum cardinality cannot be less than 0 or greater than the maximum cardinality of the relation.

Definition 24 *For a feature or structure view* $\mathrm{V} \in \mathcal{V}^f \cup \mathcal{V}^s$*, a* subfeature or subcomponent definition $\langle name, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2 \rangle \in \mathcal{S}_\mathrm{V}$ *is* well-formed *iff:*

$$(\mathrm{T}_t \not\sqsubseteq_\mathrm{V} \mathrm{T}_s) \wedge (0 \leq r_1 \leq r_2)$$

The well-formedness of implementation constraints is ensured if both the child expression and the parent expression can be evaluated to basic types by the type environment induced by constraint's context type, if these types are compatible with the binding operator and the return type of this operator (and thus the constraint) is boolean.

Definition 25 *An* implementation constraint $\langle \mathrm{T}, e^c_\mathrm{C}, e^i_\mathrm{P}, \mathrm{Op} \rangle$ *in* $\mathcal{C}^i_\mathrm{F}$ *for a feature view* $\mathrm{F} \in \mathcal{V}^f$ *is* well-formed *iff* $\exists \tau^1_\mathrm{Op}, \tau^2_\mathrm{Op} \in \boldsymbol{Type}_{basic}$ *s.t.*

$$(\Gamma_\mathrm{T} \vdash_F e^c_\mathrm{C} : \tau^1_\mathrm{Op}) \wedge (\Gamma_\mathrm{T} \vdash_F e^i_\mathrm{P} : \tau^2_\mathrm{Op}) \wedge (\Gamma_\mathrm{T} \vdash_F e^c_\mathrm{C}\ \mathrm{Op}\ e^i_\mathrm{P} : \texttt{bool}).$$

Reference relations must involved proper subfeature and attribute definitions refinements, as described in Section 5.4.

Definition 26 *For a feature view* $\mathrm{F} \in \mathcal{V}^f$*, the* subfeature reference relation $\prec^s_\mathrm{F}$ *is* well-formed *iff, for two subfeature definitions* $\sigma_1 = \langle name_1, \mathrm{T}_{s1}, \mathrm{T}_{t1}, r_{11}, r_{21} \rangle$ *and* $\sigma_2 = \langle name_2, \mathrm{T}_{s2}, \mathrm{T}_{t2}, r_{12}, r_{22} \rangle$ *such that* $\sigma_2 \prec^s_\mathrm{F} \sigma_1$*:*

$$(name_1 = name_2) \wedge (\mathrm{T}_{t1} \sqsubseteq_\mathrm{F} \mathrm{T}_{t2}) \wedge (r_{11} \geq r_{12}) \wedge (r_{21} \leq r_{22}).$$

In a similar way, the attribute reference relation $\prec^a_\mathrm{F}$ *is* well-formed *iff, for two attribute definitions* $\alpha_1 = \langle name_1, \mathrm{T}_1, \mathrm{D}_1, v_1, r_1 \rangle$ *and* $\alpha_2 = \langle name_2, \mathrm{T}_2, \mathrm{D}_2, v_2, r_2 \rangle$ *such that* $\alpha_2 \prec^a_\mathrm{F} \alpha_1$*:*

$$(name_1 = name_2) \wedge (\mathrm{D}_2 \subseteq \mathrm{D}_1) \wedge (r_2 \leq r_1) \wedge (v_2 \leq v_1).$$

Moreover, the reference relation $\prec_\mathrm{F}$ *is* well-formed *iff it is the function union of two well-formed relations* $\prec^s_\mathrm{F}$ *and* $\prec^a_\mathrm{F}$*.*

Finally, the direct reference function Ref_F *is* well-formed *iff* $\prec_\mathrm{F}$ *is well-formed.*

We now define the well-formedness rule for feature views.

Definition 27 *A* feature view $\mathrm{F} = \langle \mathcal{T}_\mathrm{F}, \mathrm{T}_\mathrm{F}^{root}, \mathcal{S}_\mathrm{F}, I_\mathrm{F}, \mathcal{D}_\mathrm{F}, \prec_\mathrm{F}, \mathcal{C}_\mathrm{F}^c, \mathcal{C}_\mathrm{F}^i \rangle$ *is* well-formed *iff:*

- *all the elements in* $\mathcal{S}_\mathrm{F}, \mathcal{D}_\mathrm{F}, \mathcal{C}_\mathrm{F}^c$ *and* $\mathcal{C}_\mathrm{F}^i$ *are well-formed*
- *the relation* I_F *and the function* $\prec_\mathrm{F}$ *are well-formed*
- *the names used in the attribute and subfeature definitions are unique for a given source type* T_s*, i.e.,*

$$\nexists(\delta_1 = \langle n_1, \mathrm{T}_s, \ldots\rangle, \delta_2 = \langle n_2, \mathrm{T}_s, \ldots\rangle) \in (\mathcal{S}_\mathrm{F} \cup \mathcal{D}_\mathrm{F}) \text{ s.t. } n_1 = n_2$$

- *the root type* $\mathrm{T}_\mathrm{F}^{root}$ *belongs to* F *or the closest parent view of F containing a root type, i.e., if* $\mathrm{T}_\mathrm{F}^{root} \in \mathrm{F_P} \neq \mathrm{F}$*,*

$$\nexists \mathrm{F}' \text{ s.t. } (\mathrm{F}, \mathrm{F}') \in Sp_\mathrm{M}^* \wedge (\mathrm{F}', \mathrm{F_P}) \in Sp_\mathrm{M}^* \wedge \mathrm{T}_{\mathrm{F}'}^{root} \in \mathcal{T}_{\mathrm{F}'}$$

- *there is no loop in the partonomy tree, i.e.,*

$$\begin{array}{l}\forall\ (\mathrm{T}, \mathrm{T}') \text{ with } (\mathrm{T} \sqsubseteq_\mathrm{F} \mathrm{T}') \vee (\mathrm{T}' \sqsubseteq_\mathrm{F} \mathrm{T}),\ \nexists(\mathrm{T}_1, \ldots, \mathrm{T}_n) \text{ s.t.,} \\ \quad \mathrm{T}_1 = \mathrm{T}, \mathrm{T}_n = \mathrm{T}' \text{ and} \\ \quad \forall \mathrm{T}_i,\ \mathrm{T}_{i+1}, \\ \qquad (\mathrm{T}_i \sqsubseteq_\mathrm{F} \mathrm{T}_{i+1}) \vee (\mathrm{T}_{i+1} \sqsubseteq_\mathrm{F} \mathrm{T}_i) \vee (\exists \sigma_i = \langle \ldots, \mathrm{T}_i, \mathrm{T}_{i+1}, \ldots\rangle \in \mathcal{S}_\mathrm{F})\end{array}$$

Before establishing the well-formedness of structure views, we have to define what is a well-formed association definition. This is similar to the well-formedness of subcomponent definitions.

Definition 28 *For a structure view* $\mathrm{S} \in \mathcal{V}^s$*, an* association definition *defined by* $\langle name, \mathrm{T}_s, \mathrm{T}_t, d, r_1, r_2\rangle \in \mathcal{A}_\mathrm{S}$ *is* well-formed *iff:*

$$(\mathrm{T}_t \not\sqsubseteq_\mathrm{V} \mathrm{T}_s) \wedge (0 \leq r_1 \leq r_2)$$

Definition 29 *A* structure view $\mathrm{S} = \langle \mathcal{T}_\mathrm{S}, \mathrm{T}_\mathrm{S}^{root}, \mathcal{S}_\mathrm{S}, \mathcal{A}_\mathrm{S}, I_\mathrm{S}, \mathcal{D}_\mathrm{S}, \mathcal{C}_\mathrm{S}^c \rangle$ *is* well-formed *iff:*

- *all the elements in* $\mathcal{S}_\mathrm{S}, \mathcal{A}_\mathrm{S}, \mathcal{D}_\mathrm{S}$*, and* $\mathcal{C}_\mathrm{S}^c$ *are well-formed*
- *the relation* I_S *is well-formed*
- *the two sets* $\mathcal{T}_\mathrm{S}^c$ *and* $\mathcal{T}_\mathrm{S}^a$ *are disjoint, i.e.,* $\mathcal{T}_\mathrm{S}^c \cap \mathcal{T}_\mathrm{S}^a = \emptyset$

- *the names used in the attribute, subcomponent and association definitions are unique for a given source type* T_s*, i.e.,*

$$\nexists(\delta_1 = \langle n_1, \mathrm{T}_s, \ldots\rangle, \delta_2 = \langle n_2, \mathrm{T}_s, \ldots\rangle) \in (\mathcal{S}_\mathrm{S} \cup \mathcal{A}_\mathrm{S} \cup \mathcal{D}_\mathrm{S}) \textit{ s.t. } n_1 = n_2$$

- *there is no loop in the partonomy tree, i.e.,*

$$\begin{array}{l}\forall\ (\mathrm{T}, \mathrm{T}')\ \textit{with}\ (\mathrm{T} \sqsubseteq_\mathrm{S} \mathrm{T}') \vee (\mathrm{T}' \sqsubseteq_\mathrm{S} \mathrm{T}),\ \nexists(\mathrm{T}_1, \ldots, \mathrm{T}_n)\ \textit{s.t.,} \\ \quad \mathrm{T}_1 = \mathrm{T}, \mathrm{T}_n = \mathrm{T}'\ \textit{and} \\ \quad \forall \mathrm{T}_i,\ \mathrm{T}_{i+1}, \\ \qquad (\mathrm{T}_i \sqsubseteq_\mathrm{F} \mathrm{T}_{i+1}) \vee (\mathrm{T}_{i+1} \sqsubseteq_\mathrm{F} \mathrm{T}_i) \vee (\exists \sigma_i = \langle \ldots, \mathrm{T}_i, \mathrm{T}_{i+1}, \ldots\rangle \in \mathcal{S}_\mathrm{S})\end{array}$$

We now define the well-formedness of concepts in realisation views.

A mapping constraint is well-formed when its compatibility expression can be evaluated to a boolean by the type environment induced by its origin structure type.

Definition 30 *For a realisation view* $\mathrm{R} \in \mathcal{V}^r$*, a* mapping constraint $\langle \mathrm{T}_s, \mathrm{T}_r, e^c\rangle$ *in* $\mathcal{C}^m_\mathrm{R}$ *is* well-formed *iff*

$$\Gamma_{\mathrm{T}_s} \vdash_R e^c : \texttt{bool}.$$

For subitem or suboperation definitions, the target type must be different from the source type, and the cardinality greater than 0.

Definition 31 *For a realisation view* $\mathrm{R} \in \mathcal{V}^r$*, a* subitem or suboperation definition $\langle \mathrm{T}_s, \mathrm{T}_t, r\rangle \in (\mathcal{S}^i_\mathrm{R} \cup \mathcal{S}^o_\mathrm{R})$ *is* well-formed *iff:*

$$(\mathrm{T}_t \neq \mathrm{T}_s) \wedge (r > 0)$$

Also, the operation types cannot be successor of themselves.

Definition 32 *For a realisation view* $\mathrm{R} \in \mathcal{V}^r$*, a* successor definition $\langle \mathrm{T}_p, \mathrm{T}_s\rangle$ *is* well-formed *iff for* $\mathrm{T} \in \mathcal{T}^o_\mathrm{R}$*,*

$$\nexists(\mathrm{T}_1, \ldots, \mathrm{T}_n) \in \mathcal{T}^o_\mathrm{R}\ \textit{s.t.}\ \eta_1 = \langle \mathrm{T}, \mathrm{T}_1\rangle, \ldots, \eta_n = \langle \mathrm{T}_n, \mathrm{T}\rangle \in \mathcal{N}_\mathrm{R}.$$

Definition 33 *A* realisation view $\mathrm{R} = \langle \mathrm{S}, \mathcal{T}_\mathrm{R}, \mathcal{S}^i_\mathrm{R}, \mathcal{S}^o_\mathrm{R}, \mathcal{N}_\mathrm{R}, \mathcal{C}^m_\mathrm{R} \rangle$ *with* $\mathcal{T}_\mathrm{R} = \mathcal{T}^i_\mathrm{R} \cup \mathcal{T}^o_\mathrm{R} \cup \mathcal{T}^r_\mathrm{R}$ *is* well-formed *iff:*

- *all the elements in* $\mathcal{S}^i_\mathrm{R}, \mathcal{S}^o_\mathrm{R}$, $\mathcal{N}_\mathrm{R}$ *and* $\mathcal{C}^m_\mathrm{R}$ *are well-formed*
- $\mathcal{T}^i_\mathrm{R}$, $\mathcal{T}^o_\mathrm{R}$ *and* $\mathcal{T}^r_\mathrm{R}$ *are pairwise disjoint*
- *there is no loop in the partonomy trees, i.e.,*
  - $\forall\ (\mathrm{T}, \mathrm{T}') \in \mathcal{T}^i_\mathrm{R}$, $\nexists(\mathrm{T}_1, \ldots, \mathrm{T}_n)$ *s.t.,* $\mathrm{T}_1 = \mathrm{T}, \mathrm{T}_n = \mathrm{T}'$ *and* $\forall \mathrm{T}_i,\ \mathrm{T}_{i+1}, \exists \sigma_i = \langle \ldots, \mathrm{T}_i, \mathrm{T}_{i+1}, \ldots \rangle \in \mathcal{S}^i_\mathrm{R}$
  - $\forall\ (\mathrm{T}, \mathrm{T}') \in \mathcal{T}^o_\mathrm{R}$, $\nexists(\mathrm{T}_1, \ldots, \mathrm{T}_n)$ *s.t.,* $\mathrm{T}_1 = \mathrm{T}, \mathrm{T}_n = \mathrm{T}'$ *and* $\forall \mathrm{T}_i,\ \mathrm{T}_{i+1}, \exists \sigma_i = \langle \ldots, \mathrm{T}_i, \mathrm{T}_{i+1}, \ldots \rangle \in \mathcal{S}^o_\mathrm{R}$

Finally, before defining well-formed configuration models, we must make sure that the specialisation function defines no loop in the feature view hierarchy and that there exists one and only one base view in that hierarchy.

Definition 34 *For a configuration model* M*, the* specialisation function Sp_M *is* well-formed *iff:*

$$\forall \mathrm{F} \in \mathcal{V}^f, (\mathrm{F}, \mathrm{F}) \notin Sp^*_\mathrm{M} \wedge (base_\mathrm{M} \neq \emptyset) \wedge \forall (\mathrm{F}, \mathrm{F}') \in base_\mathrm{M}, \mathrm{F} = \mathrm{F}'$$

For the sake of readability, we shall then write in this case $base_\mathrm{M} = \mathrm{F}$ *instead of* $base_\mathrm{M} = \{\mathrm{F}\}$.

Definition 35 *A* ***configuration model*** $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_\mathrm{M} \rangle$ *is* **well-formed** *iff all the following statements hold:*

- *All feature views* $\mathrm{F} \in \mathcal{V}^f$ *are well-formed.*
- *All structure views* $\mathrm{S} \in \mathcal{V}^s$ *are well-formed.*
- *All realisation views* $\mathrm{R} \in \mathcal{V}^r$ *are well-formed.*
- *The specialisation function* Sp_M *is well-formed.*

These rules ensure that the a model declared in ProCoLa are well-formed, and must be satisfied before starting the configuration. In order to guarantee well-formedness of the model, relevant tool support must be provided to the knowledge engineers. We present in Chapter 10 our prototype implementation in which ProCoLa models are compiled. Most of the well-formedness rules are checked by the tool during this compilation process, with the support of a Cycle Detection Analysis, presented in the next section.

6.4 Analysing ProCoLa Models

In this section, we will present several analyses of the configuration model based on the formalism defined in Section 6.2. We first present how to construct the labelled graph used for the analyses, and then the analyses themselves.

6.4.1 Graph construction

The different analyses of the configuration model are based on graphs with labelled nodes and different kind of directed edges. We thus define a number of operations on model elements and labels for each type of views.

6.4.1.1 Labels

Feature views. Consider a feature view $\mathrm{F} \in \mathcal{V}^f$. We define a function

$$element2label_{\mathrm{F}}\colon (\mathcal{T}_{\mathrm{F}} \cup \mathcal{S}_{\mathrm{F}} \cup \mathcal{D}_{\mathrm{F}} \cup \mathcal{C}^c_{\mathrm{F}} \cup \mathcal{C}^i_{\mathrm{F}}) \to \mathbf{Label}$$

that associates a label to relevant elements in the feature view, i.e, feature types, attribute and subfeature definitions, and constraints.

We also define the inverse function

$$label2element_{\mathrm{F}} : \mathbf{Label} \to (\mathcal{T}_{\mathrm{F}} \cup \mathcal{S}_{\mathrm{F}} \cup \mathcal{D}_{\mathrm{F}} \cup \mathcal{C}^c_{\mathrm{F}} \cup \mathcal{C}^i_{\mathrm{F}})$$

that retrieves the element associated to a specific label.

Structure views. Consider a structure view $\mathrm{S} \in \mathcal{V}^s$. We define the label functions on structure types, resources, resource use, attribute, subcomponent and association definitions, and constraints:

$$\begin{aligned} element2label_{\mathrm{S}} &: (\mathcal{T}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{S}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}}) \to \mathbf{Label} \\ label2element_{\mathrm{S}} &: \mathbf{Label} \to (\mathcal{T}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{S}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}}) \end{aligned}$$

Realisation views. For a realisation view $\mathrm{R} \in \mathcal{V}^r$, we define the label functions on realisation types, subitem and suboperation definitions, successor definitions, and mapping constraints as

$$\begin{array}{l} \mathit{element2label}_{\mathrm{R}} : (\mathcal{T}_{\mathrm{R}} \cup \mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \Omega_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{S}}) \to \mathbf{Label} \\ \mathit{label2element}_{\mathrm{R}} : \mathbf{Label} \to (\mathcal{T}_{\mathrm{R}} \cup \mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \Omega_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{S}}) \end{array}$$

Configuration model. Consider now the whole configuration model $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_{\mathrm{M}} \rangle$. We define the following functions:

$$\begin{array}{lcl} \mathit{element2label} & = & (\bigoplus_{\mathrm{F} \in \mathcal{V}^f} \mathit{element2label}_{\mathrm{F}}) \oplus (\bigoplus_{\mathrm{S} \in \mathcal{V}^s} \mathit{element2label}_{\mathrm{S}}) \\ & & \oplus (\bigoplus_{\mathrm{R} \in \mathcal{V}^r} \mathit{element2label}_{\mathrm{R}}) \\ \mathit{label2element} & = & (\bigoplus_{\mathrm{F} \in \mathcal{V}^f} \mathit{label2element}_{\mathrm{F}}) \oplus (\bigoplus_{\mathrm{S} \in \mathcal{V}^s} \mathit{label2element}_{\mathrm{S}}) \\ & & \oplus (\bigoplus_{\mathrm{R} \in \mathcal{V}^r} \mathit{label2element}_{\mathrm{R}}) \end{array}$$

The labelling process must ensure that any element δ satisfies $\mathit{element2label}(\delta) = l$ and $\mathit{label2element}(l) = \delta$. For the sake of simplicity, we will adopt the notation $[\delta]^l$.

Note that we assume the *element2label* and *label2element* functions to be bijective. If we have $[\delta]^l$ and $[\delta]^{l'}$, then $l = l'$; moreover, if we have $[\delta_1]^l$ and $[\delta_2]^l$, then $\delta_1 = \delta_2$.

Then the set of labels occurring in the modelling views is given by the following functions:

$$\begin{array}{lcl} \mathit{labels}^f & : & \mathcal{V}^f \to \mathcal{P}(\mathbf{Label}) \\ \mathit{labels}^s & : & \mathcal{V}^s \to \mathcal{P}(\mathbf{Label}) \\ \mathit{labels}^r & : & \mathcal{V}^r \to \mathcal{P}(\mathbf{Label}) \end{array}$$

where

$$\begin{array}{lcl} \mathit{labels}^f(\mathrm{F}) & = & \{l \mid [\delta]^l \in (\mathcal{T}_{\mathrm{F}} \cup \mathcal{S}_{\mathrm{F}} \cup \mathcal{D}_{\mathrm{F}} \cup \mathcal{C}^c_{\mathrm{F}} \cup \mathcal{C}^i_{\mathrm{F}})\} \\ \mathit{labels}^s(\mathrm{S}) & = & \{l \mid [\delta]^l \in (\mathcal{T}_{\mathrm{S}} \cup \mathcal{S}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}})\} \\ \mathit{labels}^r(\mathrm{R}) & = & \{l \mid [\delta]^l \in (\mathcal{T}_{\mathrm{R}} \cup \mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \Omega_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{S}})\} \end{array}$$

We finally define the *labels* function by:

$$\mathit{labels}(M) = \mathit{labels}^f(\mathcal{V}^f) \cup \mathit{labels}^s(\mathcal{V}^s) \cup \mathit{labels}^r(\mathcal{V}^r)$$

Example 37 *Let us return to our case study, and the* Communication *type from the base feature view, written in ProCoLa (only two constraints are shown here).*

```
featureType Communication [
 attributes:
  internetAccess: boolean;  socialNetworking: boolean;
 subFeatures:
  phone[0..1]: Phone;  network: Network;
 constraints:
  { phone[1].3G -> internetAccess };
 implementation:
  {Count(phone) = 1} <-> {Present(Physical::RFCard) and
      Present(Software::PhoneApp)};
];
```

The labelled elements from the formalisation of the Communication feature types involved here are:

- *feature types:* [Communication]^{1}, [Phone]^{8} *and* [Network]^{10}.
- *attribute declarations:* [$\alpha_{\text{internetAccess}}$]^{2}, [$\alpha_{\text{socialNetworking}}$]^{3}, *originating from the Communication type, and* [α_{3G}]^{8} *from the Phone type.*
- *subfeature declarations:* [σ_{phone}]^{4} *and* [σ_{network}]^{5}.
- *the compatibility constraint* [c_1]^{6} *and the implementation constraint* [i_1]^{7}.
- *component types, from the software structure view* [PhoneApp]^{11} *and from the physical structure view* [RFCard]^{12}.

6.4.1.2 Edges

We will need to operate on different types of edges between labelled nodes in the graph. We thus define functions returning pairs of labels to define those edges, according to the different elements in the configuration model. Three different types of edges are created: dependency edges, use edges and taxonomy edges. When two nodes (l_1, l_2) are linked through *dependency edges*, it means that deleting the first one l_1 will make the second one l_2 inconsistent with respect to the semantic correctness of the model. *Use edges* are created when a labelled element references another element (except for taxonomy relations). Finally, *taxonomy edges* represent the taxonomic relations between labelled elements. The latter edges are separated from use edges to help identifying taxonomic cycles, as will be discussed in Section 6.4.2.2.

Feature views. For a view $\mathrm{F} = \langle \mathcal{T}_\mathrm{F}, \mathrm{T}_\mathrm{F}^{root}, \mathcal{S}_\mathrm{F}, I_\mathrm{F}, \mathcal{D}_\mathrm{F}, Ref_\mathrm{F}, \mathcal{C}_\mathrm{F}^c, \mathcal{C}_\mathrm{F}^i, \mathrm{M} \rangle$, we define the functions

$$\begin{array}{lcl} depEdges_\mathrm{F} & : & (\mathcal{S}_\mathrm{F} \cup I_\mathrm{F} \cup \mathcal{D}_\mathrm{F} \cup \mathcal{C}_\mathrm{F}^c \cup \mathcal{C}_\mathrm{F}^i) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges_\mathrm{F} & : & (\mathcal{S}_\mathrm{F} \cup \mathcal{D}_\mathrm{F} \cup \mathcal{C}_\mathrm{F}^c \cup \mathcal{C}_\mathrm{F}^i) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ taxoEdges_\mathrm{F} & : & I_\mathrm{F} \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{array}$$

These two functions respectively describe the dependency and the use-relation between labelled elements, and are defined for a feature view F:

- A subfeature definition $[\sigma]^l = \langle name, [\mathrm{T}_s]^{l_1}, [\mathrm{T}_t]^{l_2}, r_1, r_2 \rangle$ in $\mathcal{S}_\mathrm{F}$ relates two types $[\mathrm{T}_s]^{l_1}$ and $[\mathrm{T}_t]^{l_2}$. If one of these two types is deleted, the subfeature definition makes no sense any more, so we have:

  $$depEdges_\mathrm{F}([\sigma]^l) = \{(l_1, l), (l_2, l)\}$$

  Also, subfeature definitions are referenced by the source type $\mathrm{T}_s]^{l_1}$, and make use of the target type $[\mathrm{T}_t]^{l_2}$ through $[\sigma]^l$:

  $$useEdges_\mathrm{F}([\sigma]^l) = \{(l_1, l), (l, l_2)\}$$

- For a pair $([\mathrm{T}]^l, [\mathrm{T}']^{l'}) \in I_\mathrm{F}$ involved in a taxonomic relation, deleting the supertype $[\mathrm{T}]^l$ would invalidate the subtype $[\mathrm{T}']^{l'}$. Also, $taxoEdges_\mathrm{F}$ records that these two types are part of the same taxonomy:

  $$\begin{array}{lcl} depEdges_\mathrm{F}([\mathrm{T}]^l, [\mathrm{T}']^{l'}) & = & \{(l, l')\} \\ taxoEdges_\mathrm{F}([\mathrm{T}]^l, [\mathrm{T}']^{l'}) & = & \{(l, l'), (l', l)\} \end{array}$$

- An attribute definition $[\alpha]^l = \langle name, [\mathrm{T}_s]^{l'}, \mathrm{D}, v, r \rangle$ in $\mathcal{D}_\mathrm{F}$ is referenced by its source type $[\mathrm{T}_s]^{l'}$, and is dependent of it: removing it would make $[\alpha]^l$ semantically inconsistent.

  $$\begin{array}{l} depEdges_\mathrm{F}([\alpha]^l) = \{(l', l)\} \\ useEdges_\mathrm{F}([\alpha]^l) = \{(l', l)\} \end{array}$$

- Constraints are dependent on their context type, as well as each variable involved in their expressions For a compatibility constraint defined by $[c_c]^l = \langle [\mathrm{T}]^{l'}, e^c \rangle \in \mathcal{C}_\mathrm{F}^c$ and an implementation constraint in $\mathcal{C}_\mathrm{F}^i$ defined by $[c_i]^l = \langle [\mathrm{T}]^{l'}, e_\mathrm{C}^c, e_\mathrm{P}^i, \mathrm{Op} \rangle$:

  $$\begin{array}{lcl} depEdges_\mathrm{F}([c_c]^l) & = & \{(l', l)\} \cup \{(l_i, l) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \\ depEdges_\mathrm{F}([c_i]^l) & = & \{(l', l)\} \cup \{(l_i, l) | [v_i]^{l_i} \in \mathbf{Vars}(e_\mathrm{C}^c) \cup \mathbf{Vars}(e_\mathrm{C}^i)\} \end{array}$$

Moreover, each constraint is used by its context type, and references all the variables from compatibility expressions. Use edges are limited to one specific view, so implementation expressions are not referencing the own variables.

$$\begin{array}{lcl} useEdges_{\mathrm{F}}([c_c]^l) & = & \{(l', l)\} \cup \{(l, l_i) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \\ useEdges_{\mathrm{F}}([c_i]^l) & = & \{(l', l)\} \cup \{(l, l_i) | [v_i]^{l_i} \in \mathbf{Vars}(e^c_{\mathrm{C}})\} \end{array}$$

Then, the set of dependency and use edges in a feature view is given by:

$$\begin{array}{c} depEdges^f : \mathcal{V}^f \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges^f : \mathcal{V}^f \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ taxoEdges^f : \mathcal{V}^f \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{array}$$

where

$$\begin{array}{lcl} depEdges^f(\mathrm{F}) & = & \{depEdges_{\mathrm{F}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}_{\mathrm{F}} \cup I_{\mathrm{F}} \cup \mathcal{D}_{\mathrm{F}} \cup \mathcal{C}^c_{\mathrm{F}} \cup \mathcal{C}^i_{\mathrm{F}})\} \\ useEdges^f(\mathrm{F}) & = & \{useEdges_{\mathrm{F}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}_{\mathrm{F}} \cup \mathcal{D}_{\mathrm{F}} \cup \mathcal{C}^c_{\mathrm{F}} \cup \mathcal{C}^i_{\mathrm{F}})\} \\ taxoEdges^f(\mathrm{F}) & = & \{taxoEdges_{\mathrm{F}}([\delta]^l) \mid [\delta]^l \in I_{\mathrm{F}}\} \end{array}$$

Example 38 *In Example 37, the following edges are created:*

depEdges:$\{(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(2,6),(4,7),(8,4),(10,5),(9,6)\}$
useEdges:$\{(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(6,2),(7,4),(4,8),(5,10),(6,9)\}$

Structure views. We define the edges functions for a structure view $\mathrm{S} \in \mathcal{V}^s$:

$$\begin{array}{lcl} depEdges_{\mathrm{S}} & : & (\mathcal{S}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup I_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}}) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges_{\mathrm{S}} & : & (\mathcal{S}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}}) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ taxoEdges_{\mathrm{S}} & : & I_{\mathrm{S}} \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{array}$$

These functions are identical to the $depEdges_{\mathrm{F}}$, $useEdges_{\mathrm{F}}$, and $taxoEdges_{\mathrm{F}}$ for subcomponent and association definitions (similar to subfeature definitions), attribute definitions, compatibility constraints, the taxonomy relation and compatibility constraints.

The edges functions for a structure view S are defined further for a resource definition $[v]^l = \langle [\mathrm{T}]^{l_1}, [\mathrm{R}]^{l_2}, u, e^c \rangle$ in $\mathcal{S}_{\mathrm{S}}$ by:

$$\begin{array}{l} depEdges_{\mathrm{S}}([v]^l) = \{(l_1, l), (l_2, l)\} \cup \{(l_i, l) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \\ useEdges_{\mathrm{S}}([v]^l) = \{(l_1, l), (l, l_2)\} \cup \{(l, l_i) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \end{array}$$

A resource definition is dependent on the interacting component type $[\mathrm{T}]^{l_1}$ and the resource used $[\mathrm{R}]^{l_2}$, as well as all variables involved in the resource use expression e^c. Furthermore, the interacting type $[\mathrm{T}]^{l_1}$ makes use of the resource definition $[v]^l$, which references the resource $[\mathrm{R}]^{l_2}$ and the variables in e^c.

Then, the set of dependency and use edges in a structure view is given by:

$$\begin{aligned} depEdges^s &: \mathcal{V}^s \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges^s &: \mathcal{V}^s \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ taxoEdges^s &: \mathcal{V}^s \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{aligned}$$

where

$$\begin{array}{lcl} depEdges^s(\mathrm{S}) & = & \{depEdges_{\mathrm{S}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup I_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}})\} \\ useEdges^s(\mathrm{S}) & = & \{useEdges_{\mathrm{S}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{S}} \cup \mathcal{U}_{\mathrm{S}} \cup \mathcal{A}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{S}} \cup \mathcal{C}^c_{\mathrm{S}})\} \\ taxoEdges^s(\mathrm{S}) & = & \{taxoEdges_{\mathrm{S}}([\delta]^l) \mid [\delta]^l \in I_{\mathrm{S}}\} \end{array}$$

Example 39 *Consider the* Motherboard *type in the physical structure view of our case study, and the interactions with the* AvailableSlots *resource already considered in Example 29 (page 104).*

This is defined in ProCoLa with:

```
componentType Motherboard [
 attributes:
  nbSlots: integer{6,10,12,14};
  hidden nbChips: integer[0..5];
        ...
 produces: AvailableSlots := nbSlots - nbChips;
];
```

with the labels: $[\text{Motherboard}]^1$, $[\alpha_{\text{nbSlots}}]^2$, $[\alpha_{\text{nbChips}}]^3$, $[v_{\text{AS}}]^4$ *and* $[\text{AvailableSlots}]^5$. *The following depdendency and use edges are created:*

$$\begin{aligned} depEdges{:}&\{(1,2),(1,3),(1,4),(5,4),(2,4),(3,4)\} \\ useEdges{:}&\{(1,2),(1,3),(1,4),(4,5),(4,2),(4,3)\} \end{aligned}$$

The resource use $[v_{\text{AS}}]^4$ *is dependent on* $[\text{Motherboard}]^1$, $[\text{AvailableSlots}]^5$, *and the variables* $[\alpha_{\text{nbSlots}}]^2$ *and* $[\alpha_{\text{nbChips}}]^3$, *while it references the resource* $[\text{AvailableSlots}]^5$ *and the variables.*

Realisation views. We now define the edges functions for a realisation view $\mathrm{R} \in \mathcal{V}^r$:

$$\begin{aligned} depEdges_{\mathrm{R}}\colon\ & (\mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \Omega_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{R}}) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges_{\mathrm{R}}\colon\ & (\mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \cup\Omega_{\mathrm{R}}\mathcal{C}^m_{\mathrm{R}}) \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{aligned}$$

The edges functions for a realisation view R are defined by:

- Dependency and use edges for subitem and suboperation definitions are similar to the ones for subfeature definitions. For a subitem or suboperation definition $[\sigma]^l = \langle name, [\mathrm{T}_s]^{l_1}, [\mathrm{T}_t]^{l_2}, r\rangle \in \mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}}$:

$$\begin{aligned} depEdges_{\mathrm{S}}([\sigma]^l) &= \{(l_1, l), (l_2, l)\} \\ useEdges_{\mathrm{S}}([\sigma]^l) &= \{(l_1, l), (l, l_2)\} \end{aligned}$$

- A successor definition $[\eta]^l = \langle [\mathrm{T}_p]^{l_1}, [\mathrm{T}_s]^{l_2}\rangle \in \mathcal{N}_{\mathrm{R}}$ depends on both types involved, and these types reference the successor definition:

$$\begin{aligned} depEdges_{\mathrm{S}}([\eta]^l) &= \{(l_1, l), (l_2, l)\} \\ useEdges_{\mathrm{S}}([\eta]^l) &= \{(l_1, l), (l_2, l)\} \end{aligned}$$

- A resource usage definition $[\omega]^l = \langle name, [\mathrm{T}_o]^{l_1}, [\mathrm{T}_r]^{l_2}\rangle \in \Omega_{\mathrm{R}}$ also depends on both types involved. It is referenced by the defining operation type $[\mathrm{T}_o]^{l_1}$ and make use of the resource type $[\mathrm{T}_r]^{l_2}$:

$$\begin{aligned} depEdges_{\mathrm{S}}([\omega]^l) &= \{(l_1, l), (l_2, l)\} \\ useEdges_{\mathrm{S}}([\omega]^l) &= \{(l_1, l), (l, l_2)\} \end{aligned}$$

- Finally, a mapping constraint: $[m]^l = \langle [\mathrm{T}_s]^{l_1}, [\mathrm{T}_r]^{l_2}, e^c\rangle \in \mathcal{C}^m_{\mathrm{R}}$ depends on both its origin structure type and its defining realisation type, as well as all the variables involved in the mapping expression. Moreover, it is used by the structure type $[\mathrm{T}_s]^{l_1}$ and references both the realisation type $[\mathrm{T}_r]^{l_2}$ and the variables in e^c:

$$\begin{aligned} depEdges_{\mathrm{S}}([m]^l) &= \{(l_1, l), (l_2, l)\} \cup \{(l_i, l) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \\ useEdges_{\mathrm{S}}([m]^l) &= \{(l_1, l), (l, l_2)\} \cup \{(l, l_i) | [v_i]^{l_i} \in \mathbf{Vars}(e^c)\} \end{aligned}$$

Then, the set of dependency and use edges in a realisation view is given by:

$$\begin{aligned} depEdges^r &: \mathcal{V}^r \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \\ useEdges^r &: \mathcal{V}^r \to \mathcal{P}(\mathbf{Label} \times \mathbf{Label}) \end{aligned}$$

where

$$depEdges^r(\mathrm{R}) = \{depEdges_{\mathrm{R}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{R}})\}$$
$$useEdges^r(\mathrm{R}) = \{useEdges_{\mathrm{R}}([\delta]^l) \mid [\delta]^l \in (\mathcal{S}^i_{\mathrm{R}} \cup \mathcal{S}^o_{\mathrm{R}} \cup \mathcal{N}_{\mathrm{R}} \cup \mathcal{C}^m_{\mathrm{R}})\}$$

Example 40 *We have a look now at the mapping constraint defined in Example 34 on page 106, and declared in ProCoLa by:*

```
operationType Coating [
                mapping: TouchScreen: { oleophobicCoating };
                ...
];
```

with the labels: $[\text{Coating}]^1$, $[\alpha_{\text{oleophobicCoating}}]^2$, $[m_{\text{map}}]^3$ *and* $[\text{TouchScreen}]^4$. *The following depdendency and use edges are created:*

$$depEdges{:}\{(1,3),(2,3),(4,3)\}$$
$$useEdges{:}\{(1,3),(3,2),(3,4)\}$$

Configuration model. Consider now the whole configuration model $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_{\mathrm{M}} \rangle$. We define the following functions:

$$\begin{aligned}
depEdges(M) &= (\bigoplus_{\mathrm{F}\in\mathcal{V}^f} depEdges_{\mathrm{F}}) \oplus (\bigoplus_{\mathrm{S}\in\mathcal{V}^s} depEdges_{\mathrm{S}}) \\
&\quad \oplus (\bigoplus_{\mathrm{R}\in\mathcal{V}^r} depEdges_{\mathrm{R}}) \\
useEdges(M) &= (\bigoplus_{\mathrm{F}\in\mathcal{V}^f} useEdges_{\mathrm{F}}) \oplus (\bigoplus_{\mathrm{S}\in\mathcal{V}^s} useEdges_{\mathrm{S}}) \\
&\quad \oplus (\bigoplus_{\mathrm{R}\in\mathcal{V}^r} useEdges_{\mathrm{R}}) \\
taxoEdges(M) &= (\bigoplus_{\mathrm{F}\in\mathcal{V}^f} taxoEdges_{\mathrm{F}}) \oplus (\bigoplus_{\mathrm{S}\in\mathcal{V}^s} taxoEdges_{\mathrm{S}})
\end{aligned}$$

Thus $labels(\mathrm{M})$, $depEdges(\mathrm{M})$, $useEdges(\mathrm{M})$ and $taxoEdges(\mathrm{M})$ will be a representation of the dependency, use and taxonomy graphs for the configuration model M.

Example 41 *If we return to Example 37, Figure 6.1 shows the graph constructed from these labelled elements and edges. This graph shows that* $[\sigma_{\text{phone}}]^4$ *is dependent of* $[Communication]^1$ *and* $[Phone]^8$, *thus removing one of the two types would make the subfeature declaration inconsistent in the formal model. Also, we can see that* $[Phone]^8$ *is used by* $[\sigma_{\text{phone}}]^4$, *and by transitivity is also used by* $[Communication]^1$.

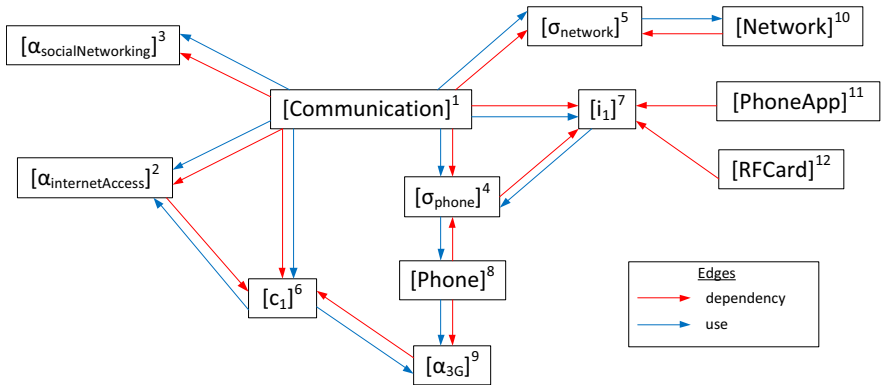


Figure 6.1: Graph for the *Communication* feature type

6.4.2 Analyses

We now present three analyses of the configuration model that rely on the graph defined previously. These analyses are based on two types of algorithms: a worklist algorithm [71] and an algorithm for finding strong components.

The Dependency Analysis permits to find other model elements dependent on a specific element, while the Usage Analysis can be used to determine which elements are used by a specific one. Finally, the Cycle Detection Analysis identifies the partonomic and taxonomic cycles in a model.

6.4.2.1 Dependency and Usage Analyses

In this section, we define the first two analyses based on the dependency and use edges of the constructed graph. We first present the objectives of each analysis, before describing the worklist algorithm used to compute them.

Dependency Analysis. The Dependency Analysis aims at finding the set of labelled elements that depends on an initial labelled element. It takes as input a set of labelled elements *init* and computes the list of labels $\{l_1, ..., l_n\}$ that will become inconsistent with respect to the model semantics and well-formedness if elements in *init* are removed from the model.

The edges used to compute the Dependency Analysis are the dependency edges, from the *depEdges* functions.

Example 42 *Recall Example 37, and consider the Phone feature type. Removing this type from the model has the following consequences on the model:*

- *The subfeature declaration* $[\sigma_{\text{phone}}]^4 = \langle \text{“}phone\text{”}, Communication, Phone, 0, 1\rangle$ *becomes invalid, as the target type is not a feature type anymore:* $Phone \notin \mathcal{T}_{\text{F}_{base}}$.
- *The attribute declaration* $[\alpha_{\text{3G}}]^9 = \langle \text{“}3G\text{”}, Phone, \{\texttt{true}, \texttt{false}\}, \text{VISIBLE}, 1\rangle$ *becomes also invalid, as its source type is not in* $\mathcal{T}_{\text{F}_{base}}$ *either.*
- *Finally, the compatibility constraint* $[c_1]^6$ *should be examined for deletion as well, as it refers to* α_{3G}*, which is not a valid attribute declaration anymore.*

The Dependency Analysis applied on $init = \{[Phone]^8\}$ *should thus return the set of labels* $\{4, 6, 9\}$.

Usage Analysis. The Usage Analysis aims at finding the set of labelled elements that are used in a model. This permits to identify, by looking at the original set of labels, the elements that are not used in the configuration model, and to take actions towards either cleaning the model or repairing it for example. The Usage Analysis takes as input a set of labelled elements *init* and computes the list of labels $\{l_1, ..., l_n\}$ that are used by the elements in *init*. The edges used to compute the Usage Analysis are the use and taxonomy edges, from the *useEdges* and *taxoEdges* functions. A complete Usage Analysis of an entire configuration model starts with the set of the root types of all feature and structure views.

One special case concerns variables from implementation expressions: due to the different semantics of these expressions (i.e., the implicit existential quantifier used to interpret them), these variables have a special relation with the implementation constraint, and thus are linked using implementation edges, and not use edges. Their usage must thus be determined from the view they are part of.

Example 43 *In Example 37,* $[Network]^{10}$ *is only used by the subfeature declaration* $[\sigma_{\text{network}}]^5$*. If* $[\sigma_{\text{network}}]^5$ *were not in the model anymore,* $[Network]^{10}$ *would not be in the Usage Analysis set computed on* $[Communication]^1$.

Note that $[\text{PhoneApp}]^{11}$ *and* $[\text{RFCard}]^{12}$ *are not reachable from Communication via the use edges, and thus are not in the Usage Analysis set computed from label 1, as they are part of the implementation of the Communication feature type. To be part of the full model Usage Analysis, these types must then be used in their own view.*

Worklist algorithm. The algorithm used to compute the two analyses introduced above is shown as Algorithm 1. It takes as argument the initial set of labels *init*, as well as the set of edges E to consider for the requested analysis. For a Dependency Analysis, we have $E = depEdges(\mathrm{M})$, while we have $E = (useEdges(\mathrm{M}) \cup taxoEdges(\mathrm{M}))$ for a Usage Analysis.

Algorithm 1: WORKLIST(*init*: Set, E: Set): Set

```
1  Analysis = ∅;
2  W = ∅;
3  foreach l ∈ init do
4  |  W = CONS(l, W);
5  |  Analysis = Analysis ∪ {l};
6  while W ≠ ∅ do
7  |  l = HEAD(W);
8  |  W = TAIL(W);
9  |  foreach (l, l') ∈ E do
10 |  |  if l' ∉ Analysis then
11 |  |  |  W = cons(l', W);
12 |  |  |  Analysis = Analysis ∪ {l'};
13 return Analysis;
```

The algorithm is based on a worklist W, which is initialised with labels from the *init* set, along with the result set, *Analysis*. The function CONS(l, W) is the list constructor, while HEAD(W) and TAIL(W) produce the head and tail of the worklist. In the *while* loop (line 6-12), the first element of the worklist is extracted and removed from W. Each of its successors l' in the set of edges E is put under examination and, if not already in *Analysis*, is added to the worklist and the result set. Line 13 finally produces the result of the algorithm.

6.4.2.2 Cycle Detection Analysis

The aim of the Cycle Detection Analysis is to detect partonomy and taxonomy cycles in a configuration model whose well-formedness with respect to cycles is not proven. This analysis is based on the different graphs created from the model, and detects the *strong components* of the graphs.

Definition 36 *Two nodes l and l' are said to be* strongly connected *whenever there is a (possibly trivial) directed path from l to l' and a (possibly trivial) directed path from l' to l.*

Defining

$$\mathcal{SC} = \{(l, l') | l \text{ and } l' \text{ are strongly connected}\}$$

we obtain an equivalence relation $\mathcal{SC} \subseteq labels(\mathrm{M}) \times labels(\mathrm{M})$ (the proof is well-known and omitted here).

The equivalence classes of $\mathcal{SC}$ are called the *strong components* (or *strongly connected components*) of the graph.

Each strong component with more than one node is a subgraph where cycles occur. Thus, detecting strong components permits to test for cycles and identify them if there are any. To find these strong components, we use a depth-first search algorithm based on Tarjan's [97] that takes a set of edges E as parameter (and the root label(s) of the graph *init*). The computation is actually split into the two Algorithms 2 and 3.

Algorithm 2: DFS(*init*: Set, E: Set): Set

```
1  foreach l ∈ init do
2      E = E ∪ {0, l};
3  L = ∅; SC = ∅;
4  visited = {0}; deleted = ∅;
5  counter = 0;
6  VisitSC(0);
7  foreach (l, l') ∈ E do
8      if l = 0 then
9          E = E \ {l, l'};
10 return SC;
```

The two algorithms use the $dfsNum$ and low data structures. The DFS numbering $dfsNum(l)$ of a node l represents the number of vertices visited before l in the depth-first search. In case a back or cross edge is discovered out of the subtree of l, it is recognised by the algorithm as it has been visited before and thus has a smaller $dfsNum$. In that case, $low(l)$ is used to record the low link of l, i.e., the smallest DFS numbering of a vertex reachable by a back or cross edge from the subtree of l. With this, any element where $low(l) = dsfNum(l)$ is the head of a (trivial) component.

Algorithm 2 creates a dummy root element for the graph with label 0 (line 1-2), useful when the set *init* contains more than one element, and initialises the

Algorithm 3: VISITSC(l: Label, E: Set)

```
1  L = CONS(l, L);
2  dfsNum(l) = counter;
3  low(l) = counter;
4  counter = counter + 1;
5  foreach (l, l') ∈ E do
6      if l' ∉ deleted then
7          if l' ∉ visited then
8              visited = visited ∪ {l'};
9              VISITSC(l');
10             low(l) = MIN(low(l), low(l'));;
11         else
12             low(l) = MIN(low(l), dfsNum(l'));;
13 if low(l) = dfsNum(l) and L ≠ ∅ then
14     create new strong component in SC;
15     repeat
16         v = LAST(L);
17         L = L \ {v};
18         deleted = deleted ∪ {v};
19         add v to the current strong component;
20     until v = l;
```

different data structures. The set *visited* records the visited nodes so they are not examined again, while *deleted* records the nodes found as part of a strong component and that are virtually deleted from the graph.

Algorithm 3 performs a depth-first search, starting from the dummy root. Each node is inserted into a stack L in the order they are visited, the $dfsNum$ and *low* are assigned and a *counter* is incremented. All successors (via the edges) of the node l are looked at, and visited if need be. When each node finishes recursing, if its low value is still set to its DFS numbering, then it is the root node of a strong component, formed by all of the nodes above it on the stack L. The stack is popped up to and including the current node, and records those nodes as being part of a new strong component.

Example 44 *Consider Example 37 with the additional subfeature declaration*

$$\sigma_{comm} \quad = \quad \langle \text{“}comm\text{”}, \text{Network}, \text{Communication}, 1, 1 \rangle$$

There is no taxonomic relation involved in this set of elements, so it is possible to find the partonomic cycles only via use edges. The modified graph is shown

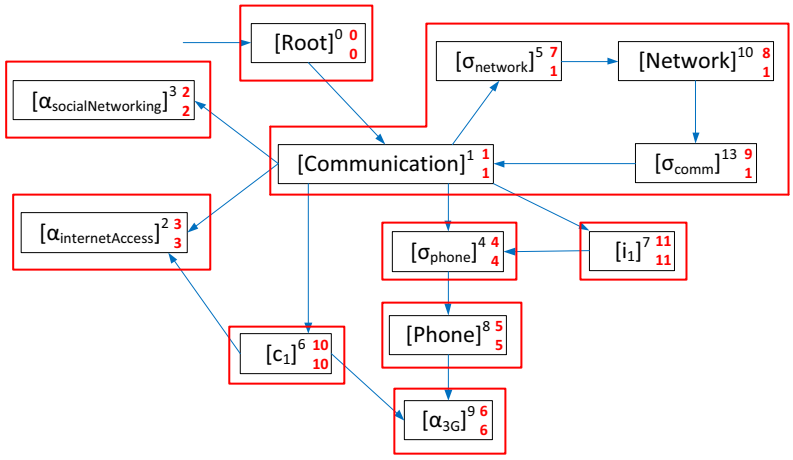


Figure 6.2: Strong components for the (modified) use graph from Figure 6.1. Red boxes indicate strong components. Red numbers for each labelled element indicates *dfsNum* value (top) and *low* value (bottom).

in Figure 6.2, with only use edges. The DFS *Algorithm has been applied, and each strong component is highlighted in a square box. Each labelled element is now assigned a DFS numbering (top number) and a low value (bottom value). The partonomy cycle introduced because of* σ_{comm} *is identified and is now part of a single strong component, headed by* $[Communication]^1$*. Each element of this strong component has been assigned the low value of 1, which is the DFS numbering of* $[Communication]^1$*.*

However, in the general case, elements involved in taxonomic relations form taxonomy clusters that must be taken into account when looking for partonomic cycles. Taxonomy edges in the graph are thus bi-directional, because instances of a specific type is also an instance of its supertypes, and may as well be specialised in one of its subtypes. Each of the types related together via taxonomy should thus be grouped together when analysing partonomic cycles.

The general Cycle Analysis is thus a three-step process:

1. First, strong components of the graph with use and taxonomic edges are identified. This isolates the partonomic cycles and/or the taxonomic clusters. Consider the model elements from Figure 6.3(a). The Figure 6.3(b) shows the use and taxonomic graph for these elements, which contains only one big strong component.

2. Each of the strong components found previously can be either a single taxonomy cluster (without any partonomic cycle) or several clusters linked together by partonomic relations. Thus, the DFS Algorithm is applied again on the taxonomic graph only within each of these strong components. This separates the different taxonomic clusters of each strong component. Figure 6.3(b) shows two taxonomy clusters in the previous example: this implies that a partonomic cycle does exist withing the strong component.

3. Finally, each taxonomy cluster is scanned for taxonomic cycles. The DFS Algorithm is applied on the dependency graph, as in Figure 6.3(d), which shows one taxonomic cycle.

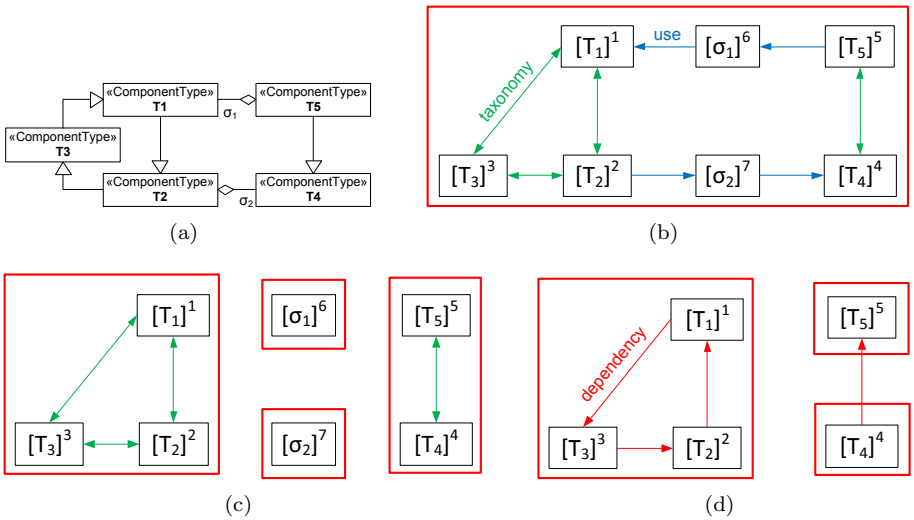


Figure 6.3: Cycle Analysis example. (a) Example model. (b) Strong components of the use and taxonomic graph. (c) Identification of the taxonomy clusters. (d) Taxonomic cycles.

6.5 Summary

In this chapter we defined ProCoLa, a modelling language for declaring the product knowledge of heterogeneous product families. We described the full syntax of ProCoLa for three types of modelling views, feature, structure and realisation views, as well as constraint expressions for compatibility constraints, implementation constraints and mapping constraints.

We also defined a formalism to represent configuration models for heterogeneous product families, in which ProCoLa has been translated. The formalism is used to:

- define a type system for constraint expressions, using a formal constraint language.
- define well-formedness rules for our modelling framework.

Such rules provide ProCoLa with clear semantics, and permit to ensure that product models in ProCoLa are created in accordance to the concepts defined in the previous chapter. The formalism has then been used to define and apply analyses on configuration models. These analyses are based on a graph derived from the formal models and allow to:

- Assess the impact of deleting model elements, using the Dependency Analysis. This analysis permits to discover what elements might be inconsistent with the model's semantics when maintenance is performed or if changes are introduced due to the model evolution.
- Detect unused elements in the models, using the Usage Analysis. Unused elements may be remains of previous evolutions of the configuration model, and may be superfluous.
- Detect partonomy and taxonomy cycles in the model. Such cycles can provoke issues when interpreting the model before configuration is started, and must be avoided.

These analyses provide insights on the correctness of the model, as well as potential issues that may arise when modifying it. Practical implementation of the analyses is discussed in Chapter 10.

Part III

Constraint Solving

Chapter 7

Constraint Satisfaction Problems and Configuration

Proposing an accessible modelling language to design and maintain product families is not sufficient: a configuration framework must not only be able to support the modeling, but also support efficient mechanisms to solve the configuration of product models at runtime. Because it is highly declarative and domain independent, the Constraint Satisfaction Problem (CSP) framework is often used for solving configuration problems. Modelling and solving Constraint Satisfaction Problems problems have been studied in depth.

In this part, we aim at defining concrete semantics for our configuration framework by translating it to Constraint Satisfaction Problems. The CSP framework needs to be extended in order to fulfill the requirements of product configuration. We thus identify relevant extensions for our framework, and introduce a novel algorithm for dealing with table constraints, before presenting how elements in ProCoLa and our configuration framework are mapped to the chosen CSP formalism.

We first provide in this chapter some necessary background on Constraint Satisfaction Problems relevant for our work, including algorithms to maintain consistency in the classic CSP and its extensions.

7.1 Classic CSP

7.1.1 Definitions

The definition of a Constraint Satisfaction Problem (CSP) can be found in [85]. A CSP is a triple $\mathcal{P} = \langle X, D, C \rangle$ where:

- X is an n-tuple of variables $X = \langle x_1, x_2, ..., x_n \rangle$,
- D is a corresponding n-tuple of *domains* $D = \langle D(x_1), D(x_2), ..., D(x_n) \rangle$, representing, for each variable x_i, the set of possible values it can take,
- C is a t-tuple of constraints $C = \langle c_1, c_2, ..., c_t \rangle$ restricting the values that the variables can simultaneously take. A constraint c_k is a pair $\langle scp(c_k), rel(c_k) \rangle$ where $rel(c_k)$ is a relation on the variables in $scp(c_k)$. $scp(c_k)$ is called the scope of c_k. A constraint c_k can be defined in *intension*, in which case the relation $rel(c_k)$ is a predicate built from variables in $scp(c_k)$, constant values and a set of functions (operators); c_k can also be defined in *extension*, with $rel(c_k)$ being the set of all tuples of allowed values. Constraints defined in extension are also called *table constraints*.

Example 45 *Figure 7.1 shows an example of a CSP, composed of five variables:* $X = \langle x_1, x_2, x_3, x_4, x_5 \rangle$ *and four constraints* $C = \langle c_1, c_2, c_3, c_4 \rangle$. *These constraints are shown in extension, i.e., as a set of allowed tuples for each combination of variables.*

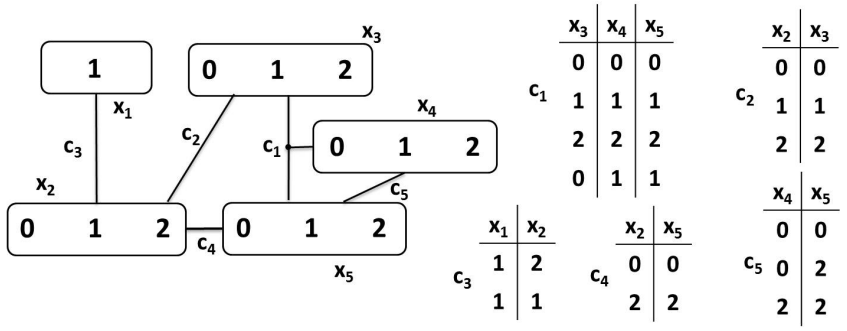


Figure 7.1: Example of a CSP. Each node represents a variable whose domain is explicitly indicated. The constraints are defined in extension to specify the set of allowed tuples.

Solving a CSP consists in assigning a value to each variable such that all constraints are satisfied. In more formal term, a *solution* to the CSP $\mathcal{P}$ is an n-tuple $A = \langle a_1, a_2, ..., a_n \rangle$ where $a_i \in D(x_i)$, $x_i = a_i$ and each c_k is satisfied in that $rel(c_k)$ holds on the projection of A onto the scope $scp(c_k)$ (i.e., for the variables only involved in $rel(c_k)$). If the set of solutions, $sol(\mathcal{P})$, is empty, then the CSP is said to be *unsatisfiable*.

Constraint satisfaction problems on finite domains are typically solved using a form of search. Search techniques are used to explore the solution space of the problem, and find solution to the CSP if it exists. The basic and very inefficient systematic search algorithm is Generate-and-Test, where each possible assignment to all variable is generated and tested as a solution of the CSP. A much more popular approach is chronological backtracking. In its simplest form, the algorithm constructs potential solutions by assigning value to variables in a depth-first manner. It extends in each step a partial solution towards a complete one, and backtracks when a constraint is violated, i.e., it retracts the most recent instantiated variable and tries another value.

7.1.2 Arc Consistency in CSP

Finding solutions for a CSP is an NP-complete task [85], so local consistency properties can be used to simplify the problem during search. Local consistency properties are properties defined on a subset of variables and constraints. One of the most common local consistency properties is Arc Consistency.

Definition 37 *For each constraint c on $(x_1, ..., x_i, ..., x_p)$, we define a tuple $\tau = (d_1, ..., d_i, ..., d_p)$ as a* support *of c if it is an allowed tuple ($\tau \in rel(c)$) and $\forall x_i \in scp(c), \tau(x_i) = d_i \in D(x_i)$. A pair (x_i, d_i) is said to be* arc consistent *if, for each $c \in C$ constraining x_i, there exists a support $\tau \in c$ such that $\tau(x_i) = d_i$. A problem $\mathcal{P}$ is* arc consistent *if $\forall x_i \in X, \forall d_i \in D(x_i), (x_i, d_i)$ is arc consistent. A problem $\mathcal{P}_k$ is* maximally arc consistent *if enlarging the current domain of any of its variables makes it not arc consistent.*

Example 46 *Figure 7.2 shows the CSP from Figure 7.1 where the domain of each variable has been pruned in order to enforce the Arc Consistency property.*

Many generic algorithms have been designed to maintain arc consistency in a CSP with binary constraints, for example AC3 [60], AC4 [68] and AC6 [16]. Still AC3, due to its simplicity, remains the main algorithm used and worked on (e.g., AC-3.3 in [59]). In such algorithms, pairs of variables (x_i, x_j) involved in a binary constraint c_{ij} are called *arcs*. These arcs are *revised* by the algorithms

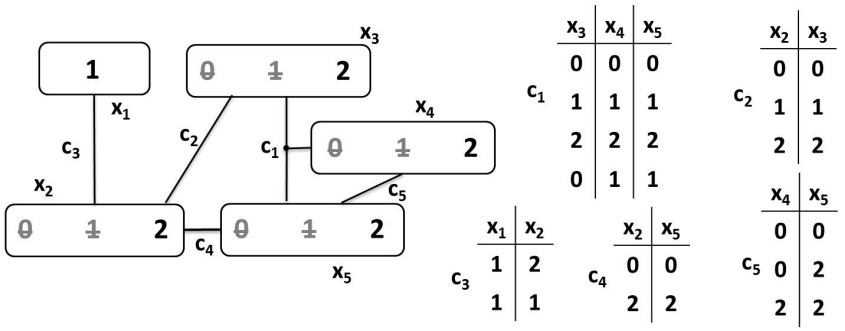


Figure 7.2: Arc Consistent CSP from Figure 7.1.

and the domain of the two variables are reduced in order to ensure the Arc Consistency property. Arcs are set to be revised when the domain of one of the variables involved is modified due to another constraint.

Algorithm 4 presents the AC3 algorithm. It uses a REVISE function (Algorithm 5) to revise arcs of the constraint graph. The main point in AC3 is that if the arc (x_i, x_j) is revised and the domain of x_i reduced, there is no need to re-revise the arc (x_i, x_j), because there is no deleted element from the domain of x_i that can have an influence on the elements from the current domain of x_j.

Consequently, these arcs are not revisited, which improves the overall performance of the algorithm.

Algorithm 4: Algorithm AC3($\langle X, D, C\rangle$: CSP): Bool

```
1 begin
2     Q ← {(x_i, c_ij) | x_i ∈ scp(c_ij), c_ij ∈ C};
3     while Q ≠ ∅ do
4         select and remove (x_i, c_ij) from Q;
5         if Revise(x_i, c_ij) then
6             if D(x_i) ≠ ∅ then return false;
7             else
8                 Q ← Q ∪ {(x_k, c_ki) | k ≠ i, k ≠ j};
9     return true;
```

Algorithm 5: Function REVISE(x_i: variable,c_{ij}: constraint): Bool

```
1 begin
2     CHANGE ← false;
3     foreach v_i ∈ D(x_i) do
4         if there is no v_j ∈ D(x_j) such that (x_i,x_j) is consistent with c_ij then
5             remove x_i from D(x_i);
6             CHANGE ← true;
7     return CHANGE;
```

Consistency algorithms reduce the domains of the variables during the solving, while keeping the problem consistent with the constraints. Using them during search greatly improves performance, as it can prune the search tree considerably. The Forward Checking algorithm [85] enforces Arc Consistency between the currently assigned variables and unassigned variables. Other techniques like Full Look Ahead [85] or MAC [81] (Maintaining Arc Consistency) go up to enforcing full Arc Consistency on the CSP.

7.2 Extensions to CSP

Problems in product configuration can be solved using Constraint Satisfaction Problems. However, configuration exhibits several issues that cannot be solved using the classic CSP approach. We present in this section several extensions to the classic CSP approach.

7.2.1 Conditional CSP

A particular need in configuration is the fact that some elements of the configuration model can be optional, or the number of parts of a subcomponent can vary. For example, the motherboard in our case study can contain one or several wireless chips, and the number of chips may be decided at configuration time, depending on the assignments of some attributes in other components. Other issues include the use of variables with hierarchical structure, or the grouping of variables. Mittal and Falkenhainer [67] define Dynamic CSPs, renamed *Conditional CSPs (CondCSPs)* later to avoid ambiguity with another representation presented in the next Section. Some variables in a CondCSP are initially active

while others are not, and constraints can be classified in two categories: *activity constraints*, that activate or deactivate variables, or *compatibility constraints* (similar to the constraints in the classic CSP). In this early formulation, only algorithms based on backtracking techniques had been presented, so several other proposals extended the original work: a new revision by Soininen and Gelle [92], where activity constraints are generalized; the Composite CSP model from [86], which models a hierarchical structure between variables and constraints, using metavariables as placeholders for subproblems; the CSPe representation [106], where a state attribute is associated to each variable, giving the possibility to represent the activity of the variable. More advanced algorithms to specify and solve CondCSP have also been developed in [43, 87] or even further by Geller and Veksler [44] with their Activity CSP formulation. This latest formulation simply associates to each variable an activity variable, which can be either true or false, and that will be integrated into the constraints to handle their activity. Defining activity variables as "classic" variables has the advantage that classic solving method can be used.

Although well-studied, these Conditional CSPs can only deal with variables known from the start and thus problems where variables are generated during solving cannot be solved with those methods. This can occur in some configuration models where the set of the components used in the final solution is not known beforehand, e.g., in models with partonomy relations containing an arbitrary number of subcomponents. Other approaches have been studied to solve that problem. Stumptner et al. [95] describe Generative CSPs (GCSPs), where constraints with metavariables can be used to express generic relations. Mailharro [61] defines another framework, capable of satisfying on-demand generation of components in configuration. His approach is based on constrained set variables, which can contain a special value (wildcard) that represents the set of all components that have not been instantiated yet. One difference between the two approaches is instead of constrained set variables, the Generative CSP by Stumptner et al. [95] uses arrays of ports that can be quantified over. Mailharro's approach provides a more symmetry-breaking-friendly syntax, while the approach from Stumptner et al. is more suitable for representing complex layout constraints (e.g.,, situations where individual component placement is important, like placing cards in a rack).

7.2.2 Dynamic CSP

Real world systems are often dynamic, involving changes in the implemented models. This is especially true for interactive configuration, where selecting or deselecting a value for a specific variable can be seen as the addition or removal of a user requirement. CSPs have therefore been extended as *Dynamic Constraint Satisfaction Problems (DCSP)* to handle such dynamic systems.

Definition 38 *A Dynamic CSP is a sequence of CSPs* $\mathcal{P}_0, \mathcal{P}_1, ..., \mathcal{P}_m$ *where each problem results from the addition or removal of a constraint* c *in its predecessor. More precisely, if* $\mathcal{P}_i = \langle X, D, C_i \rangle$*, then* $\mathcal{P}_{i+1} = \langle X, D, C_{i+1} \rangle$ *where* $C_{i+1} = C_i \pm c$*.*

Example 47 *A DCSP can be derived from the CSP in Figure 7.1:* $\mathcal{P}_0$ *starts with no constraints, and each* $\mathcal{P}_i$ *is created by adding the constraint* c_i*. Finally,* $\mathcal{P}_5$ *is created by remove the constraint* c_3*.*

Maintaining arc consistency in Dynamic CSPs is equivalent to making the problem $\mathcal{P}_i$ maximally arc consistent, provided that $\mathcal{P}_0, ..., \mathcal{P}_{i-1}$ are maximally arc consistent. Handling constraint *addition* is fairly straightforward: the new problem can indeed be made maximally arc consistent just by applying AC algorithms again.

However, the task of *removing* a constraint is a bit different. In fact, when removing a constraint, the problem stays arc consistent, but not necessary *maximally* arc consistent. This is due to the fact that some solutions can be lost, and formerly inconsistent values may need to be restored to re-establish maximal arc consistency.

Existing algorithms for arc consistency in DCSPs, e.g., DnAC4 [14] (based on AC4) are working in three phases during constraint removal: first the values originally deleted by the removed constraints are restored (initialization phase); then those restorations are propagated to other variables connected through existing constraints (propagation phase); finally, the restored values are filtered again (filtering phase).

The last phase ensures that all wrongly restored values are removed. The latest and most efficient algorithms for binary constraints are AC|DC-2i and AC3.1|DC-2i [11].

7.3 CSP with Non-Binary Constraints

Non-binary constraints are constraints involving more than two variables. Most authors have generally not extended their work to non-binary problems and *Generalised* Arc Consistency. The main reason is that any non-binary CSP can potentially always be transformed into an equivalent binary CSP, using techniques such as dual transformation [32] or hidden (variable) transformation [84]. However, this is sometimes impracticable [9]. In this Section, we present a few algorithms for non-binary problems.

7.3.1 Generalised Arc Consistency

Among existing algorithms enforcing (Generalised) Arc Consistency on non-binary CSPs are GAC3 [60] and GAC4 [69]. These algorithms are extensions of their binary counterparts, but are much more expensive. Indeed, they inspect all pairs of variables and values, and check their consistency for each constraint, resulting in an exponential worst-case time complexity.

Enforcing (Generalised) Arc Consistency on CSPs with non-binary constraints is NP-complete [17], and the best worst-time complexity obtained by generic algorithms such as GAC4 is $O(erd^r)$, with e the number of constraints, r the greatest constraint arity and d the greatest domain size.

Many other algorithms have been designed to enforce arc consistency in a more efficient way for specific problems, for example non-binary CSPs with table constraints. Tables can actually be considered as the most explicit way to represent constraints, as in theory any constraints in intension can be transformed into a table constraint.

However, this may in practice lead to time and space explosion when solving the problem. Another classic global constraint is the $\mathit{all\_different}(x_1, ..., x_n)$ constraint, which imposes that variables in the set $x_1, ..., x_n$ have different values. Algorithms such as the one from Régin [80] are capable of enforcing Arc Consistency on a problem with *all_different* constraints in polynomial time.

When dealing with Dynamic CSP, only one algorithm has been extended. The algorithm DnGAC4 [15] is an extension of DnAC4, and thus has the same complexity and performance of GAC4. The algorithm follows the three-phase approach common to DCSP algorithm for binary problems, and deals with non-binary constraints expressed in extension.

7.3.2 Simple Tabular Reduction

Table constraints are particularly relevant for configuration. Among the work on enforcing Arc Consistency for table constraints, Simple Tabular Reduction (STR) [58, 103] differs from previous methods as it virtually modifies the tables in order only to go through the supporting tuples when enforcing arc consistency. More precisely, when the domain of a variable is reduced, each table constraining that variable is updated, by removing all invalid tuples.

The STR algorithm is shown as Algorithm 6. Each constraint c is represented by a set of tuples $c.table$. These tuples are part of one of two linked lists: the *current* list, that contains the currently valid tuples (and consequently the current supports), and the *removed* list, containing the tuples that have been virtually removed from the table, because they are no longer valid.

Algorithm 6: GACSTR(c: Constraint, *depth*: Integer): Bool

```
1  S_sup = ∅;
2  foreach x ∈ scp(c) do
3  |  gacValues[x] = ∅;
4  |  add x to S_sup ;
5  prev = −1;
6  curr = c.first;
7  while curr ≠ −1 do
8  |  τ = c.table[curr];
9  |  nextT = c.next[curr];
10 |  rTail = c.restoredTail;
11 |  if ISVALID(c, τ) then
12 |  |  foreach x ∈ S_sup do
13 |  |  |  if (τ[x] ∉ gacValues[x]) then
14 |  |  |  |  add τ[x] to gacValues[x];
15 |  |  |  |  if |gacValues[x]| = |D(x)| then
16 |  |  |  |  |  remove x from S_sup;
17 |  |  prev = curr ;
18 |  else
19 |  |  REMOVETUPLE(c, prev, curr, depth);
20 |  curr = nextT;
21 foreach x ∈ S_sup do
22 |  if gacValues[x] = ∅ then
23 |  |  return false;
24 |  foreach d_x ∈ D(x) do
25 |  |  if d_x ∉ gacValues[x] then
26 |  |  |  remove d_x from values[x];
27 |  |  |  add x to changed ;
28 return true;
```

Algorithm 7: isValid(c: Constraint, τ: Tuple): Bool

```
1 foreach x ∈ scp(c) do
2     if τ[x] ∉ D(x)) then
3         return false;
4 return true;
```

Algorithm 8: removeTuple(c: Constraint, $prev, curr, depth$: Integers)

```
1 if prev = −1) then
2     c.first = c.next[curr];
3 else
4     c.next[prev] = c.next[curr];
5 c.next[curr] = c.removedHead[depth];
6 if c.removedHead[depth] = −1) then
7     c.removedTail[depth] = curr;
8 c.removedHead[depth] = curr;
```

Algorithm 9: restoreTuples(c: Constraint, $depth$: Integer)

```
1 if c.removedHead[depth] ≠ −1) then
2     c.next[c.removedTail[depth]] = c.first;
3     c.first = c.removedHead[depth];
4     c.removedHead[depth] = −1;
```

The algorithm uses the following data structures:

- The position $c.first$ of the first current tuple in $c.table$ ($c.first = -1$ if the current table is empty).
- Two arrays $removedHead$ and $removedTail$ that give the position of the first and last invalid tuples when the search is at a certain depth.
- An array $c.next$ that, for each tuple, links to the following tuple in the linked list it belongs to (current or removed list). $c.next[i] = -1$ if i is the last tuple in its list.
- A set $gacValues[x_i]$ for each variable x_i containing all values from $D(x_i)$ that have found a support in the table constraint c.

- A set S_{sup} of variables whose domain contains at least one value for which a support has not yet been found. This is an optimisation of the initial STR algorithm presented in [58]: it avoids unnecessary validity checks by iterating over variables in S_{sup} and removing from S_{sup} the variables whose whole domain has been proven consistent.

The algorithm goes over all current tuples (lines 7-20) and updates the *gacValues* sets accordingly. If a tuple is proven invalid, it is "removed" from the table and values are not added to *gacValues* (line 14). When the table has been scanned, the *gacValues* set are compared to the original domain of the variables involved, and variables with a modified domain are added to the propagation queue *changed* (lines 21-27). The head and tails of the removed list are recorded for each depth, allowing a chronological backtracking in constant time (without re-traversing the table) using the function RESTORETUPLES (Algorithm 9).

STR uses arrays of indexes to avoid moving tuples in memory, and is thus enforcing arc consistency in $O(er(d + t'))$, where t' is the maximum size of the current tables. Worst-case space complexity of STR is $O(e(n + rt))$, with t the greatest original table size. STR has been shown to be one of the most efficient approaches when dealing with table constraints.

Example 48 *Let's go back to Figure 7.1. Applying STR on constraint c_3 will give*

$$gacValues[x_1] = \{1\} \text{ and } gacValues[x_2] = \{1, 2\}$$

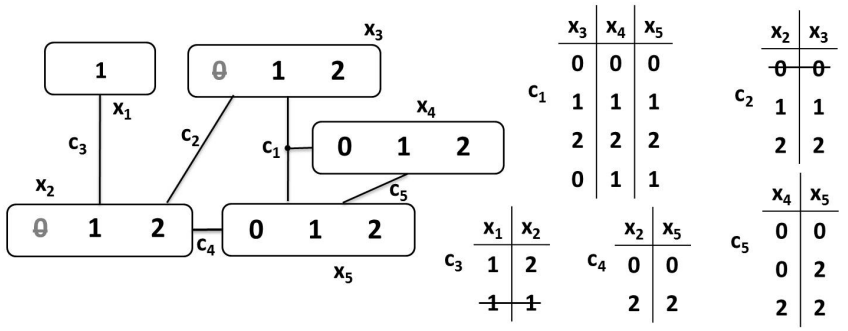
removing the value 0 in $D(x_2)$. The change then propagates to c_2: $0 \notin D(x_2)$, so the first tuple becomes invalid and is put in the removed list.

We then have:

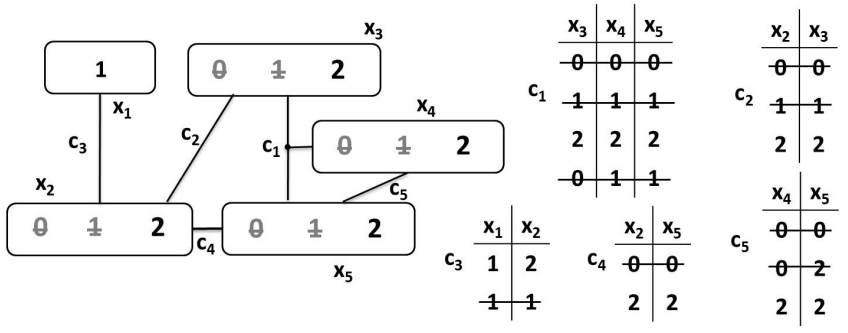
$c2.first = 2$, $c2.removedHead[0] = 0$, $c2.removedTail[0] = 0$
$c2.next[1] = -1$, $c2.next[2] = 3$ *and* $c2.next[3] = -1$

It follows:

$$gacValues[x_2] = \{1, 2\} \text{ and } gacValues[x_3] = \{1, 2\}$$



(a)



(b)

Figure 7.3: Applying STR on the CSP from Figure 7.1. (a) shows the state after first traversal of c_3 and c_2, and (b) after enforcing Arc Consistency

This eliminates the value 0 in $D(x_3)$. Figure 7.3(a) shows the CSP after these two propagations. Once all domain changes have been propagated and all tables virtually reduced, the CSP reaches Arc Consistency (Figure 7.3(b)).

Chapter 8

Non-binary Dynamic CSP with Simple Tabular Reduction

In this chapter, we focus on solving the Dynamic CSPs with non-binary constraints. Non-binary constraints defined in extension (i.e., table constraints) are commonly used in applications where databases are involved, in particular product configuration, as they can be used to represent product catalogues (see Section 2.3). Moreover, Dynamic CSPs provide a very interesting framework for solving configuration problems, as it permits to handle the addition and removal of constraints, whether they are user requirements or constraints designed by the product modeller.

As shown in the previous chapter, only the DnGAC4 algorithm [15] has been designed to enforce Arc Consistency on non-binary constraints. This algorithm is based on a relatively old Arc Consistency algorithm (GAC4). We propose in this chapter a new algorithm for solving Dynamic CSPs with extensional non-binary constraints, based on Simple Tabular Reduction (see Section 7.3.2), one of the most efficient approaches for maintaining Arc Consistency on table constraints.

8.1 The DnSTR Algorithm

In this Section, we introduce a new algorithm called *DnSTR* for solving DCSP with non-binary table constraints. It relies on a timeline of events, which is composed by timestamps. Each timestamp t is defined by two integers (dp, lt), which represent the *depth* of the timestamp (the depth is increased every time a constraint is added) and the *local time* at a given depth. The timestamps can also be compared, either based on different depth, or on different local time:

$$\forall t_1, t_2 : t_1 < t_2 \Leftrightarrow (t_1.dp < t_2.dp \vee (t_1.dp = t_2.dp \wedge t_1.lt < t_2.lt))$$

Like the other algorithms targeting DCSPs, DnSTR is working in three phases (as described in Section 7.2.2). However, the filtering phase in DnSTR is performed incrementally during constraint retraction, in order to keep the timeline consistent and use it to restore only specific tuples. Indeed, during constraint retraction, wrongly restored tuples need to be put back at the depth they would have been deleted at if the removed constraint had never been added to the problem. Thanks to this mechanism, when a value is restored at a certain depth, only tuples at this same depth have to be restored during propagation (except if the constraint associated to those tuples was added later, in which case the depth considered is this constraint addition's depth).

The algorithm works with the following data structures:

- A global depth counter *gdepth*, as well as an array *lastLTime* that for each depth *dp* associates the last local time *lt* in the current timeline.
- Two justifications arrays $justif_t$ and $justif_c$ that, for each pair (x, d_x), keep track of when and by which constraint the value d_x has been removed from $D(x)$. Each constraint is also associated with the time of addition, recorded in the array $time_c$.
- As we already pointed out, the algorithm DnSTR is based on Simple Tabular Reduction. It thus shares similar data structures for each table constraint, including an initial set of tuples *c.table*, the index of the first tuple in the current table *c.first* (i.e., the table of valid tuples), an array *c.next* that links lists of tuples (by their indexes), and two arrays *c.removedHead* and *c.removedTail* that give the index of the first and last invalid tuples at each depth. Note that $c.next[i] = -1$ if i is the position of the last tuple, either in the current table if i is a valid tuple or in the removed list if i has been removed. For each constraint c, we also store the index of the last restored tuple in *c.restoredTail*, and an array $c.removed_t$ that records the removal time (or timestamp) of each tuple.

Algorithm 10: AddConstraint(c: Constraint): Bool

```
1 add c to C ;
2 gdepth = gdepth + 1 ; lastLTime[gdepth] = 0 ;
3 time_c[c] = (gdepth, 0) ;
4 revise[gdepth] = {c} ;
5 return Filter(revise, D, (gdepth, 0)) ;
```

Algorithm 11: Filter($revise, values$: Arrays, (dp_i, lt_i): Time): Bool

```
1  while revise[dp_i] ≠ ∅ do
2      select and remove c from revise[dp_i];
3      changed = ∅;
4      if not GACStrDyn(c, changed, (dp_i, lt_i), values) then
5          return false;
6      foreach (x, jt_x) ∈ changed do
7          c_revise = {c_r ∈ C s.t. c_r ≠ c, x ∈ scp(c_r), time_c[c_r] ≤ (dp_i, lt_i)};
8          if jt_x.dp > dp_i then
9              add c_revise to revise[jt_x.dp];
10         else
11             add c_revise to revise[dp_i];
12 return true;
```

8.1.1 Constraint Addition

Adding a constraint to the DCSP is done by filtering the system through the function AddConstraint described in Algorithm 10.

The *revise* structure contains for each depth the constraints to evaluate at filtering. Constraint addition works in the following way:

1. Filter iterates over the constraints to revise at the given depth (the latest for constraint addition) and applies a modified version of STR (Algorithm 12), where only the elements in the list *values* are taken into account (*values* represents the full domain D when adding constraints, but only the restored values are re-evaluated during the filtering phase of constraint retraction).
2. GACStrDyn differs from Algorithm 6 on page 149 as it handles the restored tuples (for constraint retraction) and the update of the justifi-

Algorithm 12: GACSTRDYN(c: Constraint, *changed*: Set, (dp_i, lt_i): Time, *values*: Array): Bool

```
1  S_sup = ∅;
2  foreach x ∈ scp(c) do
3      gacValues[x] = ∅;
4      if values[x] ≠ NIL then
5          add x to S_sup ;
6  prev = −1;
7  curr = c.first;
8  while curr ≠ −1 do
9      τ = c.table[curr];
10     nextT = c.next[curr]; rTail = c.restoredTail;
11     if ISVALID(c, τ, (dp_i, lt_i)) then
12         foreach x ∈ S_sup do
13             if (τ[x] ∉ gacValues[x] and τ[x] ∈ values[x]) then
14                 add τ[x] to gacValues[x];
15                 if |gacValues[x]| = |values[x]| then
16                     remove x from S_sup;
17         prev = curr ;
18     else
19         c.removed_t[curr] = (dp_i, lt_i) ;
20         if c.restoredTail = curr then c.restoredTail = prev;
21         REMOVETUPLE(c, prev, curr, dp_i);
22     if curr = rTail then break;
23     curr = nextT;
24 foreach x ∈ S_sup do
25     foreach d_x ∈ values[x] do
26         if d_x ∉ gacValues[x] then
27             remove d_x from values[x];
28             if |D[x]| ≠ |values[x]| then remove d_x from D[x];
29             if not (justif_c[x, d_x] = c and justif_t[x, d_x] > (dp_i, lt_i)) then
30                 justif_c[x, d_x] = c;
31                 justif_t = (dp_i, lt_i);
32             add (x, justif_t[x, d_x]) to changed ;
33     if |D[x]| = 0 then return false;
34 lastLTime[dp_i] = lt_i + 1 ;
35 return true;
```

Algorithm 13: isValid(c: Constraint, τ: Tuple, (dp_i, lt_i): Time): Bool

```
1 foreach x ∈ scp(c) do
2     if (τ[x] ∉ D[x]) and (justif_t[x, τ[x]] ≤ (dp_i, lt_i)) then
3         return false;
4 return true;
```

cations. Those are computed at lines 30-31, except if the pair (x, d_x) had been removed by the same constraint at a later stage (line 29) — in which case another support for (x, d_x) is still valid at this time but will be removed later in time.

3. IsValid also differs as it checks the value's justification time, while RemoveTuple is identical to the one described in Algorithm 8.

4. Filter then updates *revise* with potentially affected constraints (lines 6-11). The timestamp parameter (dp_i, lt_i) is used to specify when the filtering occurs (line 7), so that the constraints added *after* that time are not filtered yet (but will be later, when incremental filtering is used during constraint retraction). If x was changed at a later depth (lines 8-9), the constraint to revise is inserted at that specific depth $jt_x.dp$ instead of the current one.

8.1.2 Constraint Removal

Constraint retraction is performed by the function RemoveConstraint (Algorithm 14). During the initialization stage, all pairs (x, d_x) that were removed by the retracted constraints are restored. Notice that all restored values are placed in a list *restoredValues* so that only those values are considered in the filtering stage.

During the propagation stage in function Propagate, tuples from constraints connected to restored variables are restored.
Thanks to the strictly maintained timeline, only the tuples that have been removed after a pair (x, d_x) and *in the same depth* need to be restored, except in the case where the connected constraint has been *added after* the pair's deletion. In the latter case, the tuples deleted immediately at the constraint's addition are restored as well. This makes sure that we take into account the tuples potentially deleted because of (x, d_x), even though their deletion may have happened at a later depth than the pair's removal time. The function UpdateLists (Algorithm 15) is responsible for the updates of *restore* and *revise*.

Algorithm 14: RemoveConstraint(c: Constraint): Bool

```
// Initialization phase
restoredValues = ∅ ;
foreach x ∈ scp(c) do
    foreach d_x ∈ D_0[x] \ D[x] do
        if justif_c[x, d_x] = c then
            add d_x to D[x] ;
            add d_x to restoredValues[x] ;
Remove c from C ;
// Propagation phase
revise = ∅ ;
tcr = Propagate(restoredValues, revise, time_c[c].depth);
// Filtering phase
for dp_i ← time_c[c].dp to gdepth do
    Filter(revise, restoredValues, (dp_i, lastLTime[dp_i]));
// Cleaning phase
foreach c_j ∈ tcr do
    c_j.restoredTail = −1 ;
foreach x ∈ restoredValues do
    foreach d_x ∈ restoredValues[x] do
        justif_t[x, d_x] = NIL ;
        justif_c[x, d_x] = NIL ;
    restoredValues[x] = NIL ;
return true;
```

Algorithm 15: UpdateLists(x: Var, $restored\_x$: Set, dp_{rem}: Int, $revise$, $restore$: Arrays)

```
// Update of restore and revise
foreach c_x ∈ C constraining x do
    foreach depth dp_i do
        jtime[dp_i] = {min(jt = justif_t[x, d_x])|d_x ∈ restored_x & j.dp = dp_i)};
    add {max(jt, time_c[c_x]), jt ∈ jtime} to restore[c_x] ;
    if (∃d_x ∈ restored_x s.t. justif_t[x, d_x].dp ≤ time_c[c_x].dp) then
        add c_x to revise[max(time_c[c_x].dp, dp_rem)] ;
```

Algorithm 16: PROPAGATE(*restoredValues*: Set, *revise*: Array , dp_{rem}: Int): Set

```
1  restore = ∅ ;
2  foreach x ∈ restoredValues do
3      UpdateLists(x, restoredValues[x], revise, restore, dp_rem)
4  while restore ≠ ∅ do
5      select and remove (c, minRestoredTimes) from restore;
6      newRestored = ∅; add c to tcr ;
7      foreach (dp, lt) ∈ minRestoredTimes do
8          curr = c.removedHead[dp] ; prev = −1 ;
9          rtail = c.removedTail[dp] ;
10         if c.restoredTail = −1 then c.restoredTail = rtail ;
11         RestoreTuples(c, dp) ;
12         while curr ≠ −1 do
13             nextT = c.next[curr];
14             if c.removed_t[curr] < (dp, lt) then
15                 if c.restoredTail = curr then
16                     c.restoredTail = prev;
17                 removeTuple(c, prev, curr, dp);
18             else
19                 τ = c.table[curr]; jTimes = ∅ ;
20                 foreach x ∈ scp(c) do
21                     if τ[x] ∉ D[x] then
22                         if justif_c[x, τ[x]] ≠ c then
23                             add justif_t[x, τ[x]] to jTimes ;
24                         else
25                             add τ[x] to D[x] ;
26                             add τ[x] to newRestored[x] ;
27                 if jTimes ≠ ∅ then
28                     add c to revise[min(jTimes)] ;
29                 c.removed_t[curr] = NIL ; prev = curr ;
30             if curr = rTail then break;
31             curr = nextT;
32     foreach (x ∈ newRestored) do
33         UpdateLists(x, newRestored[x], dp_rem, revise, restore) ;
34         add newRestored[x] to restoredValues[x] ;
35 return tcr;
```

We store in *restore* for each constraint the minimum timestamp at each depth from which tuples must be restored (line 2-3). Line 5 ensures that if a constraint has been added after the pair's deletion, the constraint addition time is put into *restore* instead. In this special case, the constraint is put into *revise*, in order to be re-filtered after propagation (line 5-6 in UpdateLists).
The function Propagate (Algorithm 16) is in charge of the actual propagation mechanism. UpdatesList is called at the beginning of the propagation and after each restoration. The restoration times from *restore* are stored in *minRestoredTimes* (line 5). The function RestoreTuples(c, t) is identical to the one described in Algorithm 9. After tuples have been restored for a specific depth (line 11), we immediately remove again the ones that have been deleted before the restoration time (dp, lt) considered (lines 14-17). This avoids unnecessary re-filtering and sets a new removal time for these tuples. Otherwise, for each restored tuple, values that had been deleted by the current constraint are restored. In case another constraint has invalidated a tuple, the values are not restored, and the table constraint is added to *revise* for re-filtering at the time when a value first invalidated the tuple (line 27-28). Note that the Propagate algorithm returns the set *tcr* of table constraints that have been modified during restoration.

Finally, filtration is performed on the constraints in *revise* (line 12-13 in RemoveConstraint). This is done incrementally so that constraints are evaluated in the same order as they were added, keeping the new problem's timeline consistent with what it would be if the retracted constraint had never been added in the first place. Although it may introduce redundancy in the constraint evaluation process, this type of filtering is necessary to guarantee that all invalid tuples restored in the propagation stage are put back at the correct depth.

Example 49 *An example of constraint removal can be seen in Figure 8.1, after adding all constraints in the CSP from Figure 7.1. The first subfigure shows the CSP after the addition of the different constraints (c_1 at $t = (1, 0)$, c_2 at $t = (2, 0)$, ...). Each removed value from any domain in associated to justifications: the constraint responsible for removal (from $justif_c$) and the time of deletion (from $justif_t$). The time of (virtual) deletion of a tuple is also recorded using $c.removed_t$.*

The second subfigure starts the initialisation phase after the removal of the constraint c_3. In this phase, the variable-value pairs whose deletion has been caused directly by c_3 are restored.

In the propagation phase (third subfigure), c_2 and c_4 are first to be re-evaluated, because of the restoration of $(x_2, 0)$. Its deletion time is $t = (3.0)$, so only the first tuple of c_2 is restored (i.e., the one at the same depth $dp = 3$).

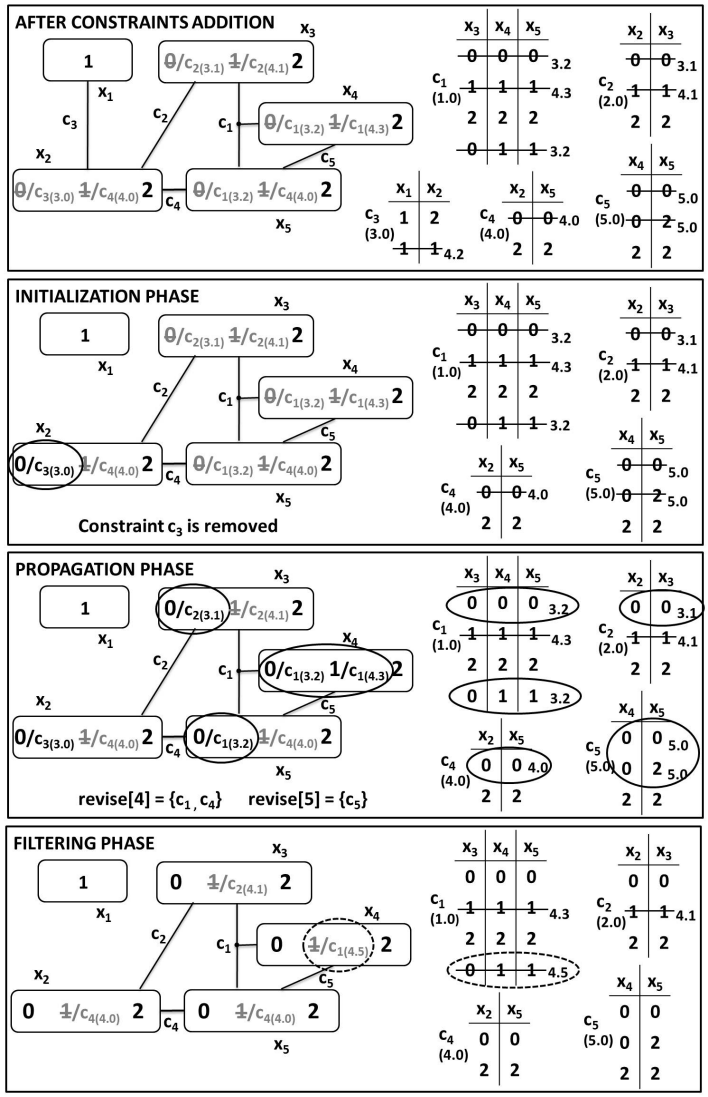


Figure 8.1: Removing constraint c3 in the CSP example from Figure 1. Associated timestamps are written as "3.1" for $(dp = 3, lt = 1)$ next to each added constraint and removed tuple. Justifications (constraint and time) are also shown next to each deleted value. Plain circles show the values and tuples restored since the previous phase, while dashed circles represent values and tuples re-filtered.

On the other hand, c_4 being added after the deletion of $(x_2, 0)$, the tuples deleted at the constraint addition are restored (i.e., its first tuple), and it is added to revise at depth 4. Because of the restoration of those tuples, $(x_3, 0)$ is restored. However, $(x_5, 0)$ cannot be restored by c_4 as it has been deleted by c_1.
More propagations follow: although the last tuple of table c_1 is restored, it is potentially invalid, as $(x_5, 1)$ has been removed by c_4. $(x_5, 1)$ is thus not restored, and the constraint c_1 is put in revise for the filtering phase. Finally, c_5 is also added to revise at depth 5, as the restored $(x_4, 1)$ had also been deleted before the addition of the tables.

The last phase (fourth subfigure) filters (only) the restored tuples from the constraints in revise*, at the corresponding depths 4 and 5, in order to keep the timeline consistent for future constraint removals. In this phase, the last tuple of c_1 is removed again, as $(x_5, 1)$ has not been restored during propagation, and therefore $(x_4, 1)$ is removed again.*

8.2 Analysis

In this section, we will discuss the correctness of DnSTR, and describe the time and space complexity of the algorithm.

8.2.1 Correctness

During constraint addition, the filtering depth is the global depth *gdepth*, and all values are checked when using the STR algorithm. As the second check in isValid is always true ($justif_t[x, \tau[x]]$ is equal to *NIL* for constraint addition), the STR algorithm used is equivalent to the original algorithm. All constraints to revise are put back in $revise[gdepth]$ in Filter as the check of line 8 is always false with $dp_i = gdepth$, so the correctness directly comes from the correctness of STR.

For constraint removal, let us consider a problem $\mathcal{P}_n$, resulting of the addition of $(c_0, ..., c_n)$, and where we are removing constraint c_i. Before the removal of c_i, problem $\mathcal{P}_n$ is arc consistent, so for all $\mathcal{P}_j$ with $j < n$, all justifications arrays are coherent with the actual timeline, all tuples have been removed at the correct depth and always at a local time greater than any justification of deleted value that might have caused its removal.

First, consider that c_i was added at $t_i = (d_i, 0)$, and $t_i + t_0$ is the first time that a value from a variable $x \notin scp(c_i)$ has been removed. All removals of values before $t_i + t_0$ are caused by c_i, and so those values are directly restored during the initialization phase, with c_i as justification constraint. For each constraint, the tuples impacted by a value removed at t_r are those deleted at a later time $t > t_r$.

Moreover, the only impacted tuples are the ones deleted in the same depth, as $\mathcal{P}_i$ was made consistent before any new constraint addition (and change of depth), *except* if they are from a constraint c_j added *after* c_i, as they may have been removed because of values deleted in $[t_i, t_i + t_0[$ at $d_j > d_i$. All those tuples are considered for restoration in UpdateLists (line 2-4) and restored in Propagate (line 11). All values supported by those restored tuples of any table c that were removed by c are restored in Propagate (line 25-26). The propagation of these restorations (via the *restore* list) ensures that a superset of all impacted values is restored.

We need now to make sure that all restored tuples that are removed during the filtering phase are put back at the correct depth and time. The incremental filtering is performed chronologically, so constraints just need to be added to *revise* at the correct depth. The local time is always increased during filtering using STR, so any removed tuple will always be placed after the first value responsible for its deletion.

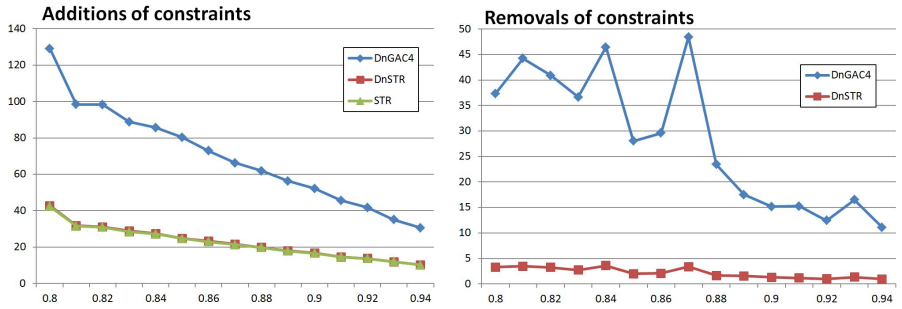
There are two cases when table constraints must be revised. The first one occurs when invalid tuples are restored. The tuples must then be placed at the earliest time when one of its value were deleted from its domain. This is taken care of in Propagate (line 28). In the second case, a value restored during propagation may actually have to be deleted by another constraint that had been added later (line 5-6 in UpdateLists).

Finally, justifications need to be maintained throughout the filtering process. When a restored value (x, d_x) is filtered by a constraint c_m at t_1, its justification constraint is obviously c_m. However, if d_x had been removed (prior to restoration) because of c_m but at a later time $t_2 > t_1$, it means that there is still another supporting tuple for (x, d_x) valid at t_1 (that will be removed only at t_2), and thus the justification time should stay equal to t_2 (GACStrDyn line 29).

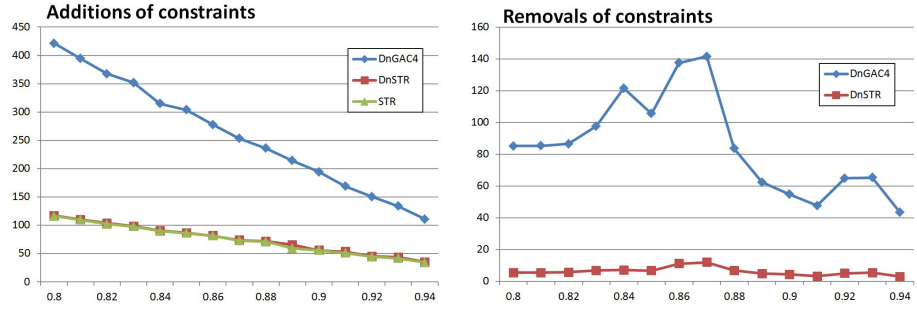
The RemoveConstraint(c) function thus correctly remove the constraint c and restore the variable domains as if the constraint had not been added, which ensures the correctness of our algorithm DnSTR.

8.2.2 Time and Space Complexity

The worst-case time complexity of STR is $O(er(d+t'))$, where r is the greatest constraint arity and d the greatest domain size (the number of constraints e has been added from Section 7.3.2 because the whole constraint system is considered). In the case of DnSTR, the worst-case time complexity of the initialization and propagation phases is the same as STR, where t' represents the size of the removed tuples list. However, the incremental filtering in the filtering phase may introduce a factor e, as the filtering could occur in the worst case as many times as there are constraints (or depths). The worst-case time complexity for DnSTR is thus $O(e^2r(d+t'))$. The space complexity of STR is $O(e(n+rt))$, with t the total size of the table and n the number of constraints added. For DnSTR, justifications arrays are in $O(nd)$. For each constraint, as for STR, the space complexity of *table* is in $O(rt)$ and that of *next* is in $O(t)$. The arrays *removedHead* and *removedTail* have size $O(e)$, as they can be filled for each depth, and $removed_t$ $O(t)$, which makes a worst-case space complexity of $O(e+rt)$ for each constraint, and $O(nd+e(e+rt))$ for DnSTR.



(a) $r = 8$



(b) $r = 9$

Figure 8.2: Runtime (in ms) as a function of p_2

Table 8.1: Memory consumption for RCSP $\langle 8, 20, 3, 100, p_2 \rangle$(in MB)

p_2	0.82	0.84	0.86	0.88	0.9	0.92	0.94
DnSTR4	8.3	8.2	7.6	7.4	7.3	6.7	5.1
STR	3.2	3	2.7	2.5	2.5	1.7	1.1
DnGAC4	12.5	12.3	10.4	8	7.1	6.9	5.7

8.3 Experimental Results

In order to evaluate the practical behaviour of the proposed algorithm, we have implemented DnSTR in a C# solver and compared it to DnGAC4 and the classic STR (where backtracking is used to remove constraints). The experiments were conducted using Random CSPs with model B [110]. Each instance is characterized by a tuple $\langle r, n, d, m, p_2 \rangle$ where r is the arity of each constraint, n the number of variables, d the uniform domain size, m the number of constraints and p_2 the tightness of the constraints. The tightness defines the number $t = (1-p_2)d^k$ of allowed tuples in each constraint. For each arity $r \in \{8, 9\}$, we did the experiments with 10 instances of RCSP $\langle r, 20, 3, 100, p_2 \rangle$, where $p_2 \in [0.8 - 0.94]$. The constraints were first added one by one (inconsistent systems were discarded). In order to see some real change in the variables' domain with such a low density (a more suitable density for a problem with arity 8 requires many more constraints), 10% random unary constraints were then added. Then 10% of all the constraints were removed. This protocol was applied 10 times per instance. The mean values of the runtime for adding and removing constraints are shown in Figures 8.2(a) and 8.2(b).

The experiments show that when adding a constraint, STR and DnSTR are almost identical and faster than DnGAC4. This is not surprising, as the STR algorithm, also used in DnSTR, has been shown to be more efficient than GAC4 [58]. For constraint removal, DnSTR outperforms both STR and DnGAC4. The results for the STR algorithm are not shown, as it performs more than 20 times worse than DnGAC4. The poor performance of STR is explained by the significant amount of work it has to perform after backtracking all the way to the removed constraint. DnGAC4 performs better than the classic STR, as it only checks restored values and updates justification arrays. However, DnSTR beats DnGAC4 again, thanks to the interaction with the dynamic tables of valid tuples. Note that the overhead of the incremental filtering necessary for preserving the timeline does not seem to have an important impact on performance.

We also measured the memory consumption of DnSTR, STR and DnGAC4 with RCSP $\langle 8, 20, 3, 100, p_2 \rangle$. In order to have significant measures, we added

all the constraints and assigned values to random variables until the problem stopped being consistent. Table 8.1 shows the mean memory consumption of the algorithms for different values of $p2$, just before the system became inconsistent. As we can see, the memory consumption between DnGAC4 and DnSTR is close, although DnSTR performs a bit better, while STR uses much less memory, as it has much less data structures to maintain.

Chapter 9

CSP Formalisation of the Configuration Framework

We now describe how the Configuration Modelling Framework and ProCoLa, detailed in Chapters 5 and 6 are converted into a CSP for the model's constraints to be solved at configuration time. We first explain our choice of CSP solving formalism and technology, then present the framework translation, and finally discuss the result of this Chapter.

9.1 CSP Formalism Chosen

The semantics behind our modelling approach (and ProCoLa) have been defined by implementing the translation of models to the Conditional CSPs (CondCSPs) formalism, using explicit activity variables with boolean domains. These activity variables are used in the CondCSP constraints to regulate the activity of the different model instances, depending on the different model elements involved, e.g., partonomic or taxonomy relations. In some cases, special variables need to be added to the model instance, e.g., to manipulate the number of subelement instances involved in a partonomic relation with a dynamic cardinality. The CondCSP constraints representing user-defined model constraints such as compatibility or implementation constraints involve also activity variables in order

to be consistent with the configuration model's semantics, especially when dealing with the existential quantifiers implied by the implementation constraints. A single CSP model is usually used for all views, allowing the propagation of any domain change to all views. However, it is also possible to isolate a single view (by removing inter-view dependencies such as implementation and mapping constraints).

Performance in interactive configuration is very important, so we chose to use Dynamic CSP (DCSP) mechanisms to take advantage of the dynamic addition and removal of CSP constraints at runtime. We combine two of the most recent algorithms for solving DCSPs: the DnSTR algorithm (see Chapter 8) to handle table constraints and an adaptation of the AC|DC-2i algorithm [11] for the other constraints. Those two algorithms use the same data structures to compute justifications when restoring values, and thus can be easily associated.

9.2 CSP Semantics of the Model

Configuration models created using our configuration framework need to be translated into the CondCSP formalism for the different constraints to be solved. At configuration time, an instance of a well-formed configuration model $\mathrm{M} = \langle \mathcal{V}^f, \mathcal{V}^s, \mathcal{V}^r, Sp_{\mathrm{M}} \rangle$ is created from the different views as follows:

- A feature view is chosen by the user from the views available in the feature views hierarchy. A tree of feature instances is created from feature types, starting from the root of the view. If this view is a refined view, the elements from the parent views have to be taken into account in order to create the instance. More details are given in Section 9.2.1.

- A view instance is created from each structure view in $\mathcal{V}^s$. As for the feature view, structure types are instantiated into a tree starting from the root component type of each structure view.

- Types from each realisation view are instantiated and mapped to instances of structural types from their associated structure view.

As explained before, type or attribute instances can be active or inactive, according to the choices of the user and the different constraints of the model: for example, mapping constraints may specify activity for specific instances of realisation types.

Once the configuration model has been instantiated, the corresponding Conditional Constraint Satisfaction Problem $\mathcal{P} = \langle X, D, C \rangle$ is created. After detailing the instantiation of feature views, we describe in the following sections how the different elements in the configuration framework are related to the CSP variables and constraints.

9.2.1 Feature View Instantiation

A feature view F_n that has been chosen by the end-user for configuration may be the refinement of one or several parent views, up until the base feature view. More specifically, there exists a sequence of feature views F_i such that

$$\forall i \in [0, n], \mathrm{F}_{i+1} = Sp_{\mathrm{M}}(\mathrm{F}_{i+1})$$

with $\mathrm{F}_0 = base_{\mathrm{M}}$ and $n \geq 0$ and Sp_{M} the specialisation function between feature views of the model M.

Each intermediary view thus needs to be taken into account in order to construct the instance of F_n. Whenever a view F_i declares a new root feature type (i.e., $\mathrm{T}^{root}_{\mathrm{F}_i} \in \mathcal{T}_{\mathrm{F}_i}$), a new tree of instances must be created from this root type. This tree becomes the *current feature tree*, i.e., the tree on which configuration will be done, until another view F_j with $j > i + 1$ declares a new feature tree. Each of the intermediate feature trees needs to be part of the CSP model just like the current feature tree, as they are all related using implementation constraints, and make the link from F_n to the base feature view F_0 (and the structure views).

When a feature type T is instantiated, its refinements need to be incorporated into its instances: each instance t of T takes into account the latest "version" of each attribute and subfeature declarations, i.e., with $\mathcal{D}_{\mathrm{F}_i}$ the set of attribute definitions of the feature view F_i, all $\alpha^* \in \bigcup_{i \in [0,n]} \mathcal{D}_{\mathrm{F}_i}$ for which

$$\alpha^* = \langle n, \mathrm{T}, \mathrm{D}, v, r \rangle \text{ and } \nexists \alpha, k \in [i, n] \text{ s.t. } \alpha \prec_{\mathrm{F}_k} \alpha^*$$

and, with $\mathcal{S}_{\mathrm{F}_i}$ the set of subfeature definitions of F_i, all $\sigma^* \in \bigcup_{i \in [0,n]} \mathcal{S}_{\mathrm{F}_i}$ for which

$$\sigma^* = \langle n, \mathrm{T}, \mathrm{T}_t, r_1, r_2 \rangle \text{ and } \nexists \sigma, k \in [i, n] \text{ s.t. } \sigma \prec_{\mathrm{F}_k} \sigma^*$$

as well as all the constraints $c \in \bigcup_{i \in [0,n]} (\mathcal{C}^c_{\mathrm{F}_i} \cup \mathcal{C}^i_{\mathrm{F}_i})$ with T as context type.

Note that properties from supertypes and subtypes also need to be taken into account, although we omit them here for sake of brevity. Taxonomic relations are treated in Section 9.2.4.

Example 50 *Consider the feature view* F_{ref} *described in Figure 5.10 on page 80 . This view is a direct refinement of the base feature view* F_{base}*, and defines no new root feature type. An instantiation of this feature view would then create a tree from the root type of the base feature view.*
However, types in F_{ref} *refine properties of the model or add constraints, e.g., the attribute declaration*

$$\alpha^*_{type} = \langle \text{“}type\text{”}, \text{Assistance}, \{\text{“}classic\text{”}, \text{“}pro\text{”}\}, \textsc{Visible}, 1\rangle$$

in $\mathcal{D}_{\mathrm{F}_{ref}}$ *is considered for instances of the Assistance type instead of*

$$\alpha_{type} = \langle \text{“}type\text{”}, \text{Assistance}, \{\text{“}classic\text{”}, \text{“}extended\text{”}, \text{“}pro\text{”}\}, \textsc{Visible}, 1\rangle$$

defined in $\mathcal{D}_{\mathrm{F}_{base}}$*.*

On the other hand, the feature view $\mathrm{F}_{packages}$ *declares a new root type: an instance tree is creating for the base feature view* F_{base}*, another for* $\mathrm{F}_{packages}$*, and both are related via implementation constraints.*

9.2.2 Type Instances and Attributes

Once the model has been instantiated, it is converted to a CondCSP model. As explained before, a CondCSP model contains two types of variables: “classic” variables and activity variables. In the coming sections, we detail how these variables are created and what constraints are derived from an instance of configuration model.

Although a model instance is tree-like thanks to partonomic and taxonomic relations between the different types instances in feature and structure views, the constraint model is flat: this means that variables are not related besides the constraints they are involved with.

For each instance t of a type T, an activity variable A_t is created. This variable represents whether or not the instance should be present in the final configuration, and thus has a boolean domain $D(A_t) = \{\texttt{true}, \texttt{false}\}$.

An instance t of type T also defines variables for each attribute of its possible types. For an attribute definition $\alpha = \langle \text{“}a\text{”}, \mathrm{T}, \mathrm{D}_a, v, r \rangle$, a variable x_a is created with domain D_a. An activity variable A_{x_a} is also associated to each attribute.

The variables of type instances and attributes may be involved in constraints, whether explicit (compatibility, implementation, mapping constraints, resource uses) or implicit (e.g., as part of partonomic or taxonomic relations). While these variables may not be active at all time during the configuration, most constraints that contain an inactive variable should be evaluated to `true`. Therefore, each constraint c is preceded by an activity guard A_c, except stated otherwise. For a constraint c on variables $(x_1, ..., x_p)$, the constraint actually added to the CondCSP is

$$A_{x_1} \wedge ... \wedge A_{x_p} \Rightarrow c$$

For the sake of readability, activity guards will be omitted in the rest of the chapter whenever possible.

9.2.3 Partonomic Relations

Partonomic relations relate instances with each other, whether it is as subfeature, subcomponent, subitem, or suboperation.

In case of a direct partonomic relation, i.e., with a static multiplicity of 1, the activity of an instance is directly equal to the activity of its subelements.

Example 51 *Consider the subcomponent definition*

$$\sigma_{support} = \langle \text{“}support\text{”}, \mathrm{DeviceServices}, \mathrm{Support}, 1, 1 \rangle$$

defined in the service structure view of our case study (Figure 5.6(c)), and an instance deviceServices of DeviceServices*. We then have:*

$$A_{deviceServices} = A_{support}$$

However, in case the partonomic relation $\sigma = \langle \text{“}s\text{”}, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2 \rangle$ has a dynamic multiplicity, a variable $count_s$ with domain $[r_1, r_2]$ is created. This variable is related to a generated model attribute that the end-user may modify to specify

the multiplicity of the partonomic relation. It entails that an activity variable A_{count_s} is also created.

For each instance t_i (with $i \in [r_1, r_2]$) created from σ, its activity is constrained by $count_s \geq i \Leftrightarrow A_{t_i}$, i.e., the activity of the instances acting as subelements are dependent on the value of $count_s$.

As stated in Section 9.2.2, this constraint is preceded by an activity guard (omitted here for brevity), and is directly evaluated to true if the instance t of T_s is not active (and thus $A_{count_s} = \texttt{false}$). The variable $count_s$ can also be used in the product model's constraints, when the built-in `Count`(...) function is used.

Example 52 *Consider the subcomponent definition*

$$\sigma_{wC} = \langle \text{“}wirelessChip\text{”}, \text{Motherboard}, \text{WirelessChip}, 0, 3 \rangle$$

defined in the physical structure view of our case study (Figure 5.6(a) on page 69), and an instance m of Motherboard*. We then have:*

$$A_{motherboard} = A_{count_{wC}}$$

and

$$A_{count_{wC}} \Rightarrow (count_{wC} \geq i \Leftrightarrow A_{wC_i}) \text{ for } i \in [0, 3]$$

with wC_1, wC_2, wC_3 being the three instances of the WirelessChip *component type.*

9.2.4 Taxonomic Relations

An instance t of a feature or structure type T contains the attributes, the subparts, the associations, and the constraints of its type and all its supertypes when involved in taxonomic relations. It *may* also contain the properties of a subtype T_{sub}, for example if the instance t is specialised as T_{sub} during configuration.

A variable $type_t$ is thus created for each instance t of T. This variable specifies the actual type of the instance: its domain is the possible types that the instance

can take, i.e., its original type or one of its subtypes. As for the $count_a$ variable, the variable $type_t$ is related to a generated model attribute that may be modified by the end-user during configuration, and is associated to an activity variable A_{type_t}.

A variable representing a property (attribute, subelement, association) in a subtype becomes active if t becomes an instance of that (sub)type. This means that, if $\mathcal{ST}(\mathrm{T}')$ represents the set of subtypes of any type T' in $D(type_t)$, and δ a property of T', we have

$$type_t \in (\mathcal{ST}(\{\mathrm{T}'\} \cup \{\mathrm{T}'\}) \Leftrightarrow A_{x_\delta}$$

Example 53 *Consider an instance scr of the* Screen *component type in the physical structure view of the Mobile Device product family. We have*

$$(\mathrm{TouchScreen}, \mathrm{Screen}) \in I_{\mathrm{S}_{phys}}$$

which means that TouchScreen *is a (direct) subtype of* Screen*. The domain of the* $type_{scr}$ *variable is* $\{\mathrm{Screen}, \mathrm{TouchScreen}\}$*, as* Screen *does not have other subtypes. The variable for the attribute* $\alpha_{technology}$ *defined in the* Screen *type is always active, while the one for* $\alpha_{oleophobicCoating}$ *defined in the* TouchScreen *type is only active if the scr instance is specialised as a* TouchScreen*:*

$$type_{scr} \in \{\mathrm{Screen}, \mathrm{TouchScreen}\} \Leftrightarrow A_{x_{technology}}$$
$$type_{scr} \in \{\mathrm{TouchScreen}\} \Leftrightarrow A_{x_{oleophobicCoating}}$$

9.2.5 Associations

Associations connect instances of component types and of association types in structure views. In a component instance t of type T, an association instance a of type T_a related to t because of an association definition $\rho = \langle \text{“}s\text{”}, \mathrm{T}, \mathrm{T}_a, d, r_1, r_2 \rangle$ is treated like a subelement, potentially with a potential variable $count_a$. It also defines an additional variable $connect_a$. The domain of this variable contains the possible associations in the destination type. Again, a generated model attribute represents the $connect_a$ variable in the type instances, and an activity variable $A_{connect_a}$ takes care of its activity.

For each pair (a_1, a_2) of possibly connected associations, constraints ensure that the two association instances match, and that all the attributes (and association types) are equal.

The following constraints are thus added to the CondCSP:

$$A_{a_1} \wedge (connect_{a_1} = a_2) \Leftrightarrow A_{a_2} \wedge (connect_{a_2} = a_1)$$

and

$$connect_{a_1} = a_2 \wedge connect_{a_2} = a_1 \\ \Rightarrow (type_{a_1} = type_{a_2}) \wedge (\forall \alpha = \langle \text{“}x\text{”}, \mathrm{T}_a, ... \rangle \in \mathcal{D}_\mathrm{S}, x_{a_1} = x_{a_2})$$

where x_{a_1} and x_{a_2} are the variables for the instances of α in a_1 and a_2.

Example 54 *Consider the association definitions*

$$\begin{aligned} \rho_{stylusTS} &= \langle \text{“}stylus\text{”}, \text{TouchScreen}, \text{Stylus}, \textsc{None}, 0, 1 \rangle \\ \rho_{stylusE} &= \langle \text{“}stylus\text{”}, \text{Enclosure}, \text{Stylus}, \textsc{None}, 0, 1 \rangle \end{aligned}$$

defined in the physical structure view of the case study. For each instance ts of TouchScreen *and each instance e of* Enclosure*, if* $stylus_{ts}$ *and* $stylus_e$ *are the variables for the associations* $\rho_{stylusTS}$ *and* $\rho_{stylusTS}$*, we have*

$$A_{stylus_{ts}} \wedge (connect_{stylus_{ts}} = stylus_e) \Leftrightarrow A_{stylus_e} \wedge (connect_{stylus_e} = stylus_{ts})$$

and

$$connect_{stylus_{ts}} = stylus_e \wedge connect_{stylus_e} = stylus_{ts} \\ \Rightarrow (type_{stylus_{ts}} = type_{stylus_e}) \wedge (size_{stylus_{ts}} = size_{stylus_e})$$

with $size_{stylus_{ts}}$ *and* $size_{stylus_e}$ *being the variables for the attribute size of the* Stylus *association type for the two instances created from* $\rho_{stylusTS}$ *and* $\rho_{stylusTS}$*.*

This means that, for the two association instances $stylus_{ts}$ *in ts and* $stylus_e$ *in e, if one connects to the other, then they should both be connected together, and, in that case, the* size *attribute of the association instance* $stylus_{ts}$ *should be mirrored in* $stylus_e$*.*

9.2.6 Model Constraints

We describe in this section how the different constraints expressed in the configuration model are taken into account in the Conditional Constraint Satisfaction Problem.

9.2.6.1 Compatibility constraints

Compatibility constraints are also subject to taxonomy constraints. In a type instance t of type T, for a constraint $c = \langle \mathrm{T}', e^c \rangle$ with e^c a *symbolic* constraint expression, the following constraint, including its activity guard, is added to the Conditional CSP:

$$A_{c_{CSP}} \wedge A_{type_t} \wedge type_t \in (\{\mathcal{ST}(\mathrm{T}') \cup \{\mathrm{T}'\}) \Rightarrow c_{CSP}$$

with c_{CSP} being the CSP constraint generated from the constraint expression e^c, where all the model attributes are replaced by their attribute variables for the corresponding instance t. T

his means that the constraint is only taken into account if its variables are active and the instance t has a corresponding type.

The case of table constraints is a bit different. As DnSTR only takes table constraints into account, using implication constraints for rows in tables is not possible. The activity of a table constraint is thus solved by inserting a new column and a new line into each table.

The new column is used for the table constraint's activity variable A_{table}. A_{table} is the conjunction of the activity variable for the property variables in the table as well as the potential guard corresponding to the taxonomic relation.

The column for the A_{table} variable is equal to true for each row except for the first one. That first row corresponds to the case where the constraint is not active, and thus all the other cells in the row are filled with a special value "*". The DnSTR algorithm has been modified to interpret this new value as the entire domain of values for the variable specified in each column.

$size_e$	$size_s$	nbSlots
"115x58"	3	6
"240x190"	9	10
"268x178"	10	12
"295x210"	12	14

A_{table}	$size_e$	$size_s$	nbSlots
0	*	*	*
1	"115x58"	3	6
1	"240x190"	9	10
1	"268x178"	10	12
1	"295x210"	12	14

Figure 9.1: Transformation of the TableSize table constraint from the Mobile Device physical structure view. The first column of the new table represents the activity guard of the constraint, while the first row is filled with the value "*", that represents the entire domain of each associated variable.

Example 55 *Consider again the table constraint* TableSize *from Example 17 declared in ProCoLa by:*

```
TableSize: {
type:table[enclosure.size,screen.size,motherboard.nbSlots];
value: {
   ["115x58" , 3 , 6 ],
   ["240x190", 9 , 10],
   ["268x178", 10, 12],
   ["295x210", 12, 14]
 };
};
```

For an instance dP of type DevicePhysical*, the conversion to CondCSP of the table constraint is shown in Figure 9.1, with $size_e$, $size_s$ and nbSlots respectively the variables for the attributes*

$\alpha_{sizeE} = \langle$"*size*", Enclosure, D_{size_e}, VISIBLE, $1\rangle$
$\alpha_{sizeS} = \langle$"*size*", Screen, D_{size_s}, VISIBLE, $1\rangle$
$\alpha_{nbSlots} = \langle$"*nbSlots*", Motherboard, $\mathrm{D}_{nbSlots}$, READONLY, $1\rangle$

The activity variable of the table constraint is equal to

$$A_{table} = A_{size_e} \wedge A_{size_s} \wedge A_{nbSlots} \wedge A_{type_{dP}} \wedge type_{dP} \in \{\text{DevicePhysical}\}$$

If A_{table} evaluates to false*, any values can fit for the variables in the table constraint, which is equivalent to the constraint being inactive.*

9.2.6.2 Implementation constraints

Activity in an implementation constraint $c = \langle \mathrm{T}, e^c_{\mathrm{C}}, e^i_{\mathrm{P}}, \mathrm{Op} \rangle$ is handled differently for the child expression e^c_{C} and the parent expression e^i_{P}. On the one hand, as for compatibility constraints, an activity guard is introduced: this activity guard contains the taxonomic constraint as well as the activity for variables in e^c_{C}.

On the other hand, the semantics of e^i_{P} include existential constraints on the different properties (attributes, subelements and maybe associations) specified in the implementation expression. Indeed, properties in e^i_{P} are not dependent on the context type T in which the constraint is declared, and thus associated CondCSP variables are not specific to instances of T, like a compatibility constraint would. Instead, these properties are defined using a local type, e.g., RFCard $\rightarrow \alpha_{cardType}$, where the attribute definition $\alpha_{cardType}$ is dependent on the RFCard type, and thus must be considered for all instances of RFCard.

Variables in e^i_{P} must thus be "expanded" to reflect the existential quantifier implied by the semantics of the expression. The conversion of implementation expressions into CSP constraints gives rise to two cases. First, boolean constructs involving properties are expanded using the CondCSP variables for all potential instances of the local type specified. The `Present`(T) function is also expanded directly for all instance of T. Corresponding activity variables ensure that a variable is not taken into account if it is inactive.

Example 56 *Consider two instances t_1, t_2 of a type* T*. The following expansions are made during the creation of CondCSP constraints:*

$$\begin{array}{lcl} \mathrm{T} \rightarrow ... \rightarrow \alpha_b & \rightsquigarrow & (A_{b_{t_1}} \wedge b_{t_1}) \vee (A_{b_{t_2}} \wedge b_{t_2}) \\ \mathrm{T} \rightarrow ... \rightarrow \delta_s \text{ is } \mathrm{T}' & \rightsquigarrow & (A_{s_{t_1}} \wedge (s_{t_1} \text{ is } \mathrm{T}')) \vee (A_{s_{t_2}} \wedge (s_{t_2} \text{ is } \mathrm{T}')) \\ \mathrm{Present}(\mathrm{T}) & \rightsquigarrow & (A_{t_1} \wedge (t_1 \text{ is } \mathrm{T})) \vee (A_{t_2} \wedge (t_2 \text{ is } \mathrm{T})) \end{array}$$

with α_b (resp. δ_s) being an attribute definition with a boolean domain (resp. a subelement or association definition) and b_{t_i} (resp. s_{t_i}) the corresponding CondCSP variable derived from instance t_i.

For example, in the first case, this means that both variable b_{t_1} and b_{t_2} representing α_b in each instance t_1 and t_2 have to be taken into account due to the implicit existential semantics of the implementation constraint.

On the other hand, for any binary relational expression, i.e., $e^i_1\ op_r\ e^i_2$, model variables with arithmetic domains cannot be directly used in disjunction as

above. If the model variables in e_1^i and e_2^i are the set $V^i = (v_1^i, ..., v_n^i)$, each model variable $v_j^i ::= \mathrm{T}_j \rightarrow v_j$ is expanded into variable instances x_{jk} for all the instances t_{jk} of its local type T_j. Sets $X_1, ..., X_p$ are created with all the different combinations of variables x_{jk} for the set V^i, and are used to convert the binary relational expression $e_1^i \ op_r \ e_2^i$ into

$$(A_{X_1} \wedge c_{e_1^i}[V^i \mapsto X_1] \ op_r \ c_{e_2^i}[V^i \mapsto X_1]) \vee ... \vee \\ (A_{X_p} \wedge c_{e_1^i}[V^i \mapsto X_p] \ op_r \ c_{e_2^i}[V^i \mapsto X_p])$$

with $c_{e_1^i}[V^i \mapsto X_1]$ representing the CondCSP constraint expression derived from e_1^i where model variables from V^i are converted into CondCSP variables from X_1, and A_{X_1} representing the activity guard from all variables in X_1.

Example 57 *Consider two instances t_1, t_2 of a type* T *and two instances t_3, t_4 of a type* T*'. The expression*

$$\mathrm{T} \rightarrow \alpha_a \geq \mathrm{T}' \rightarrow \alpha_b$$

contains the model variables $V^i = (\mathrm{T} \rightarrow \alpha_a, \mathrm{T}' \rightarrow \alpha_b)$ and would generate the combination sets

$$\begin{array}{lcl} X_1 & = & (a_{t_1}, b_{t_3}) \\ X_2 & = & (a_{t_1}, b_{t_4}) \\ X_3 & = & (a_{t_2}, b_{t_3}) \\ X_4 & = & (a_{t_2}, b_{t_4}) \end{array}$$

The generated CondCSP constraint would then be

$$(A_{a_{t_1}} \wedge A_{b_{t_3}} \wedge (a_{t_1} \geq b_{t_3})) \vee (A_{a_{t_1}} \wedge A_{b_{t_4}} \wedge (a_{t_1} \geq b_{t_4})) \vee \\ (A_{a_{t_2}} \wedge A_{b_{t_3}} \wedge (a_{t_2} \geq b_{t_3})) \vee (A_{a_{t_2}} \wedge A_{b_{t_4}} \wedge (a_{t_2} \geq b_{t_4}))$$

9.2.6.3 Mapping constraints

Finally, mapping constraints are integrated into the instances of structure views. Considering a mapping constraint $c = \langle \mathrm{T}_s, \mathrm{T}_r, e^c \rangle$, for each instance t_s of the structure type T_s, an instance t_r of the realisation type T_r is created.

Its activity is bound by

$$A_{t_r} \Leftrightarrow (A_{c_{CSP}} \wedge c_{CSP})$$

where c_{CSP} is the CSP constraint generated from the expression e^c, where all the model attributes are replaced by their attribute variables for the corresponding instance t_s. The activity guard $A_{c_{CSP}}$ ensures that if a variable is inactive in c_{CSP}, the expression $(c_{CSP} \wedge A_{c_{CSP}})$ is false and the realisation instance is not included in the configuration. Activity guards from taxonomic relations may also be added to the final constraint.

Example 58 *If scr is an instance of* Screen*, the following mapping constraint*

$$c_{coating} \quad = \quad \langle \text{TouchScreen}, \text{Coating}, \alpha_{oleophobicCoating} = \text{true} \rangle$$

generates, with oC_{scr} *the variable for the attribute* $\alpha_{oleophobicCoating}$ *in scr:*

$$A_{coating} \Leftrightarrow \\ (A_{type_{scr}} \wedge type_{scr} \in \{\text{TouchScreen}\} \wedge A_{oC_{scr}} \wedge oC_{scr} = true)$$

9.2.7 Successor relations

Successor relations constrain the order in which operations should happen. To model these relations in the Conditional Constraint Satisfaction Problem, each instance t of an operation type $\text{T} \in \mathcal{T}^o_{\text{R}}$ for a resource view R is associated a variable $index_t$ with an integer domain, and related to a generated model attribute. Then, for two instances t_1 and t_2 of types T_1 and T_2 for which $\langle \text{T}_1, \text{T}_2 \rangle \in \mathcal{N}_{\text{R}}$, the following constraint is added to the CondCSP:

$$A_{t_1} \wedge A_{t_2} \Rightarrow index_{t_1} < index_{t_2}$$

9.2.8 Resources

Each resource declared in structure views needs to be balanced: the amount produced for a resource must always be equal or greater than the amount consumed. Each time an instance produces (resp. consumes) a part of the resource, the arithmetic expression defining the production (resp. consumption) is summed (resp. subtracted) to create the final resource constraint.

In case of resource constraints, activity guards are handled differently than most other constraints. If a production (resp. consumption) expression is inactive (e.g., because one of the variables involved is inactive), the expression should be omitted in the resource calculation, instead of evaluating the whole resource constraint to true. This is handled by multiplying the numeric value (0 or 1) of the activity guards of the expression to its production (resp. consumption), so that it is not taken into account if it is inactive (i.e., the value is equal to 0).

If P (resp. C) is the set of production (resp. consumption) expressions for a specific resource, the corresponding resource constraint is expressed by:

$$0 \leq \sum_{e_P \in P}(A_{e_P} * e_P) - \sum_{e_C \in C}(A_{e_C} * e_C)$$

Example 59 *Consider the* AvailableSlots *resource declared in the Mobile Device product family. Resource uses defined in* $\mathcal{U}_{S_{phys}}$ *have to be taken into account for all contributing types:*

$$0 \leq ((A_{mb_{nbSlots}} \wedge A_{mb_{nbChips}}) * (mb_{nbSlots} - mb_{nbChips}) - (A_{rfCard} * 1 + A_{processor} * 1 + A_{gps} * 1 + A_{ethernet} * 1 + \sum_{i=1}^{3} A_{wirelessChip_i} * 1))$$

with $mb_{nbSlots}$ *and* $mb_{nbChips}$ *the variables for the motherboard's* $\alpha_{nbSlots}$ *and* $\alpha_{nbChips}$ *attributes, and* $rfCard$*,* $processor$*,* $wirelessChip_i$*,* $ethernet$ *and* gps *respectively the instances of the* RFCard*,* Processor*,* WirelessChip*,* Ethernet *and* GPS *types.*

All contributions are summed up in this constraint when the corresponding variable instances are active. For example, if the instance mb of type Motherboard *is active (and thus the variables for its attributes* $mb_{nbSlots}$ *and* $mb_{nbChips}$ *are active), then it produces an amount of resource* AvailableSlots *equal to the value of* $(mb_{nbSlots} - mb_{nbChips})$*. Also, if the gps instance is active, it consumes the resource by 1.*

9.3 Discussion

Using Conditional CSPs to solve the configuration problem in our modelling framework has both advantages and limitations in regards to other formalisms and solving techniques like Generative CSPs [61,95]. One important issue with Conditional CSPs is that all variables need to be declared when the CSP is created. This means that it is not possible to generate variables on-demand,

which does not permit to model features such as partonomic relations with an unbounded number of subelements. The variables representing all instances of the types and model elements must be generated from the start, which can lead to an large amount of variables. As in the previous sections, CondCSP constraints for partonomic and taxonomic relations must consider all the cases for the different instances of attributes and subelements, and this for all potential subtypes the instances can take. It can also get complicated when dealing with quantifiers, as is the case in implementation constraints: all combinations of instance variables must be considered.

However, the formulation as Conditional CSPs offers some very interesting advantages. The first one lies in the simplicity of the solving implementation. It is indeed possible to use the same algorithms as for normal CSPs, as handling the activity of the variables is equivalent to adding constraints including some activity variables. Then, state-of-the-art algorithms need almost no modifications to handle the newly formulated problems. In the case of our configuration framework, this permits to use DCSP algorithms such as AC|DC-2i [11] or our own DnSTR algorithm (Chapter 8) for dynamic arc consistency in table constraints.

Also, the formulation of constraints that involve all the variables permits a full propagation of the changes in one single phase, without requiring a variable generation phase like in Generative CSP solving. The consistency is ensured on all variables via classic propagation: all elements of the model are linked together, which streamline the propagation of domain changes, whether it comes from a resource constraint, the addition of a new component or feature, the connection of associations or the subtyping of an instance.

Finally, using DCSP for solving constraints at runtime brings great flexibility to the user of the model. DCSP can indeed be used to handle the dynamic addition and retraction of the end-user value assignments, but it can also be used to modify the model's constraints at runtime. This can greatly facilitate the work of the knowledge engineer with respect to diagnosis and debugging of models, as it allows to observe in real-time the effect of adding and removing model constraints on-the-fly.

Part IV

Prototype and Evaluation

Chapter 10

Prototype Implementation of the Framework

Creating a product model can be a long and complex process, even if the product modeller knows the modelling language well. In order to assist him in the modelling process, the ProCoLa modelling language has been integrated into a development environment, supplemented by some tool support to facilitate the creation and maintenance of the model. A runtime system for our framework has also been implemented in order to provide the necessary support for configuring models. In this chapter, we present our prototype implementation and discuss the features of the system. The current version of the prototype can be found in [74].

10.1 Language Integration

In order to provide support for developing configuration models in ProCoLa, we implemented a prototype compiler for the language.

ProCoLa models are recognised using two elements:

1. A *lexical analyser*, that takes as input ProCoLa files and returns a set of tokens identifying the different elements in the files, like keywords (e.g., "componentType"), identifiers (e.g., for type names), ...etc.

2. A *language parser*, which, given the grammar of the ProCoLa language and the tokens from the lexical analyser, tries to identify the different syntactic patterns in order to create an internal representation of the ProCoLa views described in the input file.

The ProCoLa compiler has been integrated within the Microsoft Visual Studio environment [94], by implementing a language service to extend the functionalities of this environment for ProCoLa models. Visual Studio provides tools to support the model development, including syntax highlighting, word completion, outlining, or syntax and semantic checks. It is our belief that such a development environment can greatly improve the productivity of the product modeller. The adaptation time to a configuration-specific language such as ProCoLa is reduced by the accessibility of the keywords defined according to well-defined configuration concepts, and an integrated textual language can provide an easy access to techniques for checking the well-formedness of the product configuration model.

Figure 10.1 shows a screenshot of a Visual Studio window in which the ProCoLa file for the physical structure view of our example case study is opened. The main panel shows the ProCoLa model of the Mobile Device example with syntax highlighting and word completion tooltip. Another advantage of such integration is the potential for extending the language integration. For example, the top left panel in Figure 10.1 shows a tree-structured model outline, akin to the PVM representation [51]. Although ProCoLa is a textual language, it is thus possible to create more graphical add-ons to give a better overview of the model.

10.2 Tool Support for Modelling

Developing a large configuration model can be a difficult and time-consuming task, especially when several views of the model are considered, due to their dependencies and interactions. In this section, we present the implementation of several mechanisms to provide additional tool support during a ProCoLa model's creation and maintenance.

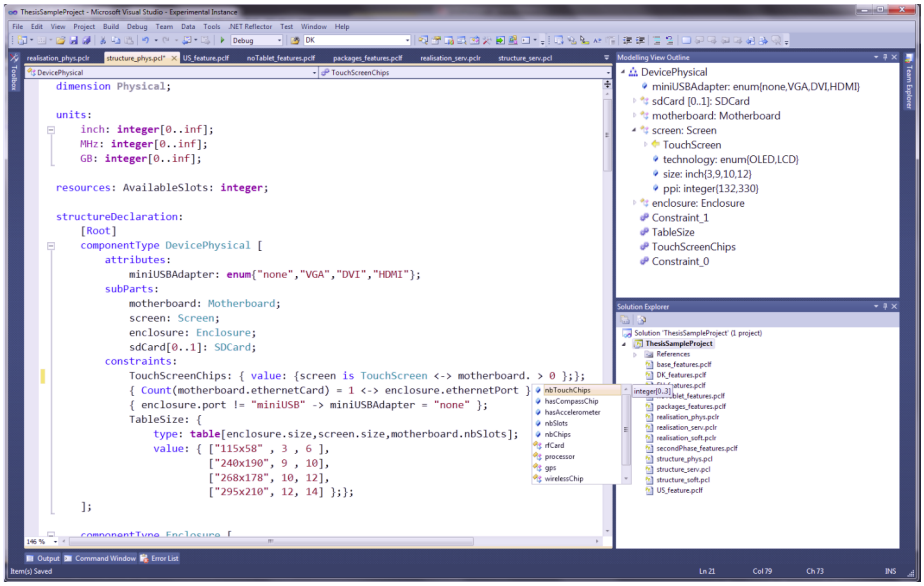


Figure 10.1: ProCoLa language integration into the Visual Studio development environment.

10.2.1 UML

As explained before, providing a graphical representation of the configuration model is important during model creation and also in the context of communication between stakeholders and model knowledge exchange. The Unified Modelling Language (UML) is a well-known language to represent object-oriented models, and its potential for representing configuration models has been demonstrated in Section 3.1. Although shortcomings of the language have been discussed, UML remains a widely used and studied formalism, as opposed to SysML (Section 3.2), relatively recent. UML is thus of prime interest for disseminating knowledge about product models in addition to ProCoLa.

We implemented a generator that automatically translates the ProCoLa model into a UML model and generates a profile to define relevant stereotypes for the different views of ProCoLa. The UML models are created using the XML Metadata Interexchange (XMI) format, an XML-based OMG standard for the exchange of UML models. One of the drawbacks of UML is the differences between the UML tools available currently when it comes to the compliance with the UML standard. The UML models generated from ProCoLa are thus tailored to be imported in the open source *Eclipse UML2 Tools* [102].

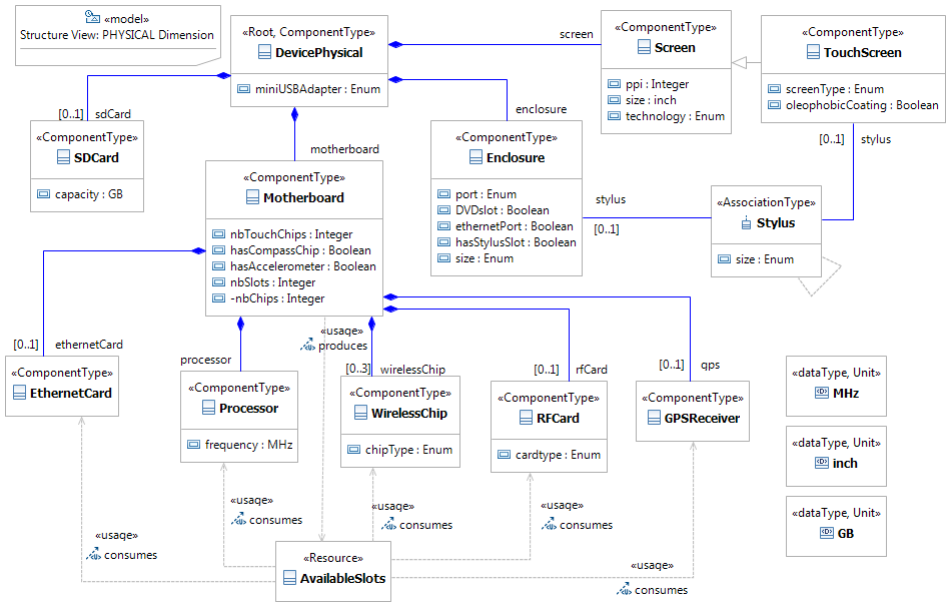


Figure 10.2: UML representation for the physical structure view of the Mobile Device product family.

Example 60 *The UML generated from the physical structure view of our example product family can be seen Figure 10.2. The XMI file for this ProCoLa view has been imported into the Eclipse tool, along with the ProCoLa profile.*

10.2.2 Table Constraints

Table constraints are very often used in product configuration because they can easily represent product catalogues. The design of ProCoLa includes the possibility of declaring table constraints inline as compatibility constraints in the feature and structure views.

However, product catalogues can become very large, which makes writing the corresponding constraints in ProCoLa tedious and difficult to maintain. That is why we implemented connections between ProCoLa and tools such as spreadsheet applications and databases in order to give the modeller the possibility to create and maintain large table constraints outside of the ProCoLa views. Besides providing a more adapted interface for declaring these constraints, such

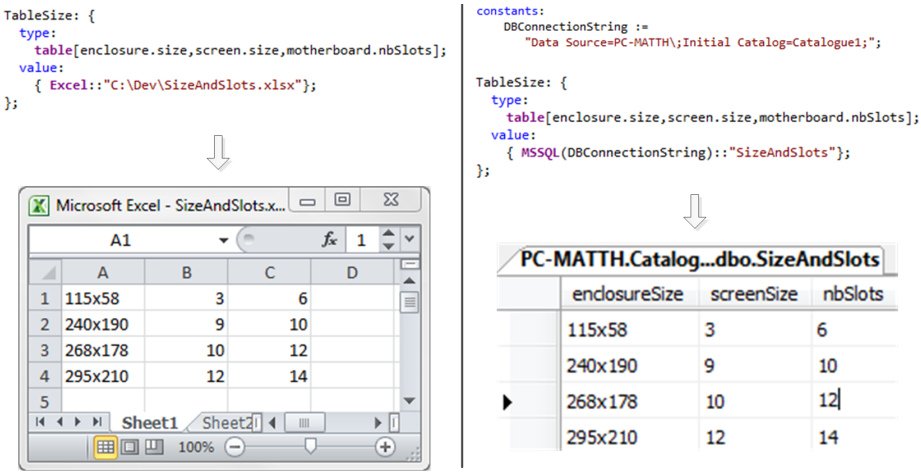


Figure 10.3: Spreadsheet applications and database integration for table constraints.

tools also permit to separate these product catalogues from the design of the view: the management of these constraints can be delegated to separate modellers, and the product catalogues may be shared and exchanged with relevant stakeholders.

The two connectors currently implemented interface with the Excel application [35] and the Microsoft SQL Server database [89]. Spreadsheets and database tables are dynamically translated to ProCoLa table constraints when the model instance is created at configuration time in order to take into account the latest version of the constraints.

Example 61 *Figure 10.3 shows how to declare in ProCoLa the table constraint TableSize from Example 17, created and maintained using Excel or SQL Server.*

10.2.3 Analyses Implementation

One of the main issue when dealing with large configuration models is their maintenance. Our modelling framework is based on multiple modelling views, which permit to create an integrated model, but may also make it more complex. Models evolve, and the elements of the different views may be modified multiple times in order to fit a model's purpose. It can thus be difficult to keep track of the model elements' dependencies, to introduce cycles in the model's architecture or to clutter the model with unused elements.

In order to help the modeller in creating and maintaining the models, the three analyses from Section 6.4.2 have been implemented for our prototype:

- The Dependency Analysis permits to identify which elements are dependent on a specific type, attribute, etc. The implementation of this analysis in our prototype gives the possibility to the modeller to lookup the results of such an analysis from within the ProCoLa models editor by selecting a specific element that he would like to removed, for example.
- The Use Analysis and the Cycle Analysis can be used on the full ProCoLa model. The Use Analysis permits to detect the unused elements of each view, giving some insights on how to cleanup the model, e.g., after removing a model type. The Cycle Analysis gives feedback on whether partonomic and taxonomic cycles exist in the model (and where they are). Such cycles can come from a bad model design or simply represent a modelling error, and have to be removed in order to ensure the well-formedness of the ProCoLa model.

10.3 Runtime Implementation

We implemented a runtime system for ProCoLa. After choosing a feature view in the hierarchy, the different views of a configuration model are instantiated as described in Section 9.2, and the resulting instances are presented using a configuration user interface.

The end-user can enter his requirements by assigning values to the different attributes of the model, or remove some of the requirements provided earlier. The consistency of the Conditional Constraint Satisfaction Problem corresponding to the configuration instance is ensured by a C# constraint solver using algorithms such as DnSTR and AC|DC-2i (Section 9.1). Each time a value is set or cleared, the solver propagates the event by maintaining maximal arc consistency, deleting or restoring values from the domains of the other variables. This results in the reduction (or augmentation) of the domains of possible values for the other attributes, or a change of activity that renders certain variables inactive, hiding some of the attributes and components from the end-user.

The configuration interface for the service structural view of the Mobile Device product family can be seen in Figure 10.4. The main goal of this interface is to provide basic guidance to the user for configuration: the order of variable selection is free (in contrast to other systems with incremental configuration), and no explanations are provided when a conflict occurs, although inconsistent values cannot be selected.

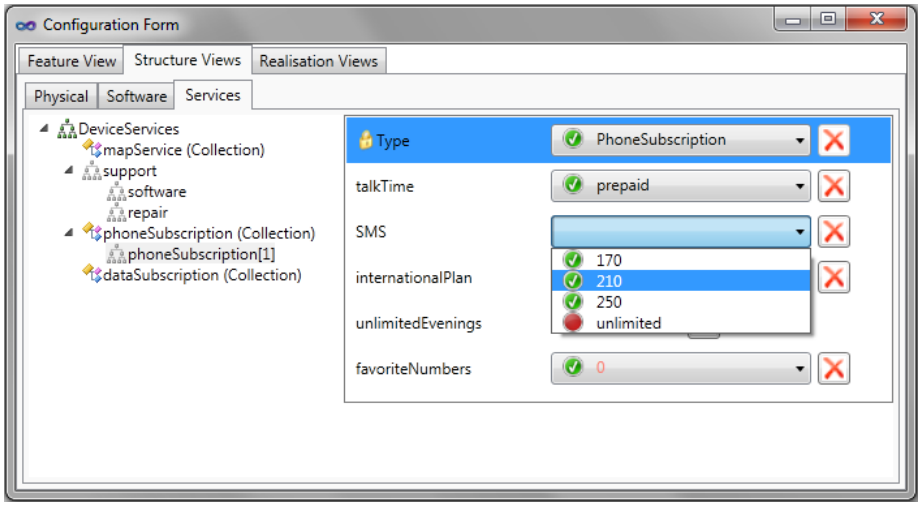


Figure 10.4: Configuration of the Mobile Device product family. The value for *talkTime* has been fixed by the end-user, while the one for *favoriteNumber* has been bound by the solver.

10.4 Debugging

Errors in complex models may provoke unwanted behaviours and incorrect results, and visualisation tools can provide support to the modeller for understanding why. Several tools for constraint debugging have been designed for general constraint programs.

For example, Meier presents Grace [66], a graphical environment for tracing Constraint Logic Program with Finite Domains on top of the constraint solver ECLiPSe; Goualard and Benhamou developed a tool [46] based on a hierarchical organization of sets of constraints; Bouvier [21] contributes with Prolog IV visual tools that are able to set breakpoints to stop the execution of the program; and Carro and Hermenegildo [22] with visualization tools focusing on the representation of run-time values of the variables and the constraints among them.

However, all these techniques and tools are not specifically targeted at product configuration. One of the main issues remains being able to provide relevant debugging information to a product modeller that is not an expert in constraint programming. The concept of model-based diagnosis has thus been adapted to configuration by Felfernig et al. [38] in order to test the knowledge base with

test cases. Those test cases are originally declared as either valid or invalid configurations, and are fed to the system to determine whether the product model is correctly defined or not.

We focus in our framework on the representation of the configuration model at runtime, attached to the real object to be solved, i.e., the constraint model. The main issue is indeed to map the two models such that the product modeller can understand as much as possible the interactions between the different variables of the model, as well as the consequences of any change in the user requirements. Representing the constraint graph is an easy task, but representing it in such a way that it is accessible to a product modeller is more challenging. Indeed, realistic configuration models may have thousands of variables and constraints, so representing the entire constraint graph is impossible and without any practical use.

In our approach to debugging we use the hierarchical structure of the configuration model to represent the constraint graph. Nodes in the graph represent variables in the constraint network, and have different visuals according to the nature of their configuration counterpart (type instances, attributes, etc). Each type instance node can be expanded or collapsed, in which case nodes representing any child variables are not displayed. Constraints defined in the product model are displayed as edges between the different variables. As the graph is aimed at the product modeller, activity variables are not represented in the graph, but activity constraints are displayed between the corresponding variables instead. Different types of constraints (user defined or activity constraints) are also visually different.

Finally, while the end-user configuring the product should only see relevant attributes and components, the product modeller debugging his model should be able to see all variables that can interact with the system, even when they are inactive. Inactive variables are thus also shown to him both in the debugging configuration form and in the constraint graph representation.

A screenshot of the implementation of this hierarchical graph can be seen in Figure 10.5. Information about nodes and edges is directly accessible by clicking on the nodes representing objects. A contextual menu is also available to collapse and expand nodes, as well as to interact with constraints. One of the main advantages of this graph is that it is fully interactive. It is possible for the product modeller to modify it by adding and removing constraints during debugging, without having to stop the debugging process for modifying the model. This feature is available in the solving engine implementation using DCSP algorithms such as AC—DC-2i or DnSTR (Section 9.1).

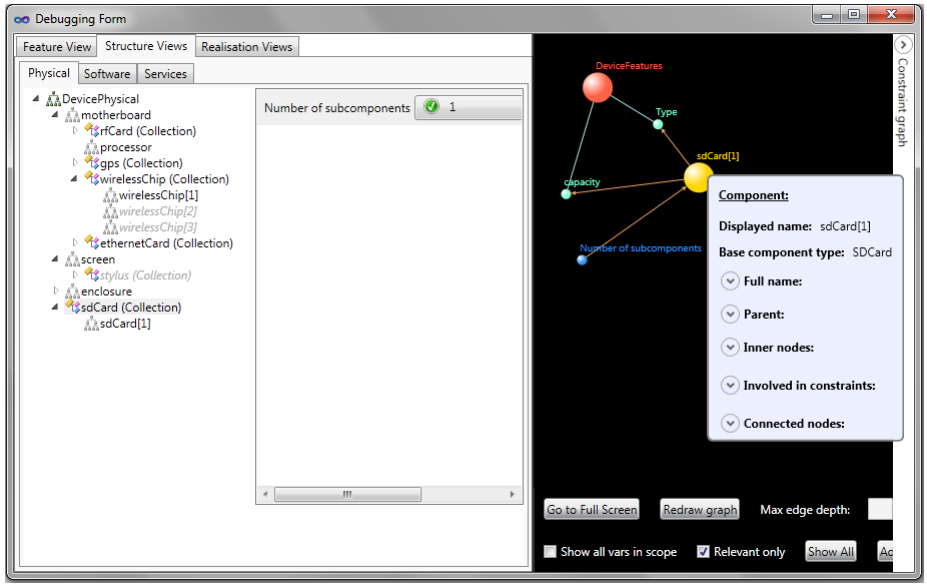


Figure 10.5: ProCoLa framework debugging form. The graph on the right represents the part of the constraint graph where the highlighted attribute "Number of subcomponents" ($count_{sdCard}$ variable) is involved. Contextual information is shown for the expanded node sdCard[1], while the DeviceFeatures node is collapsed.

10.5 Summary and Discussion

In this chapter, we presented the implementation of a prototype for our configuration framework. The different parts of the framework (Figure 10.6) together provide tool support to the modeller for the development of configuration models for heterogeneous product families. It is based on the ProCoLa modelling language, which is integrated into the Visual Studio development environment, and interacts with a constraint solver at configuration time.

Tool support is provided in the prototype, from verification of the model's well-formedness (language services, model analyses) to the integration with a well-known graphical language like UML and tools like spreadsheet applications and databases. These tools are essential to assist in the creation and maintainable of the models, in particular when the number of views and types increases, as it can become difficult to keep an overview of all model elements for the knowledge engineers.

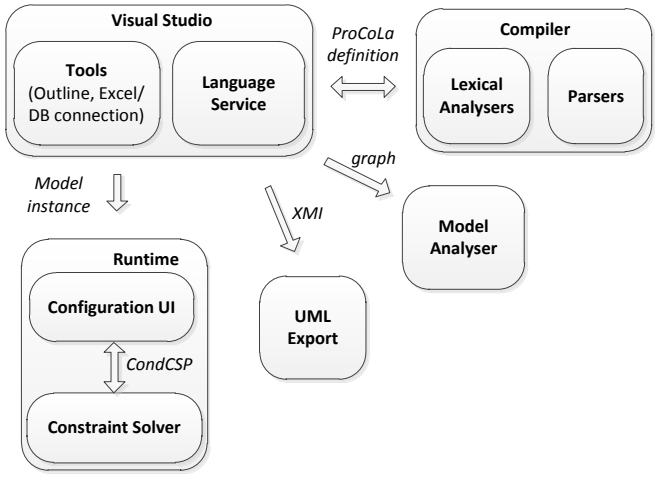


Figure 10.6: Configuration Framework architecture

Although the configuration models created with our framework can be exported as UML diagrams, the development of the different views focuses on the ProCoLa modelling language. Some users may be reluctant to use a textual language instead of a graphical interface to create configuration models, although we argue that using a language with standardised keywords and a well-defined conceptual approach as in ProCoLa may be an advantage. Firstly, despite the tool support that a development environment can provide, a model in ProCoLa may be viewed and edited in a simple text editor, without the need of a special tool, removing some barriers when communicating and exchanging product knowledge. The clear semantics of ProCoLa also allow an easy interpretation of the language, which increases the potential for implementing tools or customised graphical interfaces. Finally, we believe that a textual language with appropriate development support may be faster and more practical for advanced users when dealing with certain repetitive tasks, e.g., creating similar types, adding many subpart definitions or writing complex constraints.

The debugging tool implemented in our prototype is an attempt to help the knowledge engineer in understanding the runtime behaviour of the configuration models. The hierarchical representation of the graph helps reduce the amount of variables displayed to the modeller by collapsing and expanding parts of the graph. It is also important to show only the variables relevant to the current context in the configuration. Constraint graphs representing configuration problems can sometimes be split into smaller subgraphs. This is used to show only the relevant graph to the product modeller when he modifies a specific variable.

However, the usefulness of the debugging tool is limited when dealing with larger model, as it may be difficult to foresee all the changes a value assignment might provoke. We thus believe our debugging approach to be a complement to more advanced techniques, such as model-based diagnosis [38]. The visualisation of the configured system and the interactivity of the tool give a lot of freedom to the knowledge engineer, who can modify the model on-the-fly using dynamic constraint addition/retraction, which is made possible by the Dynamic CSP algorithms implemented in the constraint solver. On the other hand, the work by Felfernig et al. provides a generalized diagnostic of the model, but requires the creation of test cases. The question of how to define those test cases is still a good topic for more research [39]. Fully appreciating the potential of our debugging tool and the whole prototype requires large user experience studies, which is out of scope of this dissertation and left for future work.

Chapter 11

Framework Evaluation

Finding relevant models for evaluating a configuration framework is not an easy task. Models presented in the literature are often rather small in the number of variables and constraints, and most of the benchmark models coming from industry are very difficult (or almost impossible) to obtain in a project without real-life industrial case studies [100].

In this chapter, we describe a series of experiments that aim at providing some insights on the capabilities of the implemented framework. The main issue in evaluating our work is to find a single configuration model that incorporates all the dimensions described in our framework. That is why we decided to design a larger version of our case study, the Mobile Device product family, in order to assess to what extent our configuration framework is able to cope with its modelling and solving. We thus provide some experimental results for applying the different techniques developed in this dissertation. Moreover, in order to obtain more insights on the performance of our configuration framework, we modelled and experimented on several test cases used in the literature, as well as some industrial configurators benchmarks.

We will now introduce the different benchmark programs in Section 11.1, and present the evaluation results in Section 11.2.

11.1 Benchmark models

Before discussing the evaluation results, we first give an overview of the benchmark models used in this chapter. We introduce each of them, and provide some information on their general properties.

Eight different configuration models have been used in these experiments:

- **Extended Mobile Device**: The Mobile Device example described in Section 2.2 has been extended for the purpose of this evaluation. Many types have been added to the models, including feature types (e.g., cloud features, graphics), structure types (e.g., cameras, keyboard, synchronisation services, additional software libraries and applications) and corresponding realisation types. Additional feature views have also been designed, e.g., in order to target more markets. The ProCoLa files for the example can be found in [74].
- **ESVS, FS, FX, and Machine**: These configuration models represent four product families used in [99]. The first three products are screw compressors manufactured by Gardner Denver [34]. Each configuration model represents a complete sales configuration view of a compressor family. The models are detailed to production quality, except for some constant values. The fourth product represents a 4-wheel vehicle. It was modeled for demonstration purposes and represents about half of the sales view of the product. These models are originally defined in the PCML modelling language [101].
- **Bike, PC, BigPC**: These three models represent a Bike model and two versions of a Personal Computer product family. They are available through the CLib Configuration Benchmarks Library [25], where they are modelled using the PM language from [70].
- **Renault**: This relatively large test model is used in a benchmark by Amilhastre et al [5]. It has been provided by Renault DVI [83], a french car manufacturing company, and deals with the configuration of a family of cars, called Renault Megane. The variables in this problem represent the type of engine, the country, options like air cooling, etc. This model is also available in the CLib Configuration Benchmarks Library [25].

The test cases modelled in ProCoLa are characterized in Tables 11.1 and 11.2. The "#types" column gives the numbers of abstract and all feature and structure types. Realisation types, which are only modelled in the Mobile Device model, are not shown in these measures. The "#attributes" column indicates

Table 11.1: Properties of the benchmark programs

Model	#types		#attributes		constraints		
	ab	total	avg	total	total	avg arity	ref
Mobile Device	2	114	1.5	174	222	2.3	2.6
ESVS	3	10	2.4	24	11	2.2	20
FS	1	4	5.5	22	14	3.2	24.9
FX	0	1	18	18	21	2.7	13.9
Machine	5	30	0.3	8	7	2.0	2.3
Bike	0	9	3.7	34	31	2.1	10.5
PC	0	6	3.8	23	16	2.5	4.1
BigPC	0	20	5.2	103	54	3.6	49
Renault	0	1	99	99	113	4.9	11400

Table 11.2: Properties of the benchmark programs (continued)

Model	partonomy			#taxonomic
	avg	dynamic	total	relations
Mobile Device	0.96	64	109	16
ESVS	0.30	0	3	6
FS	0.25	0	1	2
FX	0	0	0	0
Machine	0.33	10	15	15
Bike	0.56	0	5	0
PC	1.33	0	8	0
BigPC	1	0	20	0
Renault	0	0	0	0

the average number of attributes per type, as well as the total number of attributes in each model. The column "constraints" specifies the total number of user-defined constraints, the average constraint arity (i.e., the number of variables per constraint) and the average number of references to properties in a constraint. In the second table, the "partonomy" column states the average partonomic relations per type (as well as associations in structure types), the number of relations with dynamic cardinalities ($[r_1, r_2]$ with $r_1 \neq r_2$), and the total number of partonomic relations in the model. Finally, the "#taxonomic relations" column specifies the number of taxonomic (subtypes/supertypes) relations between types of the model.

These tables show clear differences between the models. For example, our Mobile Device model has much more types than the other models, as its architecture

is much more complex, and involves several views. Also, very few models make use of dynamic partonomic relations and inheritance. Instead, models like the compressor product family (ESVS, FS, FX) are using mainly attributes. The three biggest models (Mobile Device, BigPC and Renault) are fairly different from each other. On the contrary to the Mobile Device, Renault has a minimalist structure, with only one type and many attributes in it. This model has been made anonymous in order to prevent any sensitive information to be divulged publicly[1], so all variables are called *VarX*. Although BigPC looks more balanced, its partonomic tree is quite flat: the average value of one partonomic relation per type is misleading, as most of these relations are within the root component.

The average constraint arity is a good indication of the complexity of each model's constraints. All models have a reasonable value for this metric, which indicates that constraints are broken up in a relatively nice way. However, the average number of property references per constraint shows that some models have much more than one reference per property involved. This is mainly due to large constraints defined in extension, where all combinations of values are enumerated. The BigPC and especially the Renault models are declaring particularly large constraints, which may have an impact on the maintenance and the runtime performance of the model.

11.2 Results

In this section, we detail the results from our benchmark with the models presented in the previous section. We first discuss our experience in the modelling of these models in ProCoLa, and then detail results related to the framework runtime. All measurements have been taken on a Macbook Pro running Windows 7 with a 2.4GHz Core 2 Duo and 3GB main memory.

11.2.1 Modelling

The benchmark test cases are originally presented using different modelling languages, e.g., PCML or the PM language. We thus attempted to model them using the ProCoLa modelling language defined in this dissertation.

The translation of each model has been done with relative ease, in the sense that

[1] In general, IP issues relative to models from industrial products have been difficult to handle throughout this project, as companies were often reluctant to share information about how their products are modelled.

all constructs from these other modelling languages were successfully mapped to ProCoLa constructs, except for default values, which are yet not available in ProCoLa. As stated before, only our extended Mobile Device case study explicitly offered different views of the product family. However, although all the models are easily mapped to a single view, some of them contain elements that could belong to different views, from a conceptual point of view. For example, the *Bike* model defines some attributes related to the physical design of the system (e.g., the frame size), while others are related to its functionalities (e.g., the bike type), and could be defined in the feature view, linked with implementation constraints such as

$$(frame.biketype = \text{“}RacerBike\text{”} or frame.biketype = \text{“}CityBike\text{”}) \rightarrow \\ ((frame.size >= 15 and frame.size <= 20))$$

It is also worth noting that some models do not include dynamic partonomy, maybe due to the limitations of the language they are defined in. For example, in the *PC* model, there needs to be at least one block of Random Access Memory (RAM), and up to four. This could easily be handled if the *ramBlock* partonomic relation between the *PC* and the *RamBlock* types had a dynamic cardinality of 1 to 4. However, the model is defined with a fixed cardinality of 4 for this relation, and the attribute *id* in the *RamBlock* type can be of value “No RAM block”. Besides requiring additional constraints in the *RamBlock* type to ensure the correctness of the model, this is not consistent with the conceptual semantics of a type, as a RamBlock instance could be present in the model without actually being needed for the real product.

Finally, many of the constraints in these models are defined in extension, as a disjunction of conjunctions of value assignments for attributes. These constraints can be written in ProCoLa using table constraints. From a modelling point of view, this has several advantages: repeating the variables in assignments may introduce unwanted mistakes, and transforming these constraints into tables reduces the clutter in the model. Also, for models with large constraints like *BigPC* or the *Renault* model, transferring these constraints to other tools like spreadsheet applications and databases may also improve their maintainability.

We provide some measurements related to modelling in Table 11.3. We recorded the time spent in seconds by our prototype for compiling ProCoLa models, for generating the analysis graph and then for applying the two full model analyses, in order to verify that there are no cycles and unused elements in the models. The measurements show that the compilation and analysis of the configuration model is done almost instantaneously, apart for the Renault model, which requires more time for compilation due to the very large size of the model.

Table 11.3: Experimental results (modelling)

Model	Compilation time	Analysis Graph generation	Full model analyses	
			Cycles	Unused elts
Mobile Device	1.08	0.25	0.24	0.14
ESVS	0.56	0.08	0.03	0.02
FS	0.57	0.07	0.02	0.02
FX	0.56	0.07	0.03	0.02
Machine	0.56	0.07	0.03	0.02
Bike	0.61	0.08	0.03	0.02
PC	0.68	0.08	0.03	0.02
BigPC	0.59	0.09	0.04	0.03
Renault	12.02	0.08	0.04	0.03

11.2.2 Runtime

In order to assess the performance of our prototype implementation at runtime, we performed some measures on the following criteria: the time in seconds spent for creating a configurable instance from each model; the time in seconds for generating the variables and constraints for the corresponding CondCSP; the time spent in seconds for the first consistency check done before the configuration starts; and finally the average time for consistency checks during configuration. In order to get that last value, we took each configuration instance, then for a 100 times, chose randomly between:

1. picking a random unassigned variable and assigning a random value among its feasible ones.
2. removing a previous assignment.

The results of the experiments can be seen in Table 11.4, with two versions of the PC, BigPC and Renault models: the original version and another one where extensive constraints have been replaced by table constraints. One of the first interesting point in this table is that the creation of the configuration instance and the generation of the CondCSP variables and constraints is done very quickly. When looking at the model consistency, most of the models besides BigPC, Renault and the Mobile Device are in average checked in a very short time ($\leq$.25s). The Mobile Device model takes longer on average, due to its high number of types and constraints, which is expected. Because of its poor performance, the only a subset of constraints were considered in the non-table version of the Renault model.

Table 11.4: Experimental results (runtime)

Model	Instance creation	CondCSP model generation	Initial consistency	Average consistency
Mobile Device	0.11	0.19	84.12	4.66
ESVS	0.08	0.08	0.29	0.11
FS	0.08	0.08	0.20	0.07
FX	0.07	0.06	0.11	0.03
Machine	0.07	0.08	0.03	0.01
Bike	0.08	0.06	0.28	0.02
PC	0.08	0.09	13.2	0.25
PC (tables)	0.09	0.18	9.5	0.01
BigPC	0.08	0.13	430.2	20.93
BigPC (tables)	0.08	0.10	0.51	0.12
Renault	0.08	0.5	894.23	39.84
Renault (tables)	0.07	1.69	2.84	0.18

What is very interesting is to see the significant difference in performance for the BigPC and Renault models, depending on whether the constraints defined in extension are declared with predicates (disjunctions and conjunctions) or as table constraints. Indeed, large constraints defined as predicates involving many references to variables do not exploit the constraints' special structure like table constraints do. When table constraints are used, our specialised algorithm *DnSTR* (see Chapter 8) can be used instead of the AC3-like algorithm used for non-table constraints. The gain in performance gets even more important when the constraints arity is higher. As a comparison, it is worth noting that the average time for consistency checks with the Renault model is on par with results by [5]. However, this comparison needs to be taken with a pinch of salt, as on the one hand the work from Amilhastre et al. requires a 2 hour-long compilation of the model into an automaton, but results in checks with a higher level of consistency than the arc consistency considered here.

Chapter 12

Conclusion

Product configuration has been an active research topic for some time now. Configuration models can be quite complex and large, and therefore, efficient methods for managing, modelling and reasoning about those models are needed. Several configuration-specific languages for modelling in configuration have been studied, but very few have been rigorously defined with precise semantics.

We started with the objective in thesis to analyse modelling and constraint-based solving methods for product configuration, and design and implement a framework for creating and configuring modern product models. In this dissertation, we studied several well-known modelling languages in the context of configuration and then presented the design and implementation of a new configuration framework for heterogeneous products composed of physical components, software and services. The aim of this framework is to synthesize, unify and extend existing approaches in the domain of product configuration, software variability and service configuration. We clearly defined the framework, first conceptually then formally, and designed a configuration-specific modelling language with clear semantics, along with several analyses for checking the correctness of configuration models. Our configuration framework is supported by a constraint system, centered around a novel algorithm for dealing with often-used table constraints. The feasibility of our approach is proven by the implementation of a practical prototype for our work.

In this chapter, we give an outlook on possible future development and detail our contributions before concluding.

12.1 Further Work

The work presented in this dissertation can be extended in various ways. Additional views could be added to our framework in order to consider other aspects of the product family configuration, like views describing what the environment of the product family may provide (or request), in a similar way as in the object-of-services world from [49]. Adding support for modelling the evolution of product models in time is also a topic for future work, whether it is on the conceptual point of view or for solving during configuration. Some work has already been done on this topic [55].

The formal analysis of configuration models defined in ProCoLa could be extended. More insight on the model's status could be derived, e.g., establishing which elements do not participate in the implementation of a higher level view, thus requiring direct configuration. Specific model metrics could also be used to improve the structure of the model, such as presenting potential taxonomic relations that could be added to improve the model. However, finding those metrics may not be an easy task, and would require more research.

Improving the constraint system is also a topic of future work. Several researchers [95,112] have considered Generative Constraint Satisfaction Problems instead of Conditional CSP to solve configuration problems (dicussed in Sections 7.2.1 and 9.3). An interesting idea would be to compare the performance of both alternatives, or even to perform an analysis of the model and determine what framework would be the most appropriate to use, improving execution times when possible. Our solving algorithms would then need to be adapted to handle GCSPs, in particular when dealing with table constraints. Other potential improvements of the constraint system include the support for soft constraints, which could be used to model suggestions and default values.

Our prototype implementation could benefit from some improvements, for example importing UML models into the framework. Integration with SysML may be considered as well. Additional features could be added to the runtime support of the prototype implementation, such as the support for explanations when a conflict occurs in the configuration.

Last but not least, demonstrating the practical applicability of our configuration framework requires modelling real-life systems in industrial contexts. Both the

expressiveness and the usability of the ProCoLa language would benefit from strong empirical evaluation.

12.2 Contributions and Concluding Remarks

In this dissertation, we have presented a new framework for modelling and solving configuration in heterogeneous product families that consist of several dimensions, e.g., physical, software, and services.

We first drew a list of requirements for industrial configurators [75, 79], based on literature and previous experience, and used it to discuss and evaluate the potential for configuration of several general modelling languages such as the Unified Modelling Language (UML), the System Modelling Language (SysML), and the EXPRESS language from STEP, the International Standard ISO 10303.

We detailed a complete conceptual framework based on the concept of modelling views [77] that differentiate the different aspects of a heterogeneous product family. This framework has been motivated by four research questions, and illustrated by the example of a Mobile Device product family. We then described the design of ProCoLa, a configuration-specific modelling language to support our conceptual work. ProCoLa has been given clear semantics using a translation to a formalism of our conceptual framework. We defined a type system and well-formedness rules for configuration models in ProCoLa, and presented several model analyses based on a graph created from configuration models.

In order to support the configuration of ProCoLa models, we designed a novel algorithm DnSTR [78] for maintaining arc consistency in table constraints for Dynamic Constraint Satisfaction Problems. This algorithm is based on Simple Tabular Reduction and allows the dynamic addition and removal of constraints at runtime. We then implemented a whole constraint system for solving the configuration of ProCoLa models by translating model instances to Conditional Constraint Satisfaction Problems, including when dealing with table constraints.

We provided a practical prototype of our configuration framework. We implemented a language service to extend the Visual Studio development environment to support ProCoLa models and provide tool support, including real-time well-formedness checks, outlining and auto completion, as well as export of ProCoLa into UML models using a predefined UML profile. We also provided connectors for handling table constraints using spreadsheet applications and databases, supporting currently the Excel tool and Microsoft SQL Server. We implemented a runtime system for configuring ProCoLa models, as well as a visual tool for

debugging and modifying models directly at runtime, supported by our own constraint solver. Finally, we performed an evaluation of our prototype implementation using an extended version of our motivating case study and multiple benchmark models used in literature.

In conclusion, this dissertation contributes to the theory and practice of product configuration by introducing a new integrated framework supported by conceptual and formal work, as well as practical implementation. We believe this framework can serve as a common basis for collaborative configuration knowledge base design and maintenance, as it takes into account the multiple views necessary to model complex products with multiple dimensions.

Appendix A

Case Study in EXPRESS

This Appendix contains the source code of the Mobile Device model from Part I in EXPRESS.

```
(* Main schema *)
SCHEMA DeviceModel;

(* User-defined functions *)
FUNCTION SumChips(mb:Motherboard): INTEGER;
  LOCAL
    result: INTEGER := mb.nbTouchChips;
  END_LOCAL;
  IF (mb.hasAccelerometer)
          result := result + 1;
  END_IF;
  IF (mb.hasCompassChip)
          result := result + 1;
  END_IF;
  RETURN(result);
END_FUNCTION;

(* User-defined types *)
TYPE MHZ = INTEGER;
WHERE SELF >= 0; END_TYPE;
TYPE INCH = INTEGER;
```

```
WHERE SELF >= 0; END_TYPE;
TYPE GB = INTEGER;
WHERE SELF >= 0; END_TYPE;
TYPE SSD_TYPE = ENUMERATION OF (flash,DRAM); END_TYPE;
TYPE MINIUSBADAPTER_TYPE = ENUMERATION OF (none,VGA,DVI,HDMI
    ); END_TYPE;
TYPE RFCARD_TYPE = ENUMERATION OF (GSM,GSM_UMTS); END_TYPE;
TYPE CHIP_TYPE = ENUMERATION OF (Wifi,Bluetooth,FM);
    END_TYPE;
TYPE PORT_TYPE = ENUMERATION OF (none,VGA,DVI,HDMI,miniUSB);
     END_TYPE;
TYPE ENCLOSURE_SIZE = ENUMERATION OF (115x58,240x190,268x178
    ,295x210); END_TYPE;
TYPE STYLUS_SIZE = ENUMERATION OF (compact,large); END_TYPE;
TYPE TOUCHSCREEN_TYPE = ENUMERATION OF (resistive,capacitive
    ); END_TYPE;
TYPE SCREEN_TECHNOLOGY = ENUMERATION OF (OLED,LCD); END_TYPE
    ;

(* Root component *)
ENTITY Device;
  motherboard: Motherboard;
  screen: Screen;
  drive: StorageDrive;
  enclosure: Enclosure;
  miniUSBAdapter: MINIUSBADAPTER_TYPE;
WHERE
  ((motherboard.nbSlots = 6) AND (enclosure.size = 115x58)
      AND (screen.size = 3)) OR
  ((motherboard.nbSlots = 10) AND (enclosure.size = 240x190)
       AND (screen.size = 9)) OR
  ((motherboard.nbSlots = 12) AND (enclosure.size = 266x178)
       AND (screen.size = 10)) OR
  ((motherboard.nbSlots = 14) AND (enclosure.size = 295x210)
       AND (screen.size = 12));
  (TYPEOF(screen) = TouchScreen) = (motherboard.nbTouchChips
       > 0);
  (SIZEOF(motherboard.ethernetCard) = 1) = enclosure.
      ethernetPort;
  (enclosure.port <> miniUSB) -> miniUSBAdapter = none;
END_ENTITY;

(* Abstract base component for storage drives *)
ENTITY StorageDrive ABSTRACT SUPERTYPE;
  capacity: GB;
END_ENTITY;
```

```
ENTITY SSD SUBTYPE OF (StorageDrive);
WHERE
  (capacity = 16) OR (capacity = 32) OR (capacity = 64);
END_ENTITY;

ENTITY HDD SUBTYPE OF (StorageDrive);
WHERE
  (capacity = 128) OR (capacity = 256) OR (capacity = 320)
      OR (capacity = 500);
END_ENTITY;

ENTITY Motherboard;
  rfCard: SET[0:1] OF RFCard;
  processor: Processor;
  gps: SET[0:1] OF GPSReceiver;
  wirelessChip: SET[0:3] OF WirelessChip;
  ethernet: EthernetCard;
  nbTouchChips: INTEGER;
  nbSlots: INTEGER;
  hasCompassChip: BOOLEAN;
  hasAccelerometer: BOOLEAN;
DERIVE
  nbChips : INTEGER:= SumChips(SELF);
WHERE
  0 <= nbTouchChips <= 3;
END_ENTITY;

ENTITY RFCard;
  type: RFCARD_TYPE;
END_ENTITY;

ENTITY Processor;
  frequency: MHZ;
WHERE
  (frequency = 600) OR (frequency = 1000) OR (frequency =
      1600);
END_ENTITY;

ENTITY GPSReceiver; END_ENTITY;

ENTITY WirelessChip;
  chipType: CHIP_TYPE;
END_ENTITY;

ENTITY EthernetCard; END_ENTITY;

ENTITY Screen SUPERTYPE;
```

```
  ppi: INTEGER;
  size: INCH;
  technology: SCREEN_TECHNOLOGY;
WHERE
  (ppi = 132) OR (ppi = 330);
  (size = 3) OR (size = 9) OR (size = 10) OR (size = 12);
END_ENTITY;

ENTITY TouchScreen SUBTYPE OF (Screen);
  type: TOUCHSCREEN_TYPE;
  oleophobicCoating: BOOLEAN;
INVERSE
  associatedStylus: SET[0:1] OF Stylus FOR
      associatedTouchScreen;
WHERE
  (type = resistive) = (SIZEOF(associatedStylus) = 1);
END_ENTITY;

ENTITY Enclosure;
  port: RFCARD_TYPE;
  DVDSlot: BOOLEAN;
  ethernetPort: BOOLEAN;
  hasStylusSlot: BOOLEAN;
  size: ENCLOSURE_SIZE;
  stylus: SET[0:1] OF Stylus;
WHERE
  (size = 115x58) = (stylus[1].size = compact);
  (size = 115x58) = (port = miniUSB);
  (SIZEOF(stylus) = 1) = (hasStylusPort);
END_ENTITY;

ENTITY Stylus;
  size: STYLUS_SIZE;
  associatedTouchScreen: TouchScreen;
END_ENTITY;

END_SCHEMA;
```

Appendix B

Formalisation of Structure and Realisation Views

In this Appendix, we present the formalisation rules of ProCoLa for structure and realisation views.

In order to interpret units and constants, we use two auxiliary mappings defined as follow:

$$
\begin{array}{ll}
u \in \mathit{Units}_{\mathrm{S}} & ::= Id_u \longmapsto D_u \text{ with } Id_u \in \mathbb{S}, D_u \in \mathcal{P}(\mathbb{N} \cup \mathbb{B} \cup \mathbb{S}) \\
c \in \mathit{Const}_{\mathrm{S}} & ::= Id_c \longmapsto \langle c, Id_u \rangle \text{ with } Id_c \in \mathbb{S}, c \in \mathbb{N} \cup \mathbb{B} \cup \mathbb{S}, Id_u \in \mathbb{S} \cup \{\epsilon\}
\end{array}
$$

The units mapping $\mathit{Units}_{\mathrm{S}}$ maps a string, the unit name Id_u, to a set of constant values, the domain of the unit. The constants mapping $\mathit{Const}_{\mathrm{S}}$ maps a string, the constant name Id_c, to a pair composed by the constant value c and a potential unit name Id_u (ϵ is used when no specific unit is assigned to the constant).

Tables B.1, B.2, and B.3 show the formalisation rules for a structure view S, while Table B.4 gives the formalisation rules for a realisation view R;

$$[C_1] \quad : \quad \texttt{c} \rightsquigarrow_{\mathrm{S}} \langle c, \epsilon \rangle$$

$$[C_2] \quad : \quad Id_c \rightsquigarrow_{\mathrm{S}} \langle c, u \rangle \qquad \text{if } (Id_c \longmapsto \langle c, u \rangle) \in Const_{\mathrm{S}}$$

$$[Type_1] \quad : \quad Id_u \rightsquigarrow_{\mathrm{S}} \langle D_u, Id_u \rangle \qquad \text{if } (Id_u \longmapsto D_u) \in Units_{\mathrm{S}}$$

$$[Type_{2-4}] \quad : \quad \texttt{integer} \rightsquigarrow_{\mathrm{S}} \langle \mathbb{N}, \epsilon \rangle \quad \texttt{boolean} \rightsquigarrow_{\mathrm{S}} \langle \mathbb{B}, \epsilon \rangle \quad \texttt{enum} \rightsquigarrow_{\mathrm{S}} \langle \mathbb{S}, \epsilon \rangle$$

$$[DomReduc_1] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{S}} \langle c_i, u \rangle}{\{C_1, ..., C_n\} \rightsquigarrow_{\mathrm{S}} \langle D = \{c_1, ..., c_n\}, u \rangle}$$

$$[DomReduc_2] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{S}} \langle c_i, u \rangle}{\texttt{[}C_1\texttt{..}C_n\texttt{]} \rightsquigarrow_{\mathrm{S}} \langle D = \{n / c_1 \leq n \leq c_n\}, u \rangle}$$

$$[DomReduc_3] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{S}} \langle c, u \rangle}{[\texttt{-inf..}C] \rightsquigarrow_{\mathrm{S}} \langle D = \{n / n \leq c\}, u \rangle}$$

$$[DomReduc_4] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{S}} \langle c, u \rangle}{[C\texttt{..inf}] \rightsquigarrow_{\mathrm{S}} \langle D = \{n / c \leq n\}, u \rangle}$$

$$[TypeDecl] \quad : \quad \frac{Type \rightsquigarrow_{\mathrm{S}} \langle D_1, u \rangle;\ DomReduc \rightsquigarrow_{\mathrm{F}} \langle D_2, u \rangle}{TypeDomReduc \rightsquigarrow_{\mathrm{S}} \langle D_2, u \rangle} \qquad \text{if } D_2 \subseteq D_1$$

$$[ConstrDecl_1] \quad : \quad \frac{CSymExp \rightsquigarrow_{\mathrm{S}} e^c}{\{CSymExp\} \rightsquigarrow_{\mathrm{S}} \langle \mathrm{T}_c, e^c \rangle}$$

$$[ConstrDecl_2] \quad : \quad \frac{ConstrVal \rightsquigarrow_{\mathrm{S}} e^c}{[Id]\texttt{:[description:[c;]]}\ ConstrVal \rightsquigarrow_{\mathrm{S}} \langle \mathrm{T}_c, e^c \rangle}$$

$$[ResourceUse] \quad : \quad \frac{CSymExp \rightsquigarrow_{\mathrm{S}} e^c}{\{Id_r\texttt{:=}CSymExp\} \rightsquigarrow_{\mathrm{S}} \langle \mathrm{T}_r, Id_r, \epsilon, e^c \rangle} \qquad \text{if } Id_r \in \mathcal{R}_{\mathrm{S}}$$

$$[Direction] \quad : \quad \texttt{provides} \rightsquigarrow_{\mathrm{S}} \textsc{Provides} \quad \texttt{requires} \rightsquigarrow_{\mathrm{S}} \textsc{Requires} \quad \epsilon \rightsquigarrow_{\mathrm{S}} \textsc{None}$$

$$[AssocDecl_1] \quad : \quad \frac{Direction \rightsquigarrow_{\mathrm{S}} d}{Direction\ Id_a\texttt{:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \rho = \langle Id_a, \mathrm{T}_a, \mathrm{T}_t, d, 1, 1 \rangle} \qquad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^a$$

$$[AssocDecl_2] \quad : \quad \frac{Direction \rightsquigarrow_{\mathrm{S}} d;\ C \rightsquigarrow_{\mathrm{S}} \langle r, \epsilon \rangle}{Direction\ Id_a\texttt{[}C\texttt{]:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \rho = \langle Id_a, \mathrm{T}_a, \mathrm{T}_t, d, r, r \rangle} \text{ if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^a$$

$$[AssocDecl_3] \quad : \quad \frac{Direction \rightsquigarrow_{\mathrm{S}} d;\ C_i \rightsquigarrow_{\mathrm{S}} \langle r_i, \epsilon \rangle \ \text{ if} \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^a}{Direction\ Id_a\texttt{[}C_1\texttt{..}C_2\texttt{]:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \rho = \langle Id_a, \mathrm{T}_a, \mathrm{T}_t, d, r_1, r_2 \rangle}$$

Table B.1: Formalisation of ProCoLa structure view

$$[Visibility] \quad : \quad \texttt{readonly} \rightsquigarrow_{\mathrm{S}} \textsc{Readonly} \quad \texttt{hidden} \rightsquigarrow_{\mathrm{S}} \textsc{Hidden} \quad \epsilon \rightsquigarrow_{\mathrm{S}} \textsc{Visible}$$

$$[AttrDecl_1] \quad : \quad \frac{Visibility \rightsquigarrow_{\mathrm{S}} v;\ TypeDecl \rightsquigarrow_{\mathrm{S}} \langle D, u\rangle}{Visibility\ Id_a\texttt{:}TypeDecl \ldots \rightsquigarrow_{\mathrm{S}} \alpha = \langle Id_a, \mathrm{T}_a, D, v, 1\rangle}$$

$$[AttrDecl_2] \quad : \quad \frac{Visibility \rightsquigarrow_{\mathrm{S}} v;\ TypeDecl \rightsquigarrow_{\mathrm{S}} \langle D, u\rangle;\ C \rightsquigarrow_{\mathrm{S}} \langle r, \epsilon\rangle}{Visibility\ Id_a\texttt{[}C\texttt{]:}TypeDecl \ldots \rightsquigarrow_{\mathrm{S}} \alpha = \langle Id_a, \mathrm{T}_a, D, v, r\rangle}$$

$$[SubPDecl_1] \quad : \quad Id_s\texttt{:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, 1, 1\rangle \qquad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^c$$

$$[SubPDecl_2] \quad : \quad \frac{C \rightsquigarrow_{\mathrm{S}} \langle r, \epsilon\rangle}{Id_s\texttt{[}C\texttt{]:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, r, r\rangle} \quad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^c$$

$$[SubPDecl_3] \quad : \quad \frac{C_i \rightsquigarrow_{\mathrm{S}} \langle r_i, \epsilon\rangle}{Id_s\texttt{[}C_1\texttt{..}C_2\texttt{]:}\mathrm{T}_t \ldots \rightsquigarrow_{\mathrm{S}} \sigma = \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, r_1, r_2\rangle} \text{ if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^c$$

$$[CTypeDecl_1] \quad : \quad \ldots\texttt{componentType T}\ldots \rightsquigarrow_{\mathrm{S}} \mathrm{T} \in \mathcal{T}_{\mathrm{S}}^c$$

$$[CTypeDecl_2] \quad : \quad \texttt{[Root]}\ldots\texttt{componentType T}\ldots \rightsquigarrow_{\mathrm{S}} \mathrm{T}_{\mathrm{S}}^{root} = \mathrm{T}$$

$$[CTypeDecl_3] \quad : \quad \ldots\texttt{abstract}\ldots\texttt{componentType T}\ldots \rightsquigarrow_{\mathrm{S}} \mathrm{T} \in \mathcal{T}_{\mathrm{S}}^{Ab}$$

$$[CTypeDecl_4] \quad : \quad \ldots\texttt{componentType T subtypeOf } \mathtt{T}_1, \ldots, \mathtt{T}_n\ldots \rightsquigarrow_{\mathrm{S}} (\mathrm{T}_1, \mathrm{T}), \ldots, (\mathrm{T}_n, \mathrm{T}) \in I_{\mathrm{S}} \qquad \text{if } \mathrm{T}_t \in \mathcal{T}_{\mathrm{S}}^c$$

$$[CTypeDecl_5] \quad : \quad \frac{A_i \rightsquigarrow_{\mathrm{S}} \alpha_i = \langle \ldots, \mathrm{T}_{ai}, \ldots\rangle}{\ldots\texttt{componentType T}\ldots\texttt{attributes} : \mathtt{A}_1, \ldots, \mathtt{A}_n\ldots \rightsquigarrow_{\mathrm{S}} \alpha_i = \langle \ldots, \mathrm{T}, \ldots\rangle \in \mathcal{D}_{\mathrm{S}}}$$

$$[CTypeDecl_6] \quad : \quad \frac{S_i \rightsquigarrow_{\mathrm{S}} \sigma_i = \langle \ldots, \mathrm{T}_{si}, \mathrm{T}_t, \ldots\rangle}{\ldots\texttt{componentType T}\ldots\texttt{subParts} : \mathtt{S}_1, \ldots, \mathtt{S}_n\ldots \rightsquigarrow_{\mathrm{S}} \sigma_i = \langle \ldots, \mathrm{T}, \mathrm{T}_t, \ldots\rangle \in \mathcal{S}_{\mathrm{S}}}$$

$$[CTypeDecl_7] \quad : \quad \frac{As_i \rightsquigarrow_{\mathrm{S}} As_i = \langle \ldots, , \mathrm{T}_{ai}, \mathrm{T}_t, \ldots\rangle}{\ldots\texttt{componentType T}\ldots\texttt{associations} : \mathtt{As}_1, \ldots, \mathtt{As}_n\ldots \rightsquigarrow_{\mathrm{S}} As_i = \langle \ldots, \mathrm{T}, \mathrm{T}_t, \ldots\rangle \in \mathcal{A}_{\mathrm{S}}}$$

$$[CTypeDecl_7] \quad : \quad \frac{U_i \rightsquigarrow_{\mathrm{S}} U_i = \langle \ldots, \mathrm{T}_{ri}, \ldots\rangle}{\ldots\texttt{componentType T}\ldots\texttt{produces} : \mathtt{U}_1, \ldots, \mathtt{U}_n\ldots \rightsquigarrow_{\mathrm{S}} U_i = \langle \mathrm{T}, \ldots, \textsc{Produces}, \ldots\rangle \in \mathcal{U}_{\mathrm{S}}}$$

Table B.2: Formalisation of ProCoLa structure view (continued)

$$[CTypeDecl_8] \quad : \quad \frac{U_i \rightsquigarrow_{\mathrm{S}} U_i = \langle ..., \mathrm{T}_{ri}, ... \rangle}{...\texttt{componentType T...consumes : U}_1, ..., \texttt{U}_\texttt{n}... \rightsquigarrow_{\mathrm{S}} U_i = \langle \mathrm{T}, ..., \textsc{Consumes}, ... \rangle \in \mathcal{U}_{\mathrm{S}}}$$

$$[CTypeDecl_9] \quad : \quad \frac{Cons_i \rightsquigarrow_{\mathrm{S}} C_i = \langle ..., \mathrm{T}_{ci}, ... \rangle}{...\texttt{componentType T...constraints : Cons}^{\texttt{c}}_1, ..., \texttt{Cons}^{\texttt{c}}_\texttt{n}... \rightsquigarrow_{\mathrm{S}} C_i = \langle ..., \mathrm{T}, ... \rangle \in \mathcal{C}^c_{\mathrm{S}}}$$

$$[ATypeDecl_1] \quad : \quad ...\texttt{associationType T...} \rightsquigarrow_{\mathrm{S}} \mathrm{T} \in \mathcal{T}^a_{\mathrm{S}}$$

$$[ATypeDecl_2] \quad : \quad ...\texttt{abstract...associationType T...} \rightsquigarrow_{\mathrm{S}} \mathrm{T} \in \mathcal{T}^{Ab}_{\mathrm{S}}$$

$$[ATypeDecl_3] \quad : \quad ...\texttt{associationType T subtypeOf T}_1, ..., \texttt{T}_\texttt{n}... \rightsquigarrow_{\mathrm{S}} (\mathrm{T}_1, \mathrm{T}), ..., (\mathrm{T}_n, \mathrm{T}) \in I_{\mathrm{S}} \qquad \text{if } \mathrm{T}_t \in \mathcal{T}^a_{\mathrm{S}}$$

$$[ATypeDecl_4] \quad : \quad \frac{A_i \rightsquigarrow_{\mathrm{S}} \alpha_i = \langle ..., \mathrm{T}_{ai}, ... \rangle}{...\texttt{associationType T...attributes : A}_1, ..., \texttt{A}_\texttt{n}... \rightsquigarrow_{\mathrm{S}} \alpha_i = \langle ..., \mathrm{T}, ... \rangle \in \mathcal{D}_{\mathrm{S}}}$$

$$[ATypeDecl_5] \quad : \quad \frac{Cons_i \rightsquigarrow_{\mathrm{S}} C_i = \langle ..., \mathrm{T}_{ci}, ... \rangle}{...\texttt{associationType T...constraints : Cons}^{\texttt{c}}_1, ..., \texttt{Cons}^{\texttt{c}}_\texttt{n}... \rightsquigarrow_{\mathrm{S}} C_i = \langle ..., \mathrm{T}, ... \rangle \in \mathcal{C}^c_{\mathrm{S}}}$$

$$[ConstDecl_1] \quad : \quad Id_c\texttt{:= c} \rightsquigarrow_{\mathrm{S}} (Id_c \longmapsto \langle c, \epsilon \rangle) \in Const_{\mathrm{S}}$$

$$[ConstDecl_2] \quad : \quad Id_c\texttt{:}Id_u\texttt{:= c} \rightsquigarrow_{\mathrm{S}} (Id_c \longmapsto \langle c, Id_u \rangle) \in Const_{\mathrm{S}}$$

$$[UnitDecl] \quad : \quad \frac{TypeDecl \rightsquigarrow_{\mathrm{S}} \langle D_u, \epsilon \rangle}{Id_u\texttt{:}TypeDecl \rightsquigarrow_{\mathrm{S}} (Id_u \longmapsto D_u) \in Units_{\mathrm{S}}}$$

$$[ResourceDecl_1] \quad : \quad Id_r\texttt{: integer} \rightsquigarrow_{\mathrm{S}} Id_r \in \mathcal{R}_{\mathrm{S}}$$

$$[ResourceDecl_2] \quad : \quad Id_r\texttt{:}Id_u\texttt{:= c} \rightsquigarrow_{\mathrm{S}} Id_r \in \mathcal{R}_{\mathrm{S}}$$

$$[StructureView] \quad : \quad \texttt{dimension } Dimension;\ StructureDecl \rightsquigarrow_{\mathrm{S}} \mathrm{S} \in \mathcal{V}^s$$

Table B.3: Formalisation of ProCoLa structure view (end)

$$[UseDecl] \quad : \quad Id_u\texttt{:T}_r \rightsquigarrow_{\mathrm{R}} \langle Id_u, \mathrm{T}_u, \mathrm{T}_r \rangle \quad \text{if } \mathrm{T}_r \in \mathcal{T}^r_{\mathrm{R}}$$

$$[SubEltDecl_1] \quad : \quad Id_s\texttt{:T}_t \rightsquigarrow_{\mathrm{R}} \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, 1 \rangle$$

$$[SubEltDecl_2] \quad : \quad Id_s\texttt{[}c\texttt{]:T}_t \rightsquigarrow_{\mathrm{R}} \langle Id_s, \mathrm{T}_s, \mathrm{T}_t, c \rangle$$

$$[MappingDecl] \quad : \quad \frac{CSymExp \rightsquigarrow_{\mathrm{S}} e^c}{\texttt{T}_s\texttt{:}\{CSymExp\} \rightsquigarrow_{\mathrm{R}} \langle \mathrm{T}_s, \mathrm{T}_m, e^c \rangle} \quad \text{if } \mathrm{T}_s \in \mathcal{T}_{\mathrm{S}}$$

$$[RTypeDecl] \quad : \quad \texttt{resourceType T}_r \rightsquigarrow_{\mathrm{R}} \mathrm{T}_r \in \mathcal{T}^r_{\mathrm{R}}$$

$$[OTypeDecl_1] \quad : \quad \texttt{...operationType T...} \rightsquigarrow_{\mathrm{R}} \mathrm{T} \in \mathcal{T}^o_{\mathrm{R}}$$

$$[OTypeDecl_2] \quad : \quad \frac{M_i \rightsquigarrow_{\mathrm{R}} m_i = \langle \mathrm{T}_{si}, \mathrm{T}_{mi}, e^c \rangle}{\texttt{...operationType T...mapping : M}_1\texttt{, ..., M}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} m_i = \langle \mathrm{T}_{si}, \mathrm{T}, e^c \rangle \in \mathcal{M}_{\mathrm{R}}}$$

$$[OTypeDecl_3] \quad : \quad \texttt{...operationType T...successors : T}_1\texttt{, ..., T}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} \langle \mathrm{T}, \mathrm{T}_i \rangle \in \mathcal{N}_{\mathrm{R}} \quad \text{if } \mathrm{T}_i \in \mathcal{T}^o_{\mathrm{R}}$$

$$[OTypeDecl_4] \quad : \quad \frac{U_i \rightsquigarrow_{\mathrm{R}} \omega_i = \langle Id_{ui}, \mathrm{T}_{ui}, \mathrm{T}_{ri} \rangle}{\texttt{...operationType T...uses : U}_1\texttt{, ..., U}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} \omega_i = \langle Id_{ui}, \mathrm{T}, \mathrm{T}_{ri} \rangle \in \Omega_{\mathrm{R}}}$$

$$[OTypeDecl_5] \quad : \quad \frac{S_i \rightsquigarrow_{\mathrm{R}} \sigma_i = \langle ..., \mathrm{T}_{si}, ... \rangle}{\texttt{...operationType T...subOperations : S}_1\texttt{, ..., S}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} \sigma_i = \langle \langle ..., \mathrm{T}, ... \rangle \in \mathcal{S}^o_{\mathrm{R}}}$$

$$[ITypeDecl_1] \quad : \quad \texttt{...itemType T...} \rightsquigarrow_{\mathrm{R}} \mathrm{T} \in \mathcal{T}^i_{\mathrm{R}}$$

$$[ITypeDecl_2] \quad : \quad \frac{M_i \rightsquigarrow_{\mathrm{R}} m_i = \langle \mathrm{T}_{si}, \mathrm{T}_{mi}, e^c \rangle}{\texttt{...itemType T...mapping : M}_1\texttt{, ..., M}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} m_i = \langle \mathrm{T}_{si}, \mathrm{T}, e^c \rangle \in \mathcal{M}_{\mathrm{R}}}$$

$$[ITypeDecl_3] \quad : \quad \frac{S_i \rightsquigarrow_{\mathrm{R}} \sigma_i = \langle ..., \mathrm{T}_{si}, ... \rangle}{\texttt{...itemType T...subItems : S}_1\texttt{, ..., S}_\texttt{n}\texttt{...} \rightsquigarrow_{\mathrm{R}} \sigma_i = \langle ..., \mathrm{T}, ... \rangle \in \mathcal{S}^i_{\mathrm{R}}}$$

$$[StructureView] \quad : \quad ResourceView \rightsquigarrow_{\mathrm{R}} \mathrm{R} \in \mathcal{V}^r$$

Table B.4: Formalisation of ProCoLa realisation view

Bibliography

[1] Mohd Syazwan Abdullah, Andy Evans, Ian Benest, and Chris Kimble. Developing a uml profile for modelling knowledge-based systems. pages 202–216, 2004. jf.

[2] Mohd Syazwan Abdullah, Richard Paige, Chris Kimble, and Ian Benest. A uml profile for knowledge-based systems modelling. pages 871–878, 2007.

[3] H. Akkermans, Z. Baida, J. Gordijn, N. Peña, A. Altuna, and I. Laresgoiti. Value webs: Using ontologies to bundle real-world services. *IEEE Intelligent Systems*, 19(4):57–66, 2004.

[4] M. Aldanondo, K. Hadj-Hamou, G. Moynard, and J. Lamothe. Mass customization and configuration: Requirement analysis and constraint based modeling propositions. *Integr. Comput.-Aided Eng.*, 10(2):177–189, 2003.

[5] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic csps—-application to configuration. *AI*, 135(1-2):199–234, 2002.

[6] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.

[7] Timo Asikainen, Tomi Mannisto, and Timo Soininen. A unified conceptual foundation for feature modelling. pages 31–40, 2006.

[8] Timo Asikainen, Timo Soininen, and Tomi Mannisto. A koala-based approach for modelling and deploying configurable software product families. *Proc. International Workshop on Product Family Engineering*, pages 225–249, Jan 2003.

[9] F. Bacchus, X. Chen, P. Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140:1–37, 2002.

[10] V. Barker, D. 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.

[11] R. Bartak and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. *Proc. FLAIRS'05*, pages 161–166, 2005.

[12] D. Batory. Feature models, grammars, and propositional formulas. *SPLC 2005, LNCS*, 3714:7–20, 2005.

[13] M. Becker. Towards a general model of variability in product families. In *Proc. of the First Workshop on Software Variability Management*, 2003.

[14] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. *Proc. AAAI'91*, pages 221–226, 1991.

[15] C. Bessière. Arc-consistency for non-binary csps. *Proc. ECAI'92*, pages 23–27, 1992.

[16] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[17] C. Bessière, E. Hebrard, B Hnich, and T. Walsh. The complexity of reasoning with global constraints. *Constraints*, 12(2):239–259, 2007.

[18] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

[19] T. Böhmann, M. Junginger, and H. Kremar. Modular service architectures: a concept and method for engineering it services. In *Proc. International Conference on System Sciences*, 2003.

[20] J. Bosch. *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[21] Pascal Bouvier. Visual tools to debug prolog iv programs. *DiSCiPl, Lecture Notes in Computer Science 1870*, pages 177–190, 2000.

[22] Manuel Carro and Manuel V Hermenegildo. Tools for constraint visualisation: The vifid/trifid tool. *DiSCiPl, Lecture Notes in Computer Science 1870*, pages 253–272, 2000.

[23] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. Xml-based feature modelling. In *Proc. 8th International Conference on Software Reuse*, 2004.

[24] P. Clements and L. Northrop. *Software Product Lines — Practices and Patterns*. Addison-Wesley, 2001.

[25] CLib. http://www.itu.dk/research/cla/externals/clib/.

[26] K. Czarnecki, M. Antkiewicz, Chang Hwan, and Peter Kim. Multi-level customization in application engineering. *Communications of the ACM - Software product line*, 49(12):61–65, 2006.

[27] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: an industrial experience report. *Proc. ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Componen Engineering*, pages 156–172, 2002.

[28] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[29] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practices*, 10(1):7–29, 2005.

[30] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practices*, 10(2):143–169, 2005.

[31] E. Dashofy, André van der Hoek, and Richard Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In ACM, editor, *Proc. International Conference on Software Engineering*, pages 266–276, 2002.

[32] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[33] Marcos Didonet Del Fabro and Patrick Albert. Software product lines: Lessons learned when applying configuration techniques. In *Proc. Workshop on Configuration, ECAI*, 2010.

[34] Gardner Denver. http://www.gardnerdenver.com/.

[35] Microsoft Excel. http://office.microsoft.com/en-us/excel/. 2010.

[36] EXPRESS. *10303-11 ISO - Part 11: The EXPRESS language reference manual*, 2004.

[37] A. Felfernig, G. Friedrich, and D. Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(4):165–176, 2001.

[38] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *AI*, 152:213–234, 2004.

[39] A. Felfernig, K. Isak, and T. Kruggel. Testing knowledge-based recommender applications. *OEGAI Journal, Special Issue on Recommender Systems*, 24(4):12–18, 2005.

[40] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Uml as domain specific language for the construction of knowledge-based configuration systems. 1999.

[41] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Generating product configuration knowledge bases from precise domain extended uml models. pages 284–293, 2000.

[42] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In J. Howard Johnson, editor, *The 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, 1997.

[43] E. Gelle and B. Faltings. Solving mixed and conditional constraint satisfaction problems. *Constraints*, 8(2):107–141, 2003.

[44] F. Geller and M. Veksler. Assumption-based pruning in conditional csp. *Proc. CP'05*, 3709:241–255, Oct 2005.

[45] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[46] Frédéric Goualard and Frédéric Benhamou. A visualization tool for constraint program debugging. *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 110, 1999.

[47] A. Haag. Sales configuration in business processes. *IEEE Intelligent Systems*, 13(4):78–85, 1998.

[48] Xiao He, Zhiyi Ma, Weizhong Shao, and Ge Li. A metamodel for the notation of graphical modeling languages. pages 219–224, 2007.

[49] M. Heiskala, J. Tiihonen, and T. Soininen. A conceptual model for configurable services. In *IJCAI Workshop on Configuration*, Scotland, 2005.

[50] Brian Henderson-Sellers and Cesar Gonzalez-Perez. Uses and abuses of the stereotype mechanism in uml 1.x and 2.0. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, pages 16–26, 2006.

[51] L. Hvam, N. H. Mortensen, and J. Riis. *Product customization*, volume XII. Springer, 2008.

[52] D. Janitza, M. Lacher, M. Maurer, U. Pulm, and H. Rudolf. A product model for mass-customisation products. *LNCS*, 2774:1023–1029, 2003.

[53] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S .A. Peterson. Feature-oriented domain analysis (foda) — feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[54] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[55] Tero Kojo, Tomi Männistö, and Timo Soininen. Towards intelligent support for managing evolution of configurable software product families. *Proc. of the 11th International Workshop on Software Configuration Management*, 2003.

[56] P. Kotler. *Marketing Management, 11th edition.* Prentice-Hall, 2003.

[57] Lam-Son Le and Alain Wegmann. Definition of an object-oriented modeling language for enterprise architecture. page 222.1, 2005.

[58] C. Lecoutre. Optimization of simple tabular reduction. *Proc. CP'08*, pages 128–143, 2008.

[59] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithm. *Proc. CP'03*, pages 480–494, 2003.

[60] A K Mackworth. On reading sketch maps. *Proc. IJCAI'77*, pages 598–606, 1977.

[61] D. Mailharro. A classification and constraint-based framework for configuration. *AI EDAM*, 12:383–395, Sep 1998.

[62] T. Männistö, H. Peltonen, T. Soininen, and R. Sulonen. Multiple abstraction levels in modelling product structures. *Data & Knowledge Engineering*, 36(1):55–78, 2001.

[63] Tomi Männistö, Hannu Peltonen, Asko Martio, and Reijo Sulonen. Modelling generic product structures in step. *Computer-Aided Design*, 30(14):1111–1118, 1998.

[64] John McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[65] H. Meier, J. J. Schramm, and A. Werding. Development of a stage model based configurator to generate more customer-specific services and to support cooperative service networks. In *3rd CIRP International Seminar on Intelligent Computation in Manufacturing Engineering*, Ischia, Italy, 2002.

[66] Micha Meier. Debugging constraint programs. *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 204–221, 1995.

[67] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. 1990.

[68] R Mohr and TC Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[69] R Mohr and G Masini. Good old discrete relaxation. *Proceedings ECAI'88*, pages 651–656, 1988.

[70] J. Moller, H. R. Andersen, and H. Hulgaard. Product configuration over the internet, 2001.

[71] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principle of Program Analysis*. 2005.

[72] OCL. *OMG Object Constraint Language v2.0 Specification*, 2001.

[73] B. J. Pine. *Mass Customization - The New Frontier in Business Competition*. Harvard Business School Press, 1993.

[74] ProCoLa. http://www2.imm.dtu.dk/ mq/procola/.

[75] Matthieu Quéva. General purpose modelling languages for configuration. Technical report, DTU Informatics, 2010.

[76] Matthieu Quéva, Tomi Männistö, Laurent Ricci, and Christian Probst. A conceptual modelling approach for managing variability in heterogeneous product families. *in progress*, 2011.

[77] Matthieu Quéva, Tomi Männistö, Laurent Ricci, and Christian Probst. Modelling configuration knowledge in heterogeneous product families. In *Proceedings of the IJCAI'11 Workshop on Configuration*, 2011.

[78] Matthieu Quéva, Christian Probst, and Laurent Ricci. Maintaining arc consistency in non-binary dynamic csps using simple tabular reduction. In *Proceedings of the Fifth European Starting AI Researcher Symposium*, Lisbon, Portugal, 2010.

[79] Matthieu Quéva, Christian Probst, and Per Vikkelsøe. Industrial requirements for interactive product configurators. In *Proceedings of the IJCAI'09 Workshop on Configuration*, 2009.

[80] J.-C. Régin. A filtering algorithm for constraints of difference in csps. *Proc. AAAI'94*, pages 362–367, 1994.

[81] Jean-Charles Régin. Maintaining arc consistency algorithms during the search without additional space cost. *Lecture notes in computer science*, 3709:520–533, 2005.

[82] Mark-Oliver Reiser and Matthias Weber. Managing highly-complex product families with multi-level feature trees. In *Proc. of the 14th IEEE International Requirements Engineering Conference*, pages 149–158, 2006.

[83] Renault. http://www.renault.fr/.

[84] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. *Proc. ECAI*, pages 550–556, 1990.

[85] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[86] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. pages 153–161, 1996.

[87] D. Sabin, E. C. Freuder, and R. J. Wallace. Greater efficiency for conditional constraint satisfaction. *LNCS - CP 2003*, pages 649–663, 2003.

[88] Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13:42–49, July 1998.

[89] Microsoft SQL Server. http://www.microsoft.com/sqlserver/en/us/default.aspx. 2008.

[90] M. Sinnema, S. Deelstra, j. Nijhuis, and J. Bosch. Cowamof: A framework for modeling variability in software product families. In Springer Verlag, editor, *Proc. of the Third Software Product Line Conference*, volume 3154, pages 197–213, 2004.

[91] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49:717–739, 2007.

[92] T. Soininen and E. Gelle. Dynamic constraint satisfaction in configuration. *Proc. of AAAI Workshop on Configuration*, Jan 1999.

[93] T Soininen, J Tiihonen, T Männistö, and R Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[94] Microsoft Visual Studio. http://www.microsoft.com/visualstudio/. 2010.

[95] M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(4):307–320, 1998.

[96] SysML. *OMG Systems Modeling Language (OMG SysML) v1.0 Specification*, 2001.

[97] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[98] J. Tiihonen, M. Heiskala, K.-S. Paloheimo, and A. Anderson. Configuration of contract-based services. In *Proc. ECAI 2006 Workshop on Configuration*, pages 25–30, 2006.

[99] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen. Empirical testing of a weight constraint rule based configurator. In *Workshop on Configuration, ECAI*, pages 17–22, 2002.

[100] Juha Tiihonen. Characterization of 26 configuration models. In *Proceedings of the IJCAI'09 Workshop on Configuration*, 2009.

[101] Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen. A practical tool for mass-customising configurable products. In *Proc. of the International Conference on Engineering Design*, 2003.

[102] Eclipse UML2 Tools. http://www.eclipse.org/modeling/mdt/?project=uml2.

[103] J. R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.

[104] UML. *OMG UML V2.1.2 Specification*, 2007.

[105] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[106] M. Véron and M. Aldanondo. Yet another approach to ccsp for configuration problem. *Proc. ECAI'00*, pages 59–62, 2000.

[107] A. Wimmer, J. I. Mehlau, and T. Klein. Object oriented product metamodel for the financial services industry. In *Proc. 2nd Interdisciplinary World Congress on Mass Customization and Personalization*, Munich, Germany, 2003.

[108] R. Winter. Mass customization and beyond — evolution of customer centricity in financial services. In *Workshop on Information Systems for Mass Customization*, Dubai, 2001.

[109] Helen Xie, Philip Henderson, and Michael Kernahan. A constraint-based product configurator for mass customisation. *International Journal of Computer Applications in Technology 2006*, 26(1/2):91–98, 2006.

[110] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171:514–534, 2007.

[111] L.A. Zaid, F. Kleinermann, and O. De Troyer. Feature assembly: a new feature modeling technique. In J. Parsons et al., editor, *ER 2010, LNCS*, volume 6412, pages 233–246, 2010.

[112] Markus Zanker, Dietmar Jannach, Marius Silaghi, and Gerhard 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.