

A Formal Approach to Software Product Families

Alexander M. Gruler

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

A Formal Approach to Software Product Families

Alexander M. Gruler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen
Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Burkhard Rost

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Christian Lengauer, Ph.D.
Universität Passau

Die Dissertation wurde am 17.05.2010 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik am 16.10.2010 angenommen.

Abstract

Software-intensive systems pervade our daily lives. Regarding the product range of organizations which develop software-intensive systems, many organizations do not produce unrelated systems, but rather produce families of similar systems that share certain commonalities. Prominent examples of such families of software-intensive systems can be found in a multitude of different application domains, comprising embedded as well as business information systems. For example the model variants of the same model series of a car manufacturer, e.g. the variants of the *7-series BMW*, or the various variants of an operating system, e.g. the various editions of the operating system *Microsoft Windows 7*, constitute such families.

In order to increase the efficiency in the development of such system families a reuse strategy seems sensible. The integrated development of a family of software-intensive systems by explicitly making use of their commonalities in a strategic and planned way is the subject of software product line engineering.

Despite its obvious motivation, the way of constructing a family of systems by taking advantage of commonalities is not sufficiently explored—in particular with respect to its theoretical foundation. How can reuse based on commonalities between system variants take place in a systematic way? What are the fundamental concepts behind commonalities and differences of related systems? How can commonalities between family members be determined? How can the relation between family members be modeled, and how are commonalities integrated into the construction of the individual family members? What are the fundamental concepts that constitute a software product family, and when can we speak of a software product family at all?

In this thesis we address these and similar questions from the point of view of an underlying modeling theory, and introduce a theoretical framework for the construction of, and the reasoning about software product families and their products.

On the one hand we do this for a very specific kind of software product families, where a product family represents the integrated, implementation-platform independent, *operational behavior* of a set of software-intensive systems. We provide a process algebraic framework for the development of such product families, which allows to benefit from behavioral commonalities for the development of individual family members. The framework comprises (i) the process algebra PF-CCS for the specification of the behavior of a set of software-intensive, reactive systems in an integrated and systematically planned way as a product family, (ii) a multi-valued, modal logic (a multi-valued version of the μ -calculus), which is tailored to the specification and verification of behavioral properties which arise when considering a large variety of similar systems, and (iii) a restructuring concept that constitutes the theoretical basis to determine behavioral commonalities in the operational behavior of family members.

On the other hand we consider software product family concepts in general, and in particular the general construction concept behind any software product family, abstracting from the concrete kind, realization and implementation of the software product family, and the kind of products which the family comprises. We formalize the conceptual construction idea behind any software product family by elaborating an axiomatization of software product family concepts. The axiomatization represents a theoretical basis to manipulate and to reason about software product families in general. In addition, it characterizes the class of software product families.

Note that this thesis does not present a practical approach in the sense that the introduced concepts and methods can directly be applied overnight for the practical development of large software-intensive systems. Essential aspects of bringing a software product line to life, such as the definition of a suitable methodological application of the introduced concepts, or the detailed embedding into a development process, are not covered in this thesis. Our approach serves as a theoretical underpinning for the construction of and the reasoning about software product families, which may guide the creation of practical frameworks.

Acknowledgements

This thesis would not have been possible without the encouragement, the support, and the understanding of many people. To all of them I am deeply grateful.

First and foremost I want to thank my doctoral advisor Prof. Manfred Broy. The conditions and the unique research environment that he has managed to create at the Chair of Software & Systems Engineering at the TUM have made the time in his group an outstanding experience with excellent work and research opportunities, and I consider myself privileged to have been a part of it. I am very grateful for the freedom and the time he has granted me to find and pursue the topic of this thesis, and for his patience with me during that time. I also want to thank Prof. Christian Lengauer, who agreed to act as second supervisor, and who provided very constructive and helpful suggestions to improve this thesis.

During the course of creating this thesis I had to undergo some situations that gnawed at my motivation. The encouragement that I received in these situations, especially from my colleagues Martin Leucker and Makarius Wenzel, was most welcome, and I want to thank you for that.

Undoubtedly I owe Martin Leucker a dept of gratitude. In his position as an experienced researcher and post-doc in our group it was him who taught me the ABC of scientific work and who introduced me to many interesting research areas. I am very happy to call him both my teacher and my friend. Special thanks to our secretary Silke Müller, the heart of the entire group. It is certainly no exaggeration to say that whenever she happened to be out of office, everyone immediately wished her

back. I also want to thank my colleague Judith Thyssen for being a wonderful office mate during the last years.

I am very grateful to Stefan Berghofer, David Cruz, Peter Höfner, Martin Leucker, Christian Leuxner, Daniel Ratiu, Martin Sachenbacher, Daniel Thoma, and Makarius Wenzel for reading parts of my thesis and for providing me very valuable feedback.

Finally, I want to thank my family, in particular my mother and my father. Ultimately, it was their care, their love, and their way of raising and educating me, that has set the course for what and where I can be today.

Munich, May 2010

Contents

1. Introduction	1
1.1. Software Product Line Engineering	2
1.2. Challenges and Their Backgrounds	6
1.2.1. Requirements for an Improved Engineering Approach	11
1.3. Contributions	12
1.4. Related Work	17
1.5. Thesis Outline	18
2. Formalization of Characteristic Software Product Family Concepts	21
2.1. Software Product Families and Lines: An Informal View	23
2.2. Axiomatization of Software Product Family Concepts	30
2.2.1. Preliminaries: Algebraic Specification	31
2.2.2. Operations for Constructing a Software Product Family	32
2.2.2.1. Core Assets and Neutral Element	33
2.2.2.2. Composition	35
2.2.2.3. Variation Points and Variants	35
2.2.2.4. Example: A Product Family of Stickmen Drawings	39
2.2.3. Axioms, Properties and Auxiliary Operations	43
2.2.3.1. Axioms for Constructors	43
2.2.3.2. Term Normal Form of Product Families	46
2.2.3.3. Configuration: Derivation of Products	60
2.2.3.4. Properties of the Variants Operator	65
2.2.3.5. Sub-Families	67
2.2.3.6. Products	70
2.2.3.7. Common Parts	72

2.2.3.8.	Optional and Mandatory Parts	75
2.2.3.9.	Evolution of Software Product Families	81
2.2.3.10.	A General Variants Operator for n Variants	85
2.2.4.	Complete Algebraic Specification of the Sort $\text{SPF } \alpha$	88
2.3.	Modeling Dependencies in Software Product Families	92
2.4.	Discussion	97
2.4.1.	On the Choice of an Algebraic Specification	97
2.4.2.	Structural Similarity to an <i>AND/OR-Tree</i>	99
2.4.3.	Combining the Axiomatization with a Type System	100
2.5.	Related Work	101
3.	PF-CCS: Product Family CCS	109
3.1.	Syntax of PF-CCS	111
3.1.1.	Well-formed PF-CCS programs.	116
3.2.	Semantics of a PF-CCS Program	121
3.2.1.	Flat Semantics	121
3.2.2.	Unfolded Semantics	125
3.2.3.	Configured-transitions Semantics	139
3.3.	Design Decisions for PF-CCS	147
3.4.	Practicability of PF-CCS	150
3.4.1.	Value-Passing PF-CCS	150
3.4.2.	Placing PF-CCS in the Development Process	151
3.5.	Related Work	155
4.	Verifying Properties of PF-CCS Software Product Families	159
4.1.	The Multi-Valued Modal μ -calculus	162
4.1.1.	Syntax of the Multi-Valued Modal μ -Calculus	163
4.1.2.	Semantics of the Multi-Valued Modal μ -Calculus	164
4.1.3.	Correctness of the Provided Semantics	172
4.2.	Model Checking	177
4.3.	Example: Verifying a Family of Windscreen Wipers	180
4.3.1.	Specification of the Product Family of Windscreen Wipers	180
4.3.2.	Verification	182
4.4.	Related Work	185
5.	Restructuring PF-CCS Software Product Families	191
5.1.	Algebraic Laws	194
5.1.1.	Distributivity of Action Prefixing over \oplus	195
5.1.2.	Distributivity of Non-Deterministic Choice over \oplus	198
5.1.3.	Distributivity of Parallel Composition over \oplus	202
5.1.4.	Miscellaneous Laws	204
5.2.	Calculating Commonalities: A Detailed Example	206
5.3.	Common Parts	214

6. Conclusion and Future Work	221
6.1. Discussion	225
6.2. Future and Ongoing Work	228
A. Selected Algebraic Specifications	231
B. Uniqueness of the Normal Form: Proofs	239
B.1. Auxiliary Lemmata	264
C. Lattices	267
D. The Modal μ-Calculus	269
Bibliography	287

List of Figures

2.1. Algebraic specification of the sort <code>Stickman</code>	39
2.2. Basic graphical shapes of stickmen drawings.	40
2.3. Term representation of a software product family of sort <code>SPF Stickman</code>	41
2.4. An example of a concrete product family: A family of stickmen.	42
2.5. A term of sort <code>SPF α</code> and its normal form.	58
2.6. Derivation of a sub-family of the product family <code>Spf</code>	69
2.7. A sub-family of the product family <code>Spf</code>	69
2.8. Graphical illustration of the distributive law of \oplus over \parallel	73
2.9. Core assets of the Σ -algebra <code>StickmanShapesII</code>	76
2.10. Restructuring a product family with the distributive law.	76
2.11. Algebraic specification of a software product family of sort <code>SPF α</code>	90
3.1. A program dependency graph.	118
3.2. Two ways of understanding alternative variants and variation points.	119
3.3. SOS rules for CCS as defined by Milner [Mil95].	124
3.4. PF-CCS SOS rules for the unfolded semantics.	128
3.5. An example of a PF-LTS and a corresponding deduction.	130
3.6. PF-LTS for the PF-CCS term $P \stackrel{def}{=} \gamma.(\alpha.P \oplus_1 \beta.P)$	131
3.7. A PF-LTS representing the unfolded semantics and its projections.	133
3.8. Transitions systems of products derived by projection.	136
3.9. Constructing of the configured-transitions semantics.	143
3.10. Configured-transitions Semantics for a simple recursive process.	144
3.11. Layer framework for a seamless model-based development process.	153
4.1. Syntax of $mv\text{-}\mathcal{L}_\mu$	163
4.2. Semantics of $mv\text{-}\mathcal{L}_\mu$ formulae.	168

5.1.	PF-LTS illustrating the distributivity of action prefixing.	197
5.2.	Action prefixing does not distribute over CCS's $+$	199
5.3.	PF-LTSs illustration the distributivity of $+$ over \oplus	201
5.4.	Specification of a front screen wiper FWS , standard version.	207
5.5.	Specification of a front screen wiper FWC with comfort features. . .	208
5.6.	PF-CCS program $(\mathcal{E}, FWFam)$ specifying a family of front wipers. .	210
5.7.	Calculation of the commonalities of FWS and FWC	211
5.8.	Restructured version of the front wiper product family.	213
A.1.	Algebraic specification of the sort Bool	232
A.2.	Algebraic specification of the sort If-then-else	232
A.3.	Algebraic specification of the sort Nat	233
A.4.	Algebraic specification of the sort Seq α	234
A.5.	Algebraic specification of the sort Set α	235
A.6.	Algebraic specification of the sort MSet α	236
A.7.	Extension of the algebraic specification of sort MSet	237

List of Definitions

2.1. Software Product Line [CN01]	23
2.2. Program Family [Par76]	24
2.3. Software Product Family (Informal Characterization)	26
2.4. Complete and Atomic Configuration	60
2.5. Equivalence of Complete Configurations	66
2.6. Sub-Family	68
2.7. Set of Derivable Products	72
2.8. Product Equivalence of Software Product Families	72
2.9. Common Part of Variants	73
2.10. Software Product Family	89
2.11. Dependency Model	93
2.12. Normal Form of a Dependency Model	94
2.13. Valid Configuration	96
3.1. Syntax of PF-CCS Process Expressions	112
3.2. PF-CCS Program	114
3.3. Complete PF-CCS Program	116
3.4. Program Dependency Graph	117
3.5. Finitely Configurable PF-CCS Program	118
3.6. Fully Expanded PF-CCS Program	120
3.7. Well-formed PF-CCS Program	120
3.8. Variation Point in PF-CCS	120
3.9. Configuration Vector	121
3.10. Fully Configured Configuration Vector	122
3.11. Flat Semantics of a PF-CCS Program	124
3.12. PF-LTS for the Unfolded Semantics	125
3.13. Conformance of Configuration Vectors	126

3.14. Concretization of a Configuration Vector	127
3.15. Unfolded Semantics of a PF-CCS Program	129
3.16. Projection of a PF-LTS (Unfolded Semantics)	132
3.17. Bisimulation	134
3.18. PF-LTS for the Configured-Transitions Semantics	140
3.19. Configured-transitions Semantics of a PF-CCS Program	141
3.20. Projection of a PF-LTS (Configured-transitions Semantics)	142
4.1. Multi-Valued Modal Kripke Structure (MMKS)	165
4.2. Consistency of Variable Environments	173
5.1. Projection of a PF-LTS According to a Set of Configurations	215
5.2. Common Part of a Set of Configurations	215
5.3. Representation Showing the Greatest Common Part	218

CHAPTER 1

Introduction

The aim of software product line engineering is to increase the efficiency in the development of a set of software-intensive systems by applying a strategic and planned form of reusing the commonalities between the products. In this chapter we provide a general, informal introduction into the field of software product line engineering, and discuss the benefits compared to independent system development. Motivated by the state of the art in the development of reactive, software-intensive systems in the automotive sector, we outline current challenges which arise from the huge product variety within automotive systems, and motivate how the adaptation of software product line engineering techniques can help to master them. In this context, we list and discuss our contributions. At the end of the chapter we provide a brief road map for reading this thesis.

Contents

1.1. Software Product Line Engineering	2
1.2. Challenges and Their Backgrounds	6
1.3. Contributions	12
1.4. Related Work	17
1.5. Thesis Outline	18

1. Introduction

1.1. Software Product Line Engineering: From an Ad-hoc to a Planned Form of Reusing Commonalities

The importance and influence of software in our every days life is undisputable. The vast majority of electronic devices with which we interact during a day is mainly controlled by software — in fact, software-based systems pervade our daily life. Obviously, this starts with systems such as personal computers, mobile phones, but also includes systems of which we do not expect it at first glance, such as for example artificial knee joints [Cam88, Boc09] which compute in real-time their resistance level according to the current environmental factors in order to smooth and optimize the movement. Taking some examples of systems from a more complex order of magnitude, we can see an airplane or a premium class automobile as two prime examples for very complex, software-based systems. For example, Boeing’s new 787 *Dreamliner*, scheduled to be delivered in 2010, requires about 6.5 million lines of code to operate its avionics and on board support systems [Cha09]. However, this number is still exceeded by modern premium class cars which consist of up to 70 electronic control units (ECUs) carrying up to 100 million lines of code [Cha09, BKPS07]. In such highly complex systems, basically the entire functionality is exclusively controlled and realized by software. In this context we speak of *software-intensive* systems.

For the development of software-intensive systems the importance of software is still increasing. A main reason for this trend is the fact that new features¹ and functions which are offered by such systems can be realized to a steadily increasing degree by software. Innovation becomes more and more software driven. A study conducted in 2005 for the European Commission [BDG⁺05] revealed that in contrast to the increasing importance of software, especially in the area of embedded systems, advantages in the development of new hardware technologies gradually lose their immediate impact on the success of a new product. For example for automotive systems, 80% of the innovations in a car come from software-intensive systems [BKPS07]. This makes software—especially from an economical point of view—a key factor for the return on investment of many products. Consequently, against the background of achieving universal business goals such as high quality of the product, a quick time to market, low cost production and maintenance, and mass customization, improving the efficiency and productivity in the area of software development is a more and more decisive factor.

With software being a key production factor, also the efficiency of the software development process moves in the focus of economic and commercial interest. There are various ways of realizing a more efficient software development: improvement of the development process, the utilization of new, innovative technologies, or the

¹For us a *feature* is any observable property which a system exhibits. This can be a functional or a non-functional property, i.e. a feature itself can have a behavior or simply be an object property, e.g. leather upholstery in the car.

increase of reuse of existing artifacts are just some examples. For software—being an intangible good which can easily be replicated—in particular the increase of the degree of reuse [Kru92, HM84, Sta84] is a very promising possibility. In fact, improving the efficiency by reusing existing software artifacts or software components for the development of new systems is a key concern since the software crisis in the late 1960s, when the idea of reusing *mass-produced software components* [McI69] was coined by Douglas McIllroy and first discussed at the NATO (Software Engineering) Conference in Garmisch in 1968 [NR69].

However, a problem with a longstanding tradition is how to realize an applicable way of reuse. Since the software crisis, many paradigms and concepts have come up which strive for a more efficient software development process by providing—directly or indirectly—improved ways of reuse:

- During the 1960s and 1970s, structuring a program (especially its source code) into logical parts was in the center of interest. The so-called *Goto* statements—leading inevitably to unstructured programs [Dij68]—were replaced by concepts such as *subroutines* (1960s) or *modules* (1970s). These ideas led to the paradigm of *procedural programming* comprising concepts such as *separation of concerns* [DDH72] or *modularity* [Par72], being essential fundamentals for reuse.
- With programs becoming larger and more complex, *procedure calls* became the dominating programming statement. Data structures and the operations on them were combined more closely, realized by the concept of *objects* [Mey97] which is pioneered in the languages Simula67 [DMN68] and Smalltalk [Kay93]. In general, the focus was less on lists of commands/tasks and more on a structure of communicating objects. *Object-oriented* concepts [Mey87] allowed reusable entities to be adapted more easily, leading to larger reusable entities. Simultaneously, the concept of *libraries* (of procedures/classes) was promoted.
- In the 1990s, the concept of constructing complex systems by assembling units of composition, so-called *components* [Szy98], with a well defined interface, moved in the focus of interest. In contrast to objects, which were designed to match a specific (mental) model and which were not explicitly designed for multiple use in other contexts, the distinguished idea of a component is to encapsulate an independently deployable behavior together with the necessary data structures. This allows components to be reused more easily in multiple contexts/programs.
- In the *service-oriented* paradigm, pieces of functionality—usually structured according to business processes—are grouped as interoperable services which are made available over a network in order to be used in the development and integration of business applications. Service-oriented design [Bel08] and

1. Introduction

in particular *Service-Oriented Architecture (SOA)* [Erl05] got ahead of the component idea, coming with improved concepts of how components can be combined, offered, searched and remotely executed. Services are being cut according to the (business) process, which increases their reusability compared to components, in general.

Although these traditional reuse strategies gradually raised the degree of reuse, they have had not the expected economic benefit [Kru92, BBM96]. Reasons therefore are complex and manifold, and we refer to [Kru92, Sch99] for a more comprehensive discussion. In summary, the most important reasons were that

- Reuse was accomplished in an *ad-hoc* or even *opportunistic* way, in general. Even though reusable entities such as algorithms, modules, classes, or components were collected in libraries, they were not explicitly designed for future reuse in a specific context or architecture in a pre-planned way. Although, many developers have successfully applied reuse opportunistically, e.g. by copying and pasting pieces of code from existing programs into new programs, such a form of opportunistic reuse only works fine in small groups but does not scale up to larger business units or entire enterprises [Sch99]. In summary, there was no strategic plan for future reuse.
- All traditional reuse strategies envisioned the idea of *general-purpose* reuse. Here, the reusable entities were not tailored to be reused in a specific, future environment or context in a pre-planned way, but were rather developed to be reused “somehow” in an (at the design time) unknown, future situation. In particular, the reuse environment/context was not taken into account during the development of the reusable entity.

In contrast to traditional reuse strategies and their aims, *software product line engineering* realizes a different reuse concept, which acts on the assumption of a different basic situation. Regarding the product range, we observe that companies across the entire software industry very often do not produce isolated/unrelated (software) systems, but rather produce a family of similar systems which exhibit certain commonalities. For example, the operating system *Windows Vista* is available in 6 different versions, ranging from *Home Basic*, *Home Premium*, *Business*, *Ultimate* to an *Enterprise* edition and *64-Bit Version*. The free, Linux-based operating system *Ubuntu* even comes in a potentially unlimited number of variants due to a packet management concept which allows to freely add or remove software components. In both cases we can identify many commonalities between the versions.

Compared to general-purpose, ad-hoc reuse, in software product line engineering the commonalities between systems in such a situation are exploited in a strategic way, by developing the systems in common, i.e. in a integrated and planned way, instead of developing the systems independently from each other and ignoring their

commonalities. Already 40 years ago, Dijkstra described a vision [DDH72]—at that time still in the context of programs rather than entire systems—which perfectly expresses the fundamental principle of modern software product line engineering:

If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say, different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common.

In accordance to Dijkstra’s vision, in software product line engineering, several systems are developed in common by modeling the variable and common parts of the systems in a consolidated way. This allows to reuse parts of one system for the construction of other systems, and thus maximizes the degree of reuse by systematically exploiting commonalities.

Fundamental to such an approach is a common architecture or construction plan which describes how the systems are constructed from the common set of atomic components, the so-called *core assets*. In this context we also speak of the *product family*². In particular, the software product family represents a set of derivable systems *in its entirety*, emphasizing how the systems are actually constructed from the common set of core assets. An individual system—in the context of software product line engineering also called *product*—can be obtained by deriving it systematically from the product family. Such a derivation process is called *configuration (configuration process)*, and we usually speak of *configuring a product family* in this context.

By implementing a strategic way of reuse, software product line engineering differs fundamentally from other development techniques. In particular, software product line engineering is not single-system development with reuse, since building a software product line means planning a plurality of products which will exist and be maintained quasi *simultaneously*—in contrast to single systems that evolve over time in an uncoordinated way. Moreover, single-system development with reuse usually starts with the latest version of a system, duplicates the code and adjusts the duplicated version to the new requirements. Thus, each new version of a system is build on its own, loosing the connection to earlier versions. As a result, knowledge of commonalities is not preserved and common parts cannot be systematically reused.

By making use of inter-product commonalities in a strategic way, software product line engineering reaches a high degree of efficiency. The fact that the application

²We call a set of software-intensive systems a *software product family* when the systems are built by explicitly making use of commonalities for their *construction*. In contrast, the term *software product line* emphasizes the economic aspect and the general concept of efficient design rather than the constructional aspect. For a more detailed differentiation we refer to Chapter 2.1, Page 27.

1. Introduction

of software product line engineering techniques actually has a dramatic economic impact—which manifests itself for example in an increase in productivity, reduction of costs, or an improvement of the quality of the product—is supported by some very successful real-life industry projects. Many of these software product line success stories are gathered in the so-called *Software Product Line Hall of Fame* [SEI], from which we want to mention two quite successful examples:

- In 1993, *Cummins Inc.*, a manufacturer of large diesel engines, faced the situation to produce new systems with a too little amount of staff and resources. By changing to a software product line approach [CN01], today, the product line contains more than 1000 separate engine applications, where (i) product cycle time was slashed from 250 person-months to a few person-months, (ii) the build and integration time was reduced from one year to one week, and (iii) the quality goals are exceeded which manifests in a very high degree of customer satisfaction.
- *CelsiusTech AB*, a Swedish naval defense contractor, successfully adopted a product line approach—called *ShipSystem 2000*—to building a family of 55 ship systems [BCKB03, BC96]. As a result (i) the need for developers dropped from 210 to roughly 30, (ii) the time to field decreased from about 9 years to about 3 years, (iii) the integration test of 1-1.5 million lines of code requires only 1-2 people, (iv) and rehosting to a new platform/OS reduced to 3 months.

After this general survey on software product line engineering, we study the current situation and the corresponding challenges in a concrete domain, which is perfectly suitable for the application of software product line engineering techniques, the automotive domain. Our approach together with our contributions, which are mainly motivated from concrete challenges of the automotive domain, are introduced in Section 1.3.

1.2. Challenges and Their Backgrounds

The automotive domain is predestinated for the application of software product line engineering techniques due to (i) the huge product variety in the product range of a car manufacturer (also called an OEM in the automotive terminology), (ii) the continuously increasing complexity of the software-intensive system “car” in combination with the intense cost and time pressure, and (iii) the large quantities in which cars and its software-intensive subsystems are produced. Certainly, there are similar situations in other domains, for example the avionics, telecommunication, or automation industry, but usually not in such a distinct kind as we encounter them in the automotive domain. For this reason we use the automotive domain as an example in place of similar application domains, and describe partially the current

situation together with the software product line specific challenges in the engineering process of software-intensive automotive systems. However, we emphasize that our theory is not exclusively tailored for automotive systems, and is likewise applicable for families of reactive systems in general.

The application of software product line engineering techniques allows for an economically more efficient development and production process, and is the basis to deal with the increasing complexity effectively. However, despite its suitability the application of software product line engineering techniques in the automotive sector is still very low. The exploitation of commonalities by reusing artifacts is usually done in an ad-hoc way without a strategic plan as suggested by software product line engineering concepts. As we will see in the following, many reasons for this situation are of a fundamental nature, i.e. not the application of existing methods and techniques in the industrial practice is the problem, but rather the lack of a fundamental theory and principles of how to deal with certain questions in the context of engineering a family of similar systems in an integrated way. In the following, we describe this situation and the currently relevant challenges in the automotive domain in more detail.

Challenges Related to the Functionality in Combination with the Huge Product Variety

In the automotive domain we observe a huge variety within the product range of a car manufacturer. Modern car models, e.g. the *7-series BMW*, usually come in a multitude of different model variants [Sch08] which differ in the extra equipment they provide, especially in the amount and kind of features. For example, for a modern premium class car around 80 optional electronic features can be ordered, which already implies the existence of 2^{80} combinatorially possible configurations. Main drivers for this huge variety are:

- The automotive market requires individuality, and the request for customizability is high. Every customer desires its own, individually configured version of a car which is tailored to his needs. Consequently, in order to serve the demands of the market an OEM offers a car in a variety of model variants.
- An OEM usually serves an international market delivering its good on a global scale. Beside the diversity in the language and culture of different regions and countries, also the diversity in the national markets and the statutory situation requires a car model to exist in various country-specific variants.

A modern car is a prime example of a software-intensive, reactive system. For a modern car, most of its features directly determine the functionality of the car and are thus directly software-relevant. We also speak of *functions* or *functional*

1. Introduction

features, e.g. simple ones such as operating the power windows, but also complex ones, such as the *Adaptive Cruise Control (ACC)* [Gmb02], in contrast to *non-functional* features, e.g. exterior color or optional leather upholstery. Regarding the functionality, a modern premium class car implements around 2000 basic *software functions*, from which 270 so-called *user functions* are composed [BKPS07]. In this context, user functions are those functions which are directly accessible by the user (driver and passenger), and which consequently strongly influence the market value of a modern car.

Specification of the Functionality of a Set of Model Variants

The implementation-platform independent functionality is of particular interest for the development of software-intensive, reactive systems, especially for such complex systems as we find them for example in the automotive domain. The overall functionality of a modern car is determined by its individual functions and features, and thus directly depends on the respective configuration of the car. The functionality of an individual car can be represented in many ways. Depending on whether the focus is on a more interface centric, black-box view, or on a more operational, glass-box view of the functionality of a system, possible representation techniques are for example stream-processing functions [Bro05], *message sequence charts* (MSC) [IT96], various kinds of automata (e.g. *I/O-automata* [LT89], *Interface-automata* [dAH01]), *modal transition systems* [LT88], *Petri nets* [Pet62], or process algebraic calculi (e.g. *ACP* [BKT84], *CCS* [Mil95], or *CSP* [Hoa85]). While all these techniques allow a more or less implementation-platform independent representation of functionality, they do not explicitly support the notion of behavioral *variability* [CHW98, SD07] in a deterministic sense, as it is needed to represent the concept of different configurations. In particular in the presence of the huge product variety which an OEM has to face, the set of possible model variants is too big as that the functionality of each model variant could be specified separately using one of the techniques mentioned above. New techniques which support the notion of (behavioral) variability are required to model the behavior of a set of model variants in an combined way.

Detection of Undesired Behavior in the Presence of Many Variants

While traditionally the functions in cars were largely independent and not related with each other, e.g. the engine control was independent of braking and steering, in modern cars previously unrelated functions start to interact and become related to one another. The main reason for this development is the technological advance which offers new possibilities (advanced electronics and programmable software) for the implementation of functions and their interaction in a car. Some of these combinations lead to an undesired or even dangerous system behavior which only results from the integration of functions, and which cannot be predicted by inspecting the modular specifications of the functions separately. Such a kind of undesired behavior is usually referred to as *feature interaction* [Zav93]. While the *detection* of erroneous behavior due to feature interaction is already problematic for the integration of standalone systems, it becomes even more complicated in connection with

the huge variety of possible combinations of functions in a car. Here, the challenge for an OEM is to assure that none of the possible combinations of functional features leads to an undesired or erroneous behavior or system state. Essentially, this reduces to the question to find out and to verify that a behavioral property holds for a set of model variants. For safety-critical functions the detection of erroneous and undesired behavior becomes an even more severe issue. For safety-critical features it is not sufficient to test only the most likely configurations, but it is essential to assure that certain errors never occur, independently of how a model variant is configured. For example with the introduction of *X-by-Wire* [WNSSL05, Jur09] technologies for safety-critical systems in a car, this question became very important. How can an OEM guarantee that all possible model variants which are equipped with *Break-by-Wire* technology (i.e. the break is not controlled mechanically but electronically) always break whenever the break pedal is pushed, independently of other functional features, e.g. driver assistance systems like ACC, or break energy reconversion systems, which might also exist in the car and influence the breaking process.

While such questions are typically addressed in the context of *single* system development by various logics (e.g. LTL [Pnu77], CTL [CE81a], CTL* [EL86], the modal μ -calculus [Koz83], or TLA [Lam94]) and the corresponding verification techniques (e.g. model checking [CGP99]) there are no particular logics, theories, or techniques that allow to address the same questions in the context of a huge amount of different model variants in an adequate way. Moreover, due to the huge variety the naive way of constructing all possible model variants and checking each variant individually is not feasible—in particular not with the steadily increasing time and cost pressure which the automotive market is currently facing. This makes the detection of undesired behavior, and the verification of behavioral properties in the presence of a highly variable system an essential but also very challenging task for an OEM, whose role in these days has become that of a system integrator rather than that of a mere system assembler.

Challenges Related to Reuse

Due to the similarity within the product range of automotive systems, the automotive domain affords many opportunities which would benefit from reuse, and where reuse is explicitly desired but not systematically implemented so far. We separate between different reuse scenarios, which mainly reflect an OEM's point of view.

1. Reuse between successive generations of the same model series.
2. Reuse between the model variants of the same model series.
3. Reuse between different model series, by offering a feature or a function in car models of different model series.

1. Introduction

In these scenarios the kind of artifacts which are reused are different, but as we will discuss in the following, in particular the abstraction level where the *operational functionality* of the model variants is considered, is very important to organize, structure and specify reusable entities.

An OEM usually releases approximately every 4.5 years a new generation of a model series. Comparing consecutive model generations, functionality typically differs only marginally. In fact, according to [BKPS07] differences in the functionality are not more than 10% between consecutive model generations. While the functionality is fairly “stable”, the software itself differs much more between consecutive model generations. In this light, platform-independent functionality seems to be a suitable quantity to specify and structure entities of reuse. In particular in combination with generative approaches [CE00], where the platform-specific, deployable code is generated from high-level models, reusing functionality is very attractive. Here, functionality refers in particular to an operational representation of functionality, since the focus is on implementation in this context. However, currently in the automotive sector the operational functionality which is realized by a set of model variants (e.g. of the same series) is not adequately represented. In particular, the entire functionality which is offered by a set of variants, e.g. by the model variants of the same series, is not represented as an integrated whole. In this context a main challenge for an OEM is to model the operational functionality of a set of model variants in a combined way, and to relate such a model to the functionality of individual model variants, as an integrated representation of the behaviors of the various model variants is the basis to determine and specify behavioral differences and commonalities between the model variants.

With respect to reuse within a model series, for example the *7-series BMW*, all models of the same series come in a standard equipment, which is characterized by the minimal set of features which exist in all model variants of this model series. Regarding the functional features, this means that all model variants within a series have a common “standard” functionality, and especially a common (operational) behavior. Think for example of the various windshield wiper systems which exist in a model series. Some model variants are equipped with a rain sensor which affects the wiping speed of the wiper arm, other models variants are equipped with a start/stop automatic for the engine which causes the wiper to stop if the engine is stopped at red lights, while other model variants have both or even none of these features. But what is the common functionality of the windshield wiper system, that is offered in all model variants? With respect to an implementation this common operational functionality can be used to guide the development of a basic implementation that is part of the every wiper system variant. While the question to determine the common behavior of a set of variants might still be answered for this simple sub-system without the usage of formal methods, determining the common functionality of some more complex systems, for example entire model variants of

a car series, is unlike harder and requires an integrated representation of the functionality of all model variants supported by the corresponding formal methods and techniques. In summary, the ability to determine the standard functionality on an operational level would be very beneficial for an OEM, since the models within one series use the same implementation platform, and an implementation of this standard functionality can be reused in every model variant. However, currently this remains a challenging task in the automotive sector which cannot be solved with the state-of-the-art development techniques.

Regarding new features, a typical procedure in the automotive domain is to introduce new features with the release of a new model generation, since new features increase the product value for the customer, and serve as valuable sales arguments. Once a new feature has been introduced in a model series it is taken over to other model series. For example, the first radar based Adaptive Cruise Control called *Distronic* was introduced with the release of the 1998 *Mercedes S-Class (W220)*, and subsequently taken over to the 1999 generation of the *E-class* and other model series. However, with the introduction of new functions to a model series, the standard functionality of the model series is usually changed, and consequently the commonalities between all model variants are affected, too. Here—similarly to the situation described above—the ability to determine the common part of the (operational) functionality is again very useful and desirable. In particular, if it is supported by formal models and methods. Then, the functionality can become a central role not only for the specification, but also for the construction and verification of model variants.

1.2.1. Requirements for an Improved Engineering Approach

Based on the previously introduced challenges we derive the following requirements for an improved engineering approach for families of software-intensive, reactive systems as they exist for example in the automotive domain. Here, with *systems* we mean the software-relevant part of systems that are produced by OEMS, e.g. entire automobiles, as well as systems that are produced by suppliers, and that exist as subsystems of automobiles. In our opinion the automotive industry would greatly benefit from:

- A standardization and a formal definition of the fundamental concepts and techniques which are necessary to develop a family of similar systems in an integrated way that allows to benefit from the commonalities between the systems.
- Concepts for the representation and modeling of the operational functionality of automotive systems in a platform-independent way. Due to the large variety

1. Introduction

in which automotive systems exist, such a representation technique has to support the notion of behavioral variability, i.e. it has to provide the concepts to represent the behavior of entire *families* of “behaviorally similar” systems in an integrated and coordinated way.

- A formalism that allows to represent the connection between the operational behaviors of similar systems, and that facilitates to develop the behavior of similar systems in an coordinated way.
- A formalism and the corresponding concepts to determine the common behavior of a set of similar (functional) features, systems, or model variants.
- The concepts to determine those model variants that exhibit certain behavioral properties, and the concepts to verify the validity of such properties for the corresponding model variants formally.
- The possibility to inspect all possible combinations of functional features, and to determine those combinations (and the corresponding systems) which exhibit undesired or erroneous behavior, without having to construct and inspect each feature-combination, i.e. each system, “manually”.
- The concepts to determine the differences in the operational functionality between successive model generations or even different model series.
- A formalism to determine an operational representation of the standard functionality which is realized by every model variant of the same model series. This is equivalent to the concepts to determine the “greatest” common behavior that is implemented by every member of a family of systems.

Certainly, from the point of view of the automotive industry, the primary interest is ultimately in the practical implementation of all of these requirements. However, practical solutions require the corresponding theoretical and conceptual foundation on which they can build. For all concepts which we introduce in this thesis, the creation of such a conceptual foundation is our declared goal.

1.3. Contributions

In Section 1.1 (Page 5) we have already introduced the distinct idea which is embodied by a software product family. It is to *construct* systems not independently but in an *integrated way* as members of a family by systematically *making use of commonalities* which exist between the systems. But what does it conceptually mean to construct the family members in an integrated way? How is the construction of

individual systems related and coordinated? What does it mean for the construction of the systems to develop them by using their commonalities? What does it mean that products have common parts, and how can a common part itself be specified? How are the commonalities (re-)used? How can the commonalities of the family members be determined?

As these questions demonstrate, while the general idea and rationals of software product line engineering seem to be clear, a corresponding theoretical foundation which addresses the construction of a software product family is not sufficiently understood. With this thesis we contribute to the theoretical foundation of software product line engineering and introduce a theoretical framework that deals with the construction, manipulation and verification of software product families. We address two aspects: In a first part we consider software product families *in general*. We abstract from the kinds of products which are constructed, as well as from realization and implementation-specific details of product families, and investigate the fundamental construction concepts which are common to software product families in general. We do this by elaborating an axiomatization of software product family concepts. In a second part, we focus on product families for a very specific purpose: we consider product families that capture the *operational functionality* of a set of software-intensive, reactive systems. Products of such families are representations of the operational behavior of individual reactive systems. To this end we introduce a process algebraic framework that allows to model the operational functionality of a set of software-intensive systems as a product family. In particular, we introduce the concepts of how to specify the integrated behavior of a set of systems as a product family, how to reason about behavioral properties of the product family and its members by means of a modal fixpoint-logic, and how to manipulate and restructure such product families in order to determine behavioral commonalities of the family members. Regarding these topics some of the results have already been published in [GLS08b, GLS08a, GHH07]. In the following we describe the contributions in more detail, and discuss them in the light of the challenges which we have described in the previous section.

Note that even though the framework which we elaborate in this thesis is motivated by challenges from the industrial practice, we contribute to the theory and conceptual development, and *not* to the practical development of software product families. In particular, our framework does not make a direct contribution to the development of large-scale, reactive systems, as they are encountered in the current industrial practice. However, with our theory we solve conceptually fundamental problems and create the theoretical basis on which a practically applicable system engineering method has ultimately to be based.

Formalization of General Software Product Family Concepts

- We axiomatize the general notion of a software product family (cf. Chapter 2.2). The axiomatization comprises the definition of the fundamental operations and axioms which capture the laws that hold in software product families

1. Introduction

in general. With the axiomatization we provide a theoretical foundation for the construction and the handling of software product families. In particular, the axiomatization allows to reason about software product family specific concepts by means of formal methods, e.g. interactive theorem provers like Isabelle [Pau94]), and allows to derive new properties.

- As part of the axiomatization (cf. Chapter 2.2.3) we identify and precisely characterize typical concepts of software product families, such as *variation points*, *variants*, *optional*, *alternative* and *mandatory parts*, *commonality*, *configuration*, etc., which lack a precise, formal definition, so far.
- We define a *unique normal form* for software product families (cf. Chapter 2.2.3.2). The normal form is a representation of a product family which explicitly shows the commonalities between alternative variants, which is free of trivial variation points, and which fulfills other well-formedness properties. Based on the normal form we define the notion of equality between software product families. The axioms which characterize the normal form can directly be turned into a functional program, and thus represent an executable algorithm to transform arbitrary product families into their normal form.
- We define the notion of commonalities between products of a software product family (cf. Chapters 2.2.3.7 and 2.2.3.8). Common parts of products are defined indirectly based on the differences between products of the family which are specified explicitly. We establish the formal connection between common and variable parts, and describe how to work with commonalities in software product families in an operational way.
- We introduce a dependency model for software product family specifications (cf. Chapter 2.3). Unlike to the representation of a software product family itself, the dependency model does not describe how products are constructed but rather restricts the set of possible configurations (and thus products) of a product family, and characterizes those ones which shall be constructed. The dependency model uses propositional logic and allows to express dependencies based on the configurations of individual variation points. Thus, it allows to specify a multitude of different kinds of dependencies, also those ones which typically [Bat05] exist in the context of variable parts, for example *requires*, and *excludes* relations. A dependency model is especially essential for the realistic application since it allows to filter out those configurations of a software product family which must not exist due to non-functional reasons, e.g. marketing decisions.

Modeling the Operational Functionality of a Set of Systems as a Software Product Family

- We introduce *Product Family CCS* (PF-CCS), a process algebraic framework for the specification of the *operational functionality* of a set of systems as a

software product family (cf. Chapter 3). The PF-CCS framework allows (i) the specification of the operational functionality of an entire family of systems in an integrated way, and (ii) the derivation of sub-families (partially configured systems) and products from a PF-CCS representation of a software product family. PF-CCS gives the formal relation between the operational behavior of one system and the operational behavior of all other systems which are developed as part of the same product family. From an economic point of view, PF-CCS is the basis to realize a strategic way of reuse at the abstraction level of the operational functionality. More precisely, parts of the behavior of one product are reused in the behavior of other products. As we have pointed out in the last section, especially in the automotive domain the operational functionality is appropriate in order to characterize and form reusable entities. With PF-CCS the desired reuse scenarios between model generations and model series can be realized and addressed. Parts of PF-CCS have already been published in [GLS08b].

- For PF-CCS we give a structural operational semantics (SOS) (cf. Chapter 3.2). The SOS semantics defines for every PF-CCS specification a corresponding *multi-valued modal labeled transition system*, which represents the operational behavior of an entire software product family in a single (multi-valued) transition system. For the purpose of constructing individual systems we show how to derive labeled transition systems which represent the operational behavior of individual products from the multi-valued transition system representing the entire product family. This gives an integrated view on the behavior of all products and allows to reason about the behavior of single derivable systems in the context of the *entire* product family. In particular, we can precisely express how the behavior of an individual system is related to the behaviors of other products of the product family. Due to its operational semantics, the PF-CCS framework is very close to an implementation and can be seen as an abstract implementation language which is suitable for the specification of the operational functionality of software-intensive, reactive systems.
- We define a restructuring concept for PF-CCS specifications and show its correctness (cf. Chapter 5). The concept allows to restructure the PF-CCS representation of a software product family in a way that individual products or sub-families are represented with a higher or lower degree of behavioral commonalities. In particular for alternative behavior (variation points) this means that we can restructure a PF-CCS specification into a form where its corresponding multi-valued labeled transition system explicitly shows their greatest (maximal bisimilar) common part and their differences. Seen in another context, this equals to compute the greatest common part of the behavior of a set of systems. We show the correctness of the calculation of the greatest common part using fundamental algebraic laws which we introduce, too. The

1. Introduction

restructuring concept is a new contribution which addresses the reuse challenges found for automotive systems. It has already been partially published in [GLS08a].

In summary, PF-CCS allows to specify the behavior of a set of systems in an integrated way as a product family. A PF-CCS specification is independent of a concrete implementation-platform, but due to its operational semantics still very close to an implementation—and thus very useful to guide the implementation of software-intensive automotive systems. PF-CCS differs from approaches such as feature models and traditional process algebras such as CCS or CSP, as it combines the modeling aspects which are essential for the specification of the operational functionality of a family of systems in an integrated way in a single theory. Typically, feature models allow the specification of variability, with the restriction that they lack a precise (operational) semantics. On the other hand, process algebras come with a precise semantical foundation but usually treat variability as a kind of non-determinism. In particular, the concept of variability as it is required by a software product line can not be implemented by such a kind of non-deterministic variability, as we will motivate in Chapter 3. Here, the PF-CCS theory addresses this situation by providing a mechanism to specify (behavioral) variability as it occurs in a software product family.

Verification of Software Product Line Specific Properties

Formal verification methods are of great interest in the area of reactive systems in general, but especially for the verification of the behavior of safety-critical systems as we find them for example in the automotive domain. For standalone systems without variable behavior there is already variety of logics and methods which support formal verification, e.g. *CTL* [EL86] and model checking [CGP99] being a common combination. However, in the context of software product families we typically have to deal with a large variety of systems, and thus with questions which do not arise during the construction of stand-alone systems. Typical questions are for example the existence of a configuration (representing a derivable product) fulfilling certain properties, the set of all such configurations, or the minimal amount of configuration steps which have to be taken in order to guarantee certain properties for the configured systems. With this thesis we contribute to the solution of such questions in the following way:

- We introduce a multi-valued version of the modal μ -calculus [Koz83] which is tailored to reason about PF-CCS product families. In particular, it allows to reason about the behavior of a product in the context of the behaviors of the other family members.

- We define a corresponding semantics (cf. Chapter 4.1.2). Formulae of the multi-valued μ -calculus are interpreted over multi-valued labeled transition systems, which we use to define the semantics of PF-CCS specifications. We call our version of the μ -calculus “multi-valued” since the interpretation of a formula over a given product family yields no longer a two-valued *true/false* result, but yields a *set of configurations*. This set characterizes exactly those products of the corresponding product family which obey the specified behavioral property.
- We prove that the result of evaluating a multi-valued μ -calculus formula over a PF-CCS product family coincides with the results of checking the same property on each of the systems individually (cf. Chapter 4.1.3). This means that the set of configurations which is the result of interpreting a multi-valued μ -calculus formula over a PF-CCS product family represents exactly those products for which the same formula evaluates to *true* when checking on the concrete representation of each product directly.
- We establish the connection between our multi-valued μ -calculus version and existing *multi-valued model checking techniques* (cf. Chapter 4.2). Thus, for the evaluation of formulae specified in our multi-valued version of the μ -calculus we can resort to existing, game-based model checking techniques.

1.4. Related Work

In the past years software product line engineering has gained notably attention in the academic community, and software product line concepts have been investigated in various areas of research and by various researchers and groups. Due to the practical relevance of software product line engineering the visibility in the industrial context has also constantly been increasing. Consequently, there is a large body of literature and publications in the area of software product line engineering on various kinds of topics.

In this thesis we consider on the one hand product families and the underlying construction principle in general, but on the other hand also the application of this general construction principle for a very specific kind of product families, where we are interested in the modeling and the verification of the behavior of a set of reactive, software-intensive systems as a product family. Instead of collecting the related approaches here—in a self-contained chapter outside an appropriate context—we discuss the related approaches within the appropriate context at the end of the respective chapters in the Sections 2.5, 3.5, and 4.4.

1.5. Thesis Outline

In Chapter 2, we elaborate characteristic properties and concepts of software product families. We define the concepts precisely in a rigorous mathematical way and axiomatize the notion of a software product family. The axiomatization represents the “essence” of any software product family and is the basis for a precise terminology for the successive chapters. As a counterpart to the axiomatization we illustrate all concepts informally using the example of a family of stickman drawings. In general, by skipping the mathematical definitions and just following the explanations and the running example, Chapter 2 is suitable for a reader who is not yet familiar with software product family concepts and who seeks an introduction. On the other hand, the axiomatization gives a mathematically formal view onto a software product family, which will be of interest in particular for the reader who seeks a basis for the application of formal methods in software product line engineering.

In Chapter 3 we introduce PF-CCS (*Product Family CCS*), a process algebraic framework for the specification of the operational functionality of similar systems as software product families. PF-CCS product families agree with the operations and postulates of the axiomatization of Chapter 2. PF-CCS is based on Milner’s process algebra CCS [Mil80], and extends it with the concept of variability as it is required for software product families. In particular, PF-CCS allows to model variable, alternative, optional and mandatory behavior. We define the syntax of PF-CCS (cf. Section 3.1) and develop an operational semantics (cf. Section 3.2) based on multi-valued labeled transition systems in an intuitive way. The specification framework PF-CCS is essential for the understanding of the successive Chapters 4 and 5.

Due to the semantics of PF-CCS, which is given in terms of multi-valued, modal transition systems, PF-CCS is well suited for the verification of behavioral properties. In Chapter 4, we introduce a multi-valued version of the modal μ -calculus as a property specification language for system families specified in PF-CCS. We introduce the semantics of our multi-valued modal μ -calculus, justify its “correctness”, and briefly show how adjusted multi-valued model checking techniques can be used for the evaluation of multi-valued modal μ -calculus formulae. PF-CCS is suitable to verify behavioral properties of an entire software product family, and to point out which variants of a software product family do not meet given behavioral properties. The discussion of all aspects related to the verification of PF-CCS product families is deliberately sourced out into an individual chapter, Chapter 4, however, the study of Chapter 3 is a prerequisite for understanding Chapter 4.

Chapter 4 requires fundamental knowledge of *lattices* and the modal μ -calculus [Koz83]. For readers which are not familiar with these topics we provide the relevant prerequisites in the Appendices C and D.

Due to the algebraic nature of a PF-CCS specification we identify certain calculation rules which allow to re-arrange a PF-CCS specification while preserving its semantics. Some of these rules are of particular interest for determining the commonalities of products. In Chapter 5, we will focus on the algebraic laws which allow to restructure a software product family in order to model its derivable products with a higher or lower degree of common behavior. We will show the application of these algebraic laws and illustrate an exemplary calculation with a brief example.

Finally, in Chapter 6 we recall all contributions of this thesis and discuss some concrete design decisions of the PF-CCS framework. This chapter explains why we have designed PF-CCS in the way we did. In addition, as part of our future work, we describe abstraction techniques for PF-CCS software product families, which we have already developed but which we will not introduce in the scope of this thesis.

The concepts and formalisms which we introduce in this thesis span several areas of research, for example algebraic specification, process algebras, labeled transitions systems, multi-valued modal logics, model-checking, lattices, etc. For some selected topics we briefly introduce the relevant prerequisites—as far as we require them for the scope of this thesis—in single chapters in the appendix, starting at Page 231. Note that we do not provide any new research results in these chapters, but merely recall existing fundamental knowledge.

Roadmap through the Thesis

In Chapter 3 we introduce the framework PF-CCS. For those readers who are interested in the specification of the operational functionality of a set of systems as a software product family, Chapter 3 is the right point to start with.

Knowledge of Chapter 3 is an essential prerequisite for Chapters 4 and 5. However, Chapters 4 and 5 are not related themselves, and can be read independently. The reader who is interested in the integrated verification of the behavior of a set of systems should read Chapters 3 and 4, while the reader who is interested in restructuring a software product family, and in the calculation of behavioral commonalities between the products, should consult Chapters 3 and 5.

In Chapter 2 we describe software product family concepts uncoupled of concrete formalisms or application areas both in a formal and an illustrative way. Chapter 2 can be read independently from all other chapters, however, we suggest to read it in connection with Chapter 3, since PF-CCS is an example of a concrete formalism that implements the axiomatization of Chapter 2. For the reader who is new to the area of software product families, by skipping the formal parts and just following the example and the intuitive explanations, Chapter 2 provides an illustrative, self-contained introduction to the area of software product families, which can be read independently.

Formalization of Characteristic Software Product Family Concepts: What Constitutes a Software Product Family?

Due to a wide range of application areas in which software-intensive systems are developed in the framework of a software product line engineering process, there is a magnitude of different notions of what a software product family precisely is. In this chapter we introduce fundamental concepts, operations, and properties of software product families which are independent of a concrete realization of the respective families. These operations embody a universal design/construction concept and represent the essential ingredients for dealing and reasoning about software product families, their products, and their commonalities from a constructional point of view. We define this universal design concept in a precise mathematical and axiomatic way by means of an algebraic specification. The resulting axiomatization gives a general characterization of the notion of a software product family. Moreover, it represents a standardization that establishes the formal basis to determine whether an approach is a valid realization of a software product family.

Contents

2.1. Software Product Families and Lines: An Informal View	23
2.2. Axiomatization of Software Product Family Concepts	30
2.3. Modeling Dependencies in Software Product Families	92
2.4. Discussion	97
2.5. Related Work	101

2. Formalization of Characteristic Software Product Family Concepts

Up to today, many companies have already successfully adopted a software product line engineering approach. Some prime examples, which we have already briefly described in Section 1.1, are the software product line¹ approaches realized by the companies *CelsiusTech AB* [BCKB03], and *Cummins, Inc.* [CN01], in which the software for a family of ships and diesel engines was developed as a software product line, respectively. A more recent example is a software product line approach for imaging equipment by *Philips Healthcare* [Pro99, SEI] where the software product line comprises systems that support medical diagnosis and interventions. These, and many other examples of successful software product line approaches, are documented as part of the *Software Product Line Hall of Fame* [SEI], to which new examples of successful, industrial scale software product line approaches are added every year.

Although the software product line community accepts all of these approaches as instances of software product lines, they exhibit fundamental differences: They differ for example in

- the kind of products that actually constitute the software product line,
- the kind of commonalities between the products,
- the kind and size of assets from which the products are assembled,
- the implementation platform on which the software product line is based,
- the abstraction level the software product line concepts actually applies to,
- the aims and the application area of the software product line,
- and the process in which the product line and the products are constructed.

In fact, the specific realizations of the concept *software product line* vary from company to company, span various abstraction layers in the development process, and make a comparison between the different approaches very difficult. Nevertheless, the software product line community considers all these different approaches as “realizations” of software product lines. In this light, the following questions arise naturally: Why are the approaches of these companies accepted to be realizations of software product lines? What actually are the software product line specific concepts these companies used in their approaches? Ultimately, these questions reduce to the more fundamental question: what actually makes a software product line a software product line?

¹In compliance with the terminology used in the respective examples we also speak of a software product line instead of a software product family. However, note that this imprecise terminology is only used for the beginning of this section up to Section 2.1, where we explain the difference between both terms in detail.

2.1. Software Product Families and Lines: An Informal View

In the remainder of this chapter we discuss this question and characterize the notion of a software product family formally. Based on common definitions of a software product line and a software product family, we (informally) collect and discuss the important concepts behind the constructional aspects of software product line engineering, make a distinction between a software product family and a software product line, but most importantly provide a comprehensive axiomatization (cf. Section 2.2) that defines a software product family and its concepts in an implementation-platform independent, mathematically precise way. The axiomatization is a property-oriented specification which gives a rather operational than denotational characterization of software product families. In particular with respect to (i) the construction from products by explicitly using commonalities, (ii) the operational handling of the commonalities of products, and (iii) the formal derivation of other laws that hold for product families, such a property-oriented characterization is the basis to reason about software product family concepts on a formal level independently from the concrete realization of the software product family.

In addition, the axiomatization provides a formal basis to determine whether an approach is a valid realization of a software product family. In particular with respect to Chapter 3, where we introduce a framework based on the process algebra CCS [Mil80] for the specific purpose of specifying the behavior of a family of similar systems, the axiomatization is the basis to show that this particular CCS-based realization indeed allows the specification of software product families that fulfill the axioms. Moreover, the axiomatization gives us the formal basis to reason that the CCS-specific realization of Chapter 3 is only one possible choice, that the CCS-specific concepts and mechanisms are not essential in order to realize the product family concepts, and that other realizations (for example based on other process algebras like CSP or ACP) are equally possible.

2.1. Software Product Families and Lines: An Informal View

Despite their different realizations, the initially mentioned examples of successful software product line approaches are based on a common philosophy. At its heart is the idea that a set of similar systems can be constructed more efficiently if these systems are developed in combination rather than independently from each other, taking advantage of the commonalities among the systems in a planned way. This idea is fundamental to all popular characterizations of software product lines. It is for example expressed in the well-established definition of a *software product line* given by Paul Clements and Linda Northrop:

Definition 2.1 (Software Product Line [CN01]). *A software product line is a set of software intensive systems sharing a common, managed 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).*

2. Formalization of Characteristic Software Product Family Concepts

A similar characterization of *program families*—coined already in the 1980s and therefore related more closely to programs rather than to systems—which also embodies this fundamental idea of commonality is given by David L. Parnas:

Definition 2.2 (Program Family [Par76]). *We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.*

Commonalities within a Set of Systems

Central to both definitions, Definition 2.1 and Definition 2.2, is the aspect of commonalities between systems (respectively programs). In particular for software-intensive systems we observe many kinds of commonalities, some examples are:

- A trivial—and for most people probably not very important—commonality between systems is the number of lines of code, or the space in terms of memory which is necessary to store or execute the program code of individual systems.
- A more interesting commonality is whether systems with completely different functionalities and different implementations have the same conceptual component structure or architecture. For example, an online-banking system and a system realizing a webshop, which both operate in a distributed way via the Internet, can share the same client-server architecture, although they are used for different purposes and exhibit different functionalities.
- Systems have commonalities with respect to their realization, comprising for example the choice of a specific software/hardware platform, middleware, programming language, or program library. A typical example of such a kind of commonality is that different systems use the *Java Enterprise Edition* [Ort09] as their realization platform.
- Systems have commonalities regarding their functionality, i.e. their black-box functionality, but especially also their operational behavior. For example, two systems that produce a sorted sequence can use different sorting algorithms, e.g. *Quicksort* and *Mergesort*, to actually sort the sequence. Another example is the *eCall*-function [Com09], which is a function that is (compulsorily) common to all modern cars produced in certain EU countries. It is implemented differently across different car models, but exhibits the same observable behavior that an automatic emergency call is always initiated in a specific way in the case of a crash. As we will see in the Chapters 3–5, in particular the *commonalities in the (operational) behavior* of systems will be central to this thesis, and in the remainder of this thesis we will introduce a framework for the specification of the operational functionality of set of systems as a product family, which allows to model and to reason about behavioral commonalities.

2.1. Software Product Families and Lines: An Informal View

- Another commonality between systems is the way, e.g. the development process or the methodology, in which they are developed. For example, systems may be developed according to the *V-Model XT* [BR05], *Extreme Programming* [Bec00], and may for example also comprise the methodical concept of applying *unit-tests*.
- Systems have commonalities with respect to non-functional aspects such as quality attributes, e.g. the degree of maintainability, or the fact that systems are developed by the same company or even the same team of developers. Such kinds of commonalities are for example interesting with respect to the product costs, the estimated makespan of a system, or even the quality of the systems.
- From a marketing point of view, an important commonality of systems is the market segment together with the target group, at which the systems aim. Such economic commonalities are for example important for the sales or administration process which is associated with the production process.
- Finally, an important commonality between the systems produced by the same company is the branding of all systems with the company's corporate identity. For software-intensive systems a prominent example is the uniform look and feel throughout the product range of *Apple* products, especially within the series of *Mac OS* operating systems [App10]. Car manufacturers also strive for branding their products with a distinctive company image by designing for example the user interface in a company-specific way which creates a uniform usability experience throughout the entire model range of the manufacturer.

Regarding the integrated development of a set of similar systems, commonalities between the systems can be incorporated into the system development process to different degrees, at different stages, and for different purposes. At a first level, consider a company which develops a set of products which happen to have certain commonalities. Here, the commonalities between the systems are neither initially planned nor systematically exploited, and the systems are developed independently from each other with respect to the commonalities. Such a scenario is typical for systems which are originally planned as single systems only, and from which several variants are derived in the course of time in an ad-hoc way, e.g. by “copy-paste” operations.

At a second level, a company realizes that there are certain commonalities within the set of systems the company produces. In order to increase the efficiency in the actual production process, the company takes explicitly advantage of the commonalities between the systems in a planned way right from the beginning of the development process. Thereby, the commonalities can contribute differently to increase the overall development efficiency. For example, some commonalities are used to increase the efficiency of the sales process, some are beneficial to decrease the production costs, and some increase the efficiency of the actual construction of the products. For

2. Formalization of Characteristic Software Product Family Concepts

software-intensive systems, especially those commonalities are interesting which can be used to increase the efficiency in the *construction* of the systems. More precisely, for the construction of each system, those parts which the system has in common with other systems, are explicitly determined and used in the assembly of each system. Compared to the first level, this represents a *methodological* exploitation of the commonalities between systems. In this context, i.e. if we consider a set of systems from the point of view of *constructing* each system by making use of common parts, we also speak of a *family of systems*.

At a third level, the methodological exploitation of commonalities for the construction of the family members—and the resulting efficiency increase—can even affect the development of new systems, which are no family members so far. For example, imagine a company X which produces a family of engine control units for different types of car engine systems, by explicitly making use of commonalities for the construction. In the situation where company X gets an order by customer Y to produce a new system which happens to be similar to one of X 's existing family members, company X might even offer its customer Y to slightly modify its original requirements for the new system in a way that it can basically be realized as a member of the existing family. In this case, company X can produce the new product more efficiently with little additional effort (by making use of the commonalities with other family members), and thus offer it to a lower price to customer Y .

Software Product Families

Based on this consideration of the kinds of commonalities and the way in which commonalities are used for the development of a set of systems, we characterize a *software product family* (informally) as shown in the following definition. A formal, precise definition is based on the axiomatization and will be given in Definition 2.10 in Section 2.2.4 on Page 89.

Definition 2.3 (Software Product Family (Informal Characterization)). *We call a set of systems a software product family if the systems are constructed in an integrated way as members of the family by explicitly making use of the commonalities between the members in a systematic manner.*

With this definition we take over main ideas of the prominent definitions of Clements/Northrop and Parnas (Def. 2.1 and 2.2), but highlight the aspect of being constructed in an integrated way, based on the exploitation of inter-product commonalities. In particular, for a software product family it is essential that commonalities between systems not only exist, but that these commonalities are explicitly used in a systematic way for the *construction* of each system. This makes a software product family a methodical concept, i.e. a software product family emphasizes the *methodical* exploitation of commonalities for the construction of the products of a software product family.

2.1. Software Product Families and Lines: An Informal View

But how can commonalities explicitly be used for the construction of individual products? How has the notion of commonality to be understood at the level of constructional units from which each system is assembled? How are such commonalities actually modeled? These questions address the architecture and structure of the software product family and its members, and have to be answered at the level of the construction units from which the products are assembled. For our approach we take a constructive view on a product family and its products, and characterize a product family by means of atomic construction units and functions that describe how the respective products are constructed. At this level, constructing a set of systems in an *integrated* way means to explicitly handle the commonalities and differences between the systems. Rather than specifying common parts, we specify the points in which the systems differ, and use these differences to indirectly define the commonalities, and to reason about them.

From this point of view, a software product family is more than just a set of (unrelated) products. It is a set of products represented in an integrated, combined way, which exactly defines how each product is constructed from a set of construction units and (composition) functions. This integrated representation is basically a kind of construction plan of the software product family and its products. Since such a construction plan is essential for a product family, we usually do not distinguish between the software product family itself, i.e. the set of products, and the construction blueprint/representation and refer to both by the term *software product family*. As we will see in Section 2.2, we specify constructional units, the operations to assemble the family members from constructional units, and the operations to derive individual family members from the representation of a software product family in a mathematical way by means of an axiomatization. This formalism allows us to define the concepts of a software product family in a systematic way, as well as to reason about properties of a software product family and its products.

Software Product Line vs Software Product Family

Recall the different kinds of commonalities which we have identified at the beginning of this chapter (Page 24). While many of them can be used to increase the efficiency in the *construction* of a set of products, some of them rather improve the efficiency of other aspects which are associated with the commercial production of software-intensive systems. For example, the commonality of addressing the same target group in the market rather increases the efficiency of the sales process than the efficiency in the product construction. In this light, many commonalities which are irrelevant for the construction of the products can be used to increase the efficiency of the entire product development. In particular, beside the constructional aspect, increasing the efficiency by exploiting such commonalities is an essential factor for the success of a software product *line* as a commercial/marketing concept.

In this thesis we only consider to increase the efficiency in the *construction* of products. According to the software product line community [CN01], this is only one

2. Formalization of Characteristic Software Product Family Concepts

specific aspect which is usually considered in the context of a software product line. For the differentiation from our notion of a software product family which we have given in Definition 2.3, and the general idea of a software product *line*, as it is for example emphasized in the definition of Clements/Northrop (Definition 2.1), we follow Parnas. At his keynote talk at the Software Product Line Conference 2008 in Limerick [Par08] he drew a clear distinction between the engineering concept of a *program family*, which he has coined in the 1970s, and the economic/commercial concept of a *software product line*. He pointed out that (i) a *program family* actually is intended to be an engineering concept which deals with the constructional aspect of software product line engineering, (ii) and that an explicit differentiation between the economic aspect of software product line engineering, and the technical aspect of how each product is constructed, is essential. For the scope of this thesis, we take over this differentiation in the same way to distinguish between our notion of a software product family and the wider concept of a software product line. Thereby, we do not want to give a precise definition of a *software product line*. We merely want to separate between the two notions in order to restrict the scope of this thesis clearly to the constructional aspect, i.e. to software product families.

Both terms, *software product line* and *software product family* refer to a set of software-intensive systems which are developed as part of a software product line engineering process. *Software product line engineering* denotes a software development paradigm which is directed to the efficient construction of a set of software products for a particular market segment or mission. However, the terms *software product line* and *software product family* highlight different aspects of software product line engineering: an economical/holistic and a constructional/engineering aspect.

When speaking of a *software product line*, we emphasize the economic/commercial aspect of software product line engineering. The term *software product line* denotes a set of (software) products which are offered to a customer. Here, *product* actually refers to a commercial product, i.e. a commodity, and *product lining* denotes the marketing strategy of offering several products with commonalities to a customer for sale. With the marketing concept of a product line a company seeks more than just to improve the efficiency in the production process. As the example of a unique product branding within the product range of a company shows, the advantages of offering a set of products as a product line can be beyond constructional efficiency reasons. In particular, the definition of a software product line given by Clements and Northrop (Def. 2.1) emphasizes these kinds of economic aspects.

The scope of the software product line and the choice of products which are offered as part of a product line are driven by economic aspects based on a market analysis. Typically, such economic aspects are for example the quantitative improvement in productivity, time to market, product quality, higher customizability, and customer satisfaction. Beside these aspects, bringing a software product line successfully to market strongly depends on other entrepreneurial and organizational factors such as

2.1. Software Product Families and Lines: An Informal View

- the initial cost overhead for developing and establishing the software product line and the corresponding risk analysis,
- the realizability within a given organizational structure,
- an adequate production and development management which allows to plan and keep track of the products and variable parts (, in particular concerning the question of how the reusable assets are made available within the company),
- the system landscape and infrastructure,
- training costs, and finally
- an adaption of the entire company philosophy.

Thus, building a successful software product line and establishing it in the market requires more than the technical expertise to actually construct a set of products as part of a software product family, it also requires skillful economic and organizational management. A major share of the entire research carried out in industry and academia is dealing with the economic aspects of software product line engineering.

However, both aspects, the economic and the constructional one, are frequently confused and mixed. After the preceding discussion, we fix the following picture for the scope of this thesis: In general, the products offered as part of a software product line do not necessarily have to have any commonalities concerning their constructional/technical realization. Actually, they can be constructed as part of different software product families, or even as independent systems. Vice versa, systems which are constructed from the same software product family can be offered as products by several software product lines. However, ideally, for an efficient construction of the products of a software product line, they should be part of the same software product family. Thus, the degree of efficiency in which a software product line can be realized depends strongly on the degree to which the products of the software product line are actually members of the same product family.

The remainder of this thesis is devoted exclusively to the engineering aspect of how to *construct* a set of systems as a software product family. In particular, we will *not* deal with economic, marketing or “non-constructive” aspects of software product line engineering and questions related to software product *lines*. In the light of the preceding discussion this means that we investigate and precisely define what a software product *family* is.

2.2. Axiomatization of Software Product Family Concepts

The aim of a software product family is to guide the *construction* of a set of systems by describing how each system can be constructed in a systematic way that explicitly makes use of the commonalities with the other systems. Thereby, it is not important whether we consider a product family of code fragments, conceptual component structures, functions, or processes, since the basic concepts of constructing systems as part of a software product family are always the same and independent of its specific realization. In particular, the act of constructing the systems can be described by certain realization-independent operations and laws which characterize the way in which the operations work. Thus, from a mathematical point of view we can see a software product family as an *algebraic structure*, i.e. an *algebra*. Since we emphasize in particular the laws, i.e. the axioms, which define the interplay between the operations of such an algebraic structure, we also speak of an *axiomatization*.

We formalize software product family concepts as an axiomatization since we are interested in a property-oriented, formal specification of the conceptual idea behind software product families, which is independent of the specific implementation and realization of a concrete software product family, and which characterizes the entire class of software product families. Such an axiomatization is the formal basis to reason about properties that universally hold in the class of product families. In particular, it comes with a profound body of proof and derivation techniques, that allow to conclude new laws and theorems. In addition, an axiomatization specifies the product family concepts in terms of the characteristic operations and laws that are typical for any software product family. Further reasons for an axiomatization/algebraic specification are given at the end of this chapter in Section 2.4.1.

We introduce the axiomatization, i.e. its sorts, operations and axioms, step-by-step and explain every concept in detail. Finally, this results in an exhaustive algebraic specification of the abstract computation structure of software product families, which is shown in Figure 2.11 on Page 90.

As a counterpart to the formal specification we illustrate the operations and axioms in an intuitive way using the example of a family of stickmen drawings. Figure 2.4a (Page 42) shows some examples of stickmen. We construct the stickmen drawings as a product family, i.e. we assemble each stickman from a set of atomic shapes according to the construction blueprint which is specified by the product family. Although stickmen drawings are not software artifacts at all, technically, the product family of stickmen is one example of a computation structure for the sort $\text{SPF } \alpha$.

Before we introduce the algebraic specification of software product families in Sections 2.2.2–2.2.4 we briefly recapitulate the preliminaries on algebraic specifications in the following section.

2.2.1. Preliminaries: Algebraic Specification

The axiomatization itself will be given by means of an *algebraic specification*. With an algebraic specification we have chosen a specification technique that allows to (formally) characterize a *class of computation structures*, i.e. in our case the class of software product families. Following mainly [Wir90], we sketch the main concepts of algebraic specifications very briefly in this section. For a detailed introduction of the necessary concepts and the general idea of algebraic specifications we refer to [Wir90, EM85]. Note that syntactically we follow [Bro98, BFG⁺93a, BFG⁺93b] to denote the algebraic specification.

With the algebraic specification of a software product family we specify the sort SPF α which axiomatizes the operations and laws which are characteristic of software product families. In general, an algebraic specification (Σ, Eq) consists of a syntactic signature Σ and a set Eq of equational axioms. The (*syntactic*) *signature* $\Sigma = (S, F)$ consists of a set S of *sorts* (*sort names*) and a set F of many-sorted *function symbols* together with a mapping *type* : $F \rightarrow S^* \times S$ which associates sorts to the function symbols. Given the signature $\Sigma = (S, F)$ we call any algebra A a Σ -*algebra* if it consists of a family $\{M_s^A\}_{s \in S}$ of non-empty carrier sets M_s^A , and a family of functions f^A , such that A associates a carrier set M_s^A to every sort $s \in S$, and a function $f^A : M_{s_1}^A \times \dots \times M_{s_n}^A \rightarrow M_{s_{n+1}}^A$ to every function symbol $f \in F$, where *type*(f) = $(s_1 s_2 \dots s_n, s_{n+1})$. We denote the class of all Σ -algebras by $ALG(\Sigma)$. Every Σ' -algebra with a conformant signature $\Sigma = \Sigma'$ which fulfills the equations Eq of an algebraic specification is called a *model* of the algebraic specification (Σ, Eq) .

A Σ -*homomorphism* relates the sorts and functions of two Σ -algebras, which usually operate on different carrier sets, in a way that the operations are preserved. Formally, a Σ -homomorphism $h : A \rightarrow B$ between two Σ -algebras A and B is a family of maps $\{h_s : A_s \rightarrow B_s\}_{s \in S}$ such that for each $f : s_1, \dots, s_n \rightarrow s_{n+1} \in F$ and each $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$ we have $h_s(f^A(a_1, \dots, a_n)) = f^B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$.

Every signature Σ induces a set of terms $Terms(\Sigma, X)$ which can be formed from the function symbols and a family X of S -sorted, free variables. Terms without variables of X are called *ground terms* and will be denoted by $Terms(\Sigma)$. For any signature Σ , a fundamental Σ -algebra is the so-called *term algebra* $T(\Sigma, X)$ which has as carrier sets for each sort $s \in S$ the set of terms $Terms(\Sigma, X)_s$, and in which we associate every function symbol $f \in F$ with a corresponding function $f^{T(\Sigma)}(t_1, \dots, t_n) =_{\text{def}} f(t_1, \dots, t_n)$ for each $f : s_1, \dots, s_n \rightarrow s_{n+1} \in F$ and $t_1 \in Terms(\Sigma, X)_{s_1}, \dots, t_n \in Terms(\Sigma, X)_{s_n}$, with the meaning that the evaluation of $f^{T(\Sigma)}$ at (t_1, \dots, t_n) is the term “ $f(t_1, \dots, t_n)$ ” $\in Terms(\Sigma, X)$. Thus, the functions in the term algebra represent the (syntactic) construction of terms over signature Σ . If the terms are ground terms only, we call the term algebra the *ground term algebra* and write $Terms(\Sigma)$.

2. Formalization of Characteristic Software Product Family Concepts

For every Σ -algebra A , there exists a Σ -homomorphism $v : T(\Sigma, X) \rightarrow A$ which associates terms with the elements of A , which is called *interpretation*. For the ground term algebra, this Σ -homomorphism is uniquely defined, and is called *term interpretation*. Every Σ -algebra A for which all elements of its carrier sets can be denoted by terms over $T(\Sigma)$, i.e. where the corresponding Σ -homomorphism $v : T(\Sigma) \rightarrow A$ is surjective, is called a Σ -*computation structure*, or *computation structure* for short. In this case, we also say that the Σ -algebra A is *term-generated*. Since the homomorphism is surjective, the term interpretation induces an equivalence relation on the set of terms. Every Σ -computation structure is isomorphic to the ground term algebra modulo this equivalence relation on the set of ground terms. Practically, for the scope of this thesis, computation structures for the sort $\text{SPF } \alpha$ represent software product families for specific purposes in a form which we can actually use to work with.

The relation “there exists a Σ -homomorphism from Σ -computation structure A to B ” defines a partial order on the set of all computation structures. A so-called *abstract computation structure* is a class of Σ -computation structures which are isomorphic w. r. t. this relation. Classes of isomorphic computation structures form a lattice under this partial order. In summary, with the algebraic specification of the sort $\text{SPF } \alpha$ we characterize the abstract computation structure of software product families, i.e. the class of all Σ -computations structures which fulfill the axioms required by the algebraic specification.

2.2.2. Operations for Constructing a Software Product Family

In Definition 2.3 we have already given an informal characterization of a software product family. Central to this definition is the idea that a software product family represents the entirety of a set of similar systems, and at the same time incorporates the knowledge of how every single product is precisely constructed. In order to achieve this, the representation of a product family has to comprise the information of

- the common parts of the products,
- the product-specific, variable parts, and
- the information of how these parts have to be combined in order to construct each of the products.

We realize this by representing an entire software product family as a single term of sort $\text{SPF } \alpha$. The sort $\text{SPF } \alpha$ is a *parameterized sort* whose elements represent software product families defined over the generic sort α . A parameterized sort

specification offers an abbreviatory notation technique which allows to represent a set of concrete specifications in a comfortable way. In order to obtain a concrete sort (specification) we have to replace the generic placeholder α with a suitable actualization, i.e. a concrete sort such as for example a sort representing Java software components. Depending on the choice of α , the parameterized sort $\text{SPF } \alpha$ represents many kinds of actual software product family sorts, e.g. the sort of software product families of Java software components, the sort of software product families of AUTOSAR components [GBR08] running on ECUs (*electronic control units*) in an automobile, the sort of software product families of functional units represented by Simulink [Inc09] models, or the sort of software product families of implementation independent components specified by I/O-automata or process algebraic terms. However, the product family concepts are independent of the concrete choice of α .

Every element of sort $\text{SPF } \alpha$ can be represented by means of four functions: `asset`, `||`, `⊕`, and `ntrl`. In particular, nesting and combining these four functions in an algebraically correct way always yields an element of sort $\text{SPF } \alpha$. In the context of an algebraic specification such functions are called *constructors*. We introduce these four constructors in the following Sections 2.2.2.1–2.2.2.3.

Note that the (term) structure which we can build by composing these constructors is basically that of a binary AND/OR tree [LS93, Sch01], if we interpret the composition operations as *AND*-nodes, and the variants operators as *OR*-nodes. While an AND/OR-tree is a structure that is commonly known, we have not simply used this structure for the specification of software product families, since (i) there is no set of commonly accepted laws for AND/OR-trees which allow a restructuring of such a tree, similarly to the way in which we use the axioms for the constructors to restructure the term representation of an element of sort $\text{SPF } \alpha$, and (ii) since we are in particular interested for our considerations in the operations such as selection, determination of mandatory parts, etc., which are not predefined in the context of an AND/OR-tree, either.

2.2.2.1. Core Assets and Neutral Element

Core Assets

All systems which are defined by the product family are assembled from a common set of components/constructional entities. The atomic components are the basic building blocks of a product family, which are not decomposed further in the context of the product family. In the context of a software product family we call such basic components (*core*) *assets*. The common set or pool of all core assets is called the *universe*. Since the core assets are the smallest entities of composition, commonalities between the products are expressed in terms of common assets and asset-structures. Thus, in a component-based approach where we can associate

2. Formalization of Characteristic Software Product Family Concepts

commonalities to atomic assets, we can reason about commonalities at the level of asset structures. In particular, with knowledge of the construction blueprint of each product, we can reason how certain commonalities are present in each product on basis of the assets that exist in the product (cf. Section 2.2.3.7).

We represent a core asset by a parameterized function `asset` with the following signature, where α is a placeholder for an arbitrary sort:

$$\text{asset} : \alpha \rightarrow \text{SPF } \alpha \tag{2.1}$$

The function basically wraps an element of the generic sort α and yields an element of sort `SPF α` . Illustratively, for an $a \in \alpha$, the term `asset(a)` represents an elementary building block with the name a . While technically, such a wrapping is necessary in order to realize a set of type-compatible functions, semantically the wrapping means to abstract from all sort-specific properties and laws that hold for elements of sort α . This means that we use elements of the sort α simply to distinguish between different kinds of assets in a more comfortable way, rather than to define several different (constant) asset functions directly for the sort `SPF α` .

Note that with a single atomic asset we can already construct a software product family. Formally, for any $a \in \alpha$ the term `asset(a)` represents already a software product family. Obviously, a single asset represents a quite trivial software product family—as it only allows to derive the single product `asset(a)`—but it is still a valid product family in our sense.

Neutral Element

Beside atomic assets, there is another constructor for the sort `SPF α` , called `ntrl`. Similarly to the constructor `asset`, which has the character of a constant as it does not have an argument of sort `SPF α` , the function `ntrl` is a constant with the signature

$$\text{ntrl} : \text{SPF } \alpha \tag{2.2}$$

The constructor `ntrl` is a special kind of element which is used to explicitly model the *empty product family*. As we will see in Section 2.2.3.8, the neutral element and its meaning is essential to model the idea of optional parts.

2.2.2.2. Composition

Every single product of a product family is constructed by assembling the necessary subset of core assets in an appropriate way. This immediately requires that we have a notion of *composition*. We model the composition in the context of a product family as a function

$$\parallel : \text{SPF } \alpha, \text{ SPF } \alpha \rightarrow \text{SPF } \alpha \quad (\textit{infix}) \quad (2.3)$$

The operation \parallel is a binary function which allows to compose two elements of type $\text{SPF } \alpha$, yielding again an element of the same type, i.e. a product family. We call \parallel the *composition operation* and usually denote it using an infix notation. An element which is the result of a composition is called a *compound element*. Note that every *product* of a product family is a compound element in the sense that it can be represented by a term which is a composition of core assets, only. As we will see in the following Section 2.2.2.3, variation points themselves are also of sort $\text{SPF } \alpha$. For the composition this means, that we can combine assets, variation points, and compound elements in a hierarchical way yielding composed objects of sort $\text{SPF } \alpha$.

The composition operation—as we define it for the sort $\text{SPF } \alpha$ —abstracts from the information of how the composition is actually realized by the underlying computation structure, and even by the specific sort α . It just defines the elements of sort $\text{SPF } \alpha$ which are combined, forming a compound, more complex element which again is of the same type $\text{SPF } \alpha$. The actual meaning of the composition operation is specific for the individual computation structure. For example, the composition for a product family $\text{SPF } \text{JAVA}$ of Java [GJSB05] code structures (of sort JAVA) could be realized by *superimposition* [ALMK10], or the standard Java-inheritance mechanism [GJSB05], while the composition in a product family $\text{SPF } \text{CSP}$ of CSP processes [Hoa85] can be realized as a parallel execution of processes.

2.2.2.3. Variation Points and Variants

As we have already stated in the introduction, a software product family is always a component-based system. The two functions `asset` and \parallel are essential for every component-based system since they realize the concept of assembling a product from smaller parts. However, in contrast to the “traditional” idea of component-based systems (cf. Section 1.1), a software product family additionally comprises the notion of variability, realized by the concept of alternative and optional parts. It is only by this apparently little extension that a software product family achieves the

2. Formalization of Characteristic Software Product Family Concepts

kind of reuse which was striven for (but not achieved) by all preceding component-based approaches.

In general, *variability* denotes the changeableness and the diversity in a set of distinguishable entities. Especially in the context of a software product family, *variability* denotes the variation between individual products. Since we are interested in the construction of the products, i.e. in how the products are actually assembled, modeling variability in a software product family essentially means to specify

- the points in which individual products differ,
- and the actual differences, i.e. the concrete instances/values for the different choices.

The function \oplus realizes these two requirements. It makes the variability in a software product family *explicit* by introducing the notion of *variation points* and *variants*.

$$\oplus : \text{SPF } \alpha, \text{ SPF } \alpha, \text{ NAT} \rightarrow \text{SPF } \alpha \quad (2.4)$$

We call the function \oplus the *variants operator*. It is a function that represents a point (in the specification of a product) which requires the alternative selection between its two arguments of type $\text{SPF } \alpha$. We call such a point a *variation point* and its arguments *variants*. If both variants of a variation point are equivalent (with respect to a certain equivalence relation), we call the variation point *trivial*. We use a mixfix notation for the variants operator, i.e. usually we write

$$V_1 \oplus_n V_2$$

instead of

$$\oplus(V_1, V_2, n)$$

to denote a variation point with the variants V_1 and V_2 .

Since a variation point is of sort $\text{SPF } \alpha$, variation points can be used like assets or composed objects as “regular” construction entities. Illustratively, a variation point represents a kind of placeholder or “hole” in the blueprint of a product which is “filled” with exactly one of its variants when constructing any product of the software product family. The meaning of a variation point $V_1 \oplus_n V_2$ is that in any product only one of its two variants V_1 or V_2 will be present. In particular, no product of the entire product family will contain both variants simultaneously. The selection between the variants is made *deterministically* according to a predefined configuration. For the term representation this means that the entire term representing a variation point (in the software product family) will be replaced by the subterm which represents

2.2. Axiomatization of Software Product Family Concepts

the selected variant (in one of the concrete products). We discuss the concept of configuration and the meaning for the term representation in more detail in Section 2.2.3.3.

Compared to traditional component based approaches, the explicit representation of variation points as constructional entities is a key idea of *strategic reuse*: in a software product family we not only collect the reusable elements (as also done in a traditional component based approach), but we also specify exactly the way *how* and the position *where* an element has to be integrated into the respective product, if the element will be actually present in the product.

The respective act of selecting for a variation point the variant which shall exist in the final product is called *configuration process*. In order to keep track of the information which variants are chosen for which variation points we use the concept of a *configuration*: A *configuration* is any mapping which represents the information which variants are chosen for which variation points. If a configuration associates one variant to *every* variation point of the software product family, it identifies exactly one product in a unique way.

We require that the configuration decision can be taken separately for every unrelated, individual variation point. This implies to have a unique reference for every variation point. Thus, whenever a new variation point is added to a product family, it is labeled with a fresh identifier. Here—similarly to the idea of a fresh variable in a logical formula—fresh means that the identifier is not used so far as the label of any other variation point which is already included in the product family. In order to check the freshness we provide a function `has_ntVP` which is also precisely defined in an algebraic way by means of axioms. However, for didactic reasons we will defer the corresponding discussion of the function `has_ntVP` together with the introduction of the corresponding axioms to Section 2.2.3.9.

Further, note that due to the distributive law (Axiom A-3, Page 90) we can transform the term representing a product family in a way that the same variation point (with the same identifier) may appear multiple times within the term. In such a situation, the term representation can actually contain multiple variation points with the same name. However, this is not problematic as we will precisely discuss in Section 2.2.3.9, and can be ignored for now.

The (fresh) identifier represents the third argument of the respective variants operator. It is of type `Nat` (representing the natural numbers). We have chosen the labels to be of sort `Nat` for the sake of simplicity, since it (i) provides a sufficient amount of different elements in order to label the variation points consecutively, and (ii) it eases the technical treatment of dealing with variation points in an algebraic specification. However, any other sort which provides a sufficient amount of different elements, e.g. a sort representing sequences of characters, is suitable, too. Note that

2. Formalization of Characteristic Software Product Family Concepts

similarly to the sort $\text{SPF } \alpha$, also the sort Nat is specified by means of an algebraic specification which can be found in Appendix A (cf. Page 233).

The variants operator—as it is introduced in Equation 2.4—allows to model the direct choice between (only) two alternative variants. Certainly, in a realistic context we can easily think of situations which require to express a choice between more than two, say n , variants. However, important for modeling a software product family is only the fact that a corresponding variants operator models the choice of exactly one variant out of a (well-defined) set of (countably) many variants. In fact, conceptually there is no difference between a variants operator which provides the choice between two alternative variants and a general version which offers a direct choice between another (finite) number of alternative variants. Nevertheless, a general variants operator—offering a choice between more than two variants—comes with some technical overhead concerning its algebraic specification. Thus, for the sake of simplicity we will defer a precise definition of a general variants operator and the corresponding discussion to Section 2.2.3.10, and introduce the following axiomatization with a binary variants operator, only.

However, in some of the examples we will use the general version of the variants operator already prior to its formal definition. We denote it in an mixfix-style where the possible variants are represented as a finite sequence (denoted by $\langle \dots \rangle$ -brackets), and the unique identifier of the variation point is denoted as a subscript to the operator symbol. We write for example

$$\oplus_{42}(\langle V_1, V_2, V_3, V_4 \rangle)$$

to represent a general variation point with the unique identifier 42 that offers an alternative choice between the four variants V_1, V_2, V_3 , and V_4 .

Since a variation point gives reason to different products, i.e. one product which contains the one variant, and another product which contains the other variant, every software product family naturally induces a set of products, given by all combinatorially choices for its variation points.

Note that for the entire axiomatization we do not provide a (sub-) typing concept, i.e. we do not sub-divide the elements of sort $\text{SPF } \alpha$ into sub-sorts, as it is commonly done [LW94] for example for the data types of object-oriented programming languages. This means that the kind of the variants of a variation point cannot be further restricted in the sense that both its variants must be elements of the same subtype. Although this does not affect the conceptual construction principle behind a variation point, it still has effects for some practical questions with respect to the notion of mandatory parts of a software product family. Since a more detailed discussion of this topic requires the definition of functions that are still to be introduced in the following, we defer a detailed discussion to Section 2.4.3, and the end of Section 2.2.3.8.

2.2.2.4. Example: A Product Family of Stickmen Drawings

As a running example, we describe the software product family concepts which we introduce in this chapter with an illustrative example of a concrete product family. We use a slightly unconventional running example: we illustrate the concepts with a computation structure of stickmen drawings, which are no software-artifacts at all. Although the example might not seem to be appropriate at a first glance, it is very suitable to visualize the concepts and axioms in a graphical way, which greatly eases the understanding of the concepts, especially for someone who is not yet familiar with algebraic techniques and methods. In addition, it underlines the fact that a product family constitutes a construction concept which is independent of the kind of its assets (i.e. the sort α) and the way how its assets are actually constructed.

Formally, we introduce the sort **Stickman** in order to represent stickmen drawings by terms. The algebraic specification of sort **Stickman** is shown in Figure 2.1. For the scope of this example it is sufficient that the sort **Stickman** consists merely of a set of constants (nullary functions), as we use these constants only to represent the graphical shapes with the same name. Technically, by instantiating the parameterized sort **SPF** α with the sort **Stickman** we can represent product families of stickman drawings of sort **SPF** **Stickman**.

```

SPEC StickmanSpecification = {
defines sort  Stickman

    Face      : Stickman
    Legs      : Stickman
    Smiling   : Stickman
    Sad       : Stickman
    Normal    : Stickman
    Female_Torso : Stickman
    Male_Torso : Stickman
    Coffee   : Stickman
    Left_Hand : Stickman

    Stickman generated_by  Face, Legs, Smiling, Sad, Normal, Female_Torso
                           Male_Torso, Coffee, Left_Hand
}

```

Figure 2.1.: Algebraic specification of the sort **Stickman**.

Computation Structure of Stickmen Drawings

In order to illustrate the terms of sort **SPF** **Stickman** graphically, we use the concrete computation structure of stickmen drawings as an interpretation of the terms of sort

2. Formalization of Characteristic Software Product Family Concepts

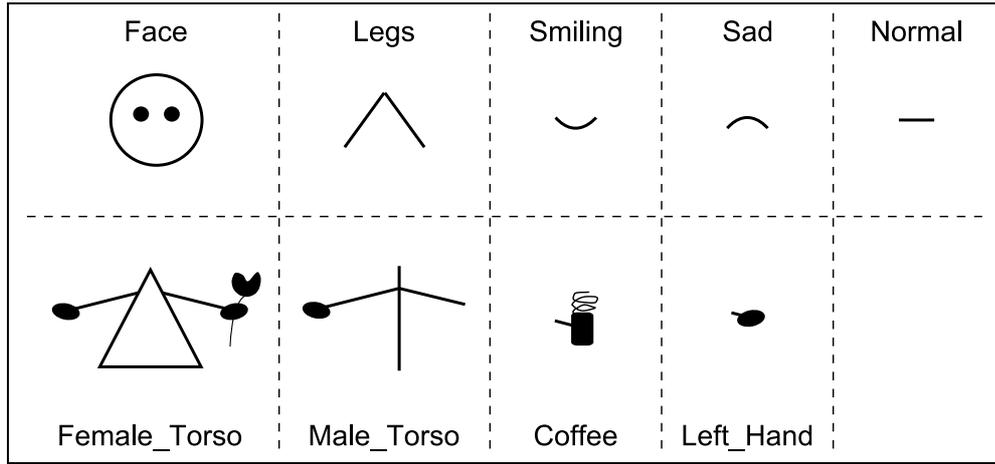


Figure 2.2.: Basic graphical shapes in the computation structure of stickmen drawings. Each shape is represented by the corresponding constant with the same name shown in Figure 2.1.

SPF Stickman. Some examples of such stickmen drawings are shown in Figure 2.4a on Page 42. Every stickman is assembled from primitive, graphical shapes which are shown in Figure 2.2. We interpret each element of sort `SPF Stickman` with the respective shape of Figure 2.2 that has the same name.

In the computation structure of stickmen drawings, we interpret the composition function \parallel by the graphical combination of primitive shapes, i.e. by putting two shapes together in an intuitively correct way, respecting for example the points where we combine two primitive shapes, and the orientation/rotation of both shapes with respect to each other. Certainly, this does not prevent the creation of irrational compound elements, i.e. it does not prevent to combine for example the shape `Legs` directly with the shape `Face`. While from the point of view of the computation structure of stickman shapes, such an irrational combination is not desired and can be prevented by respective axioms within the sort α , from the point of view of the software product family of sort `SPF Stickman`, such combinations are not restricted and represent valid product families.

Regarding the interpretation of variation points, we interpret the operation \oplus in the computation structure of stickman drawings as empty (white) rectangles surrounded with a dashed frame. The name n of the variation points is given by a label “`VP n` ” which is attached to the rectangle with a solid arrow which originates at the label and points into the rectangle. The variants of every variation point are represented by shapes surrounded by dashed rectangles that have the same size as the rectangle of the variation point, and that are connected to the corresponding variation point by dashed lines. In order to derive a concrete stickman drawing a dashed rectangle

$$\begin{aligned}
 Spf := & \left(\left(\text{asset}(\text{Coffee}) \oplus_3 \text{asset}(\text{Left_Hand}) \right) \parallel \text{Male_Torso} \right) \\
 & \oplus_2 \text{asset}(\text{Female_Torso}) \left. \right) \\
 & \parallel \left(\left(\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad}) \right) \parallel \text{asset}(\text{Face}) \right) \\
 & \parallel \text{asset}(\text{Legs})
 \end{aligned}$$

Figure 2.3.: Term representation of a software product family of sort SPF Stickman.

will be replaced by exactly one of its associated alternative variants. This is done in a way that the rectangles of the variation point and the respective variant overlap.

A very simple example of a product family of sort SPF Stickman is given by the following term

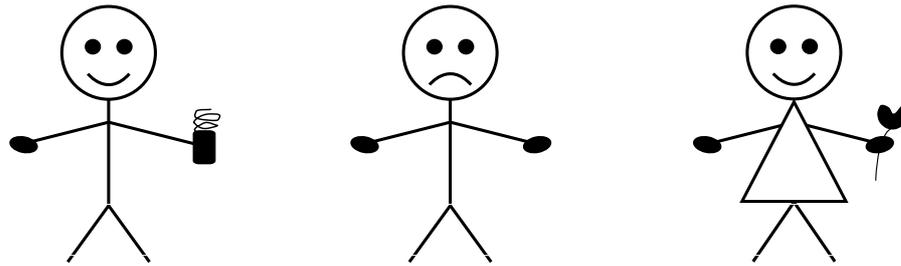
$$\text{asset}(\text{Coffee})$$

As it does not contain any variation point, it corresponds to a single product only, which is represented by the same term. Another example of a software product family of sort SPF Stickman is given by the slightly larger term *Spf* shown in Figure 2.3. It represents a product family of stickmen which comprise complete figures with male and female torsos. All of them have either a smiling or sad facial expression. The male stickmen variants can additionally hold a coffee cup in their left hand. Figure 2.4a shows three examples of stickmen drawings in the concrete computation structure of stickman shapes, which can be derived from the product family *Spf*.

The term shown in Figure 2.3 actually represents an entire product family. It specifies the structural relation between concrete parts and variation points, i.e. it determines a kind of architecture. In order to illustrate this idea more clearly we consider Figure 2.4c which shows the realization of this term in the concrete computation structure of stickman shapes.

The graphical illustration of a software product family given in Figure 2.4c illustrates the idea behind our characterization of a software product family given in Definition 2.3 (cf. Page 26) more clearly: A software product family represents a structural relation between construction entities. It defines an architecture and contains the exact information how every product can be assembled from the core assets by integrating all variation points together with their variants as construction units into the construction plan of the products. Due to its variability it induces a set of products. However, compared to a “plain” set of products, a software product family inevitably also models the information of how individual products are related. This again is the basis for defining so called *mandatory* and *optional* parts (cf. Section 2.2.3.8) and the indispensable foundation to reason about commonalities and differences between the set of similar products.

2. Formalization of Characteristic Software Product Family Concepts



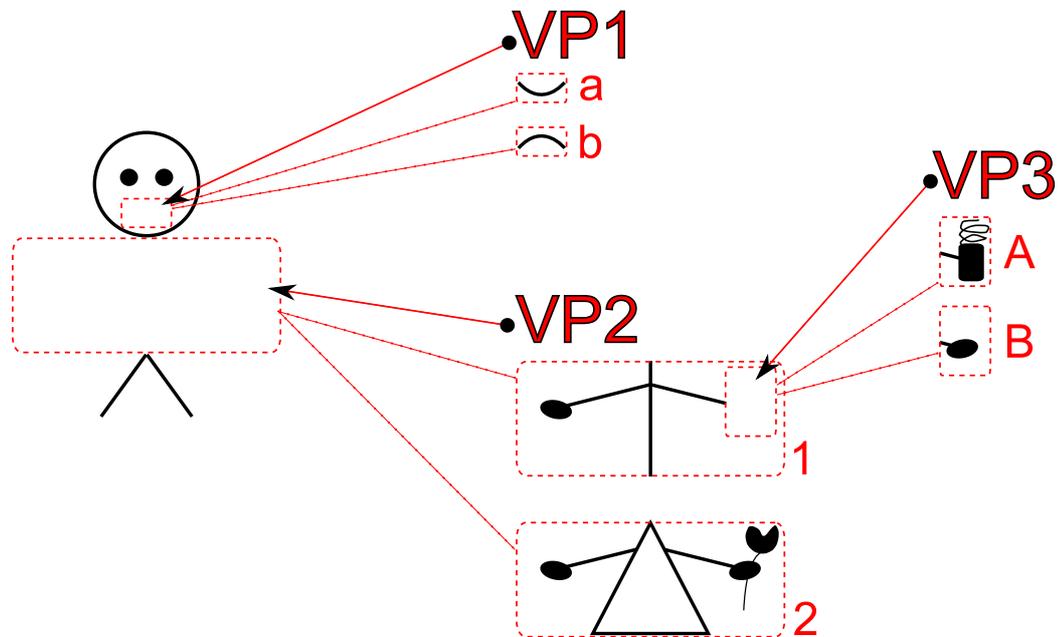
(a) Graphical illustration of some products of a family of stickmen.

a1A

b1B

a2

(b) The names of the selected variants which have to be chosen for the family shown in Figure (c) to derive the corresponding products shown in Figure (a).



(c) The actualization of the corresponding model of the product family *Spf* (cf. Figure 2.3) in the concrete computation structure of stickman drawings.

Figure 2.4.: An example of a concrete product family: A concrete computation structure for the sort *SPF Stickman*, which realizes the sort by means of graphical shapes. The stickman drawings shown in Figure (a) are products of the product family shown in Figure (c). They can be derived by configuring the product family with the variants as indicated below every product in Figure (b).

2.2.3. Axioms, Properties and Auxiliary Operations

The constructors as introduced in the preceding section allow to specify individual software product families as a composition of assets and variation points. In order to be a software product family of sort $\text{SPF } \alpha$, the elements of the sort not only have to be built by means of these constructor operations, they also have to exhibit certain properties and fulfill certain laws. These properties and laws basically reflect the interplay of the constructors and additional auxiliary functions, and thus characterize the semantics of the sort. We precisely define these properties and laws by means of axioms, which are given as equations using the constructor functions, auxiliary functions and many-sorted variables, in the following. For the remainder of this section, let capital latin letters like P, Q, R, A, B denote variables of sort $\text{SPF } \alpha$, the letters a, x denote variables of sort α , and the letters i, j, n, m denote variables of sort Nat . As usual, we adopt the convention that variables without explicit quantifier shall be universally quantified.

2.2.3.1. Axioms for Constructors

Every element of sort $\text{SPF } \alpha$ can be assembled from the constructors only. For the constructors we observe several laws which characterize the relation and the interplay between them. In the following we introduce these laws.

Recall from Section 2.2.1 that since the laws are given as equalities, they induce an equivalence relation on the set of terms, which means that different terms (elements of the corresponding term-algebra) can actually represent the same element in the concrete computation structure, that is the sort $\text{SPF } \alpha$ in our example. In Section 2.2.3.2 we address this situation and introduce a term normal form which allows to deal with such an equivalence relation.

Associativity and Commutativity of the Composition Operation

The composition is a commutative and associative operation for which we observe the following laws

$$P \parallel Q = Q \parallel P \tag{A-1}$$

$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \tag{A-2}$$

Allowing commutativity and associativity for the composition operator means that neither the order nor the nesting structure of how we compose elements by means of the composition operation are relevant. In particular, this means that for our

2. Formalization of Characteristic Software Product Family Concepts

kind of composition, a compound element merely represents the same structural information as a plain set containing the same elements.

Distributivity of the Composition over the Variants Operator

The connection between the composition and the variants operator is given by the following distributive law:

$$(P \parallel Q) \oplus_i (P \parallel R) = P \parallel (Q \oplus_i R) \quad (\text{A-3})$$

It defines a left-distributivity, stating that whenever an element P is included in both variants of the same variation point, it can be factored out, thereby reducing the alternative selection to the “remaining” parts Q and R of the original variants. For the application of a configuration selection this means that if such an element P can be factored out for a certain variation point it is always included in the final product, independently from the configuration choice for this variation point. In particular with respect to determining the common parts of products the distributive law is very important, as we will describe in detail in Section 2.2.3.7.

Since the composition is commutative (Axiom A-1), we can derive right-distributivity from left-distributivity (and vice versa) as a theorem, i.e. from A-1 and A-3 we can conclude

$$(Q \parallel P) \oplus_i (R \parallel P) = (Q \oplus_i R) \parallel P \quad (\text{T-1})$$

The distributive law alters the term representation of a product family in a way which requires a more detailed consideration. We observe two interesting situations.

Firstly, the application of the distributive law allows to introduce several variation points with the same identifier. Consider for example the distributive law (Axiom A-3) from right to left: If the common part P either is a variation point itself or contains variation points, these variation points appear twice in the resulting term of the product family, since the element P gets “duplicated” by the law. In particular, this means that identifiers for variation points are not necessarily unique anymore within the representation of a product family. However, this does neither affect the configuration process nor the set of products defined by the product family, as we will show in Section 2.2.3.3. For more details we refer to the introduction of the function `has_ntVP` in the remainder of this section on Page 81.

Secondly, regarding the term representation of a product family, the distributive law actually alters the variants of a variation point since it factors out their common part. Consider the distributive law (Axiom A-3) from left to right: After applying

2.2. Axiomatization of Software Product Family Concepts

the law the former variants $(P \parallel Q)$ and $(P \parallel R)$ have become Q and R , respectively. However, this is not in conflict to the essential property we required for the configuration process, i.e. that the outcome of a configuration of the product family always has to be fixed and precisely defined. As we will show in Section 2.2.3.3, although the distributive law changes the outcome of an atomic configuration it does not change the overall result of the respective variation point with respect to the configuration process, since the common part which was factored out is always composed to the selected variant subsequently.

Neutral Element for the Composition

We motivated the constructor `ntrl` as a special element which represents the empty product family, i.e. the notion of *nothing*. Algebraically, we can characterize this property more precisely by defining the constant `ntrl` as the *neutral element* (*identity element*) for the composition function, i.e.

$$P \parallel \text{ntrl} = P \tag{A-4}$$

While the axiom defines a right-identity we can directly derive a left-identity

$$\text{ntrl} \parallel P = P \tag{T-2}$$

from the right-identity since the composition is commutative. As we will see in Section 2.2.3.8, the properties of the element `ntrl` to be the left- and right-identity for the composition operation is essential to model the idea of optional parts.

In combination with the distributive law (Axiom A-3), the neutral element allows to factor out common parts also from term structures like

$$P \oplus_i (P \parallel R) = P \parallel (\text{ntrl} \oplus_i R) \tag{T-3}$$

$$(P \parallel R) \oplus_i P = P \parallel (R \oplus_i \text{ntrl}) \tag{T-4}$$

as the following simple derivation exemplarily shows for Theorem T-3.

$$\begin{aligned} P \oplus_i (P \parallel R) &\stackrel{A-4}{=} (P \parallel \text{ntrl}) \oplus_i (P \parallel R) \\ &\stackrel{A-3}{=} P \parallel (\text{ntrl} \oplus_i R) \end{aligned}$$

The resulting representation is in particular useful to define and to determine *optional parts*, as we will discuss in detail in Section 2.2.3.8.

Idempotence of the Variants Operator

A variation point that offers a choice between *identical* variants always results in

2. Formalization of Characteristic Software Product Family Concepts

the same product with respect to this particular variation point. For example, the variation point $P \oplus_i P$ always represents the element P , no matter which of its variants is selected. We call such a variation point a *trivial variation point*. A trivial variation point $P \oplus_i P$ can always be replaced by its variant P . Algebraically, this means that the variants operator is *idempotent*, which is expressed by the law

$$P \oplus_i P = P \tag{A-5}$$

Applying the idempotence law for a given element of sort SPF α from right to left allows to introduce “new” variation points with an arbitrary label i at any time. In particular, since we do not restrict the label i , it can also take values which already exist as the label of a variation point in the considered element. We call variation points with identical labels *clashing*. However, introducing clashing variation points by means of applying Axiom A-5 does not alter the result of any selection operation since in the case of clashing variation point labels (i) the configuration selection of clashing variation points is not influenced, and (ii) any configuration choice for the term $P \oplus_i P$ always results in the element P , since both variants are equivalent to P . We discuss these cases in more detail in Section 2.2.3.3, where we formalize the notion of configuration selection.

2.2.3.2. Term Normal Form of Product Families

Let the (syntactic) signature for the sort SPF α be defined as the tuple

$$\Sigma_{\text{SPF}} = \left(\{\alpha, \text{SPF } \alpha\}, \right. \\ \left. \{\text{asset, ntrl, ||, } \oplus, \text{selR, selL, is_product, is_mand, has_ntVP, Assets, modify}\} \right)$$

with the following type association of sorts to the function symbols

$$\begin{array}{ll} \text{asset} & : \alpha \rightarrow \text{SPF } \alpha \\ \text{ntrl} & : \text{SPF } \alpha \\ \text{||} & : \text{SPF } \alpha, \text{SPF } \alpha \rightarrow \text{SPF } \alpha \\ \oplus & : \text{SPF } \alpha, \text{SPF } \alpha, \text{Nat} \rightarrow \text{SPF } \alpha \\ \text{selR, selL} & : \text{Nat, SPF } \alpha \rightarrow \text{SPF } \alpha \\ \text{is_product} & : \text{SPF } \alpha \rightarrow \text{Bool} \\ \text{is_mand} & : \text{SPF } \alpha, \text{SPF } \alpha \rightarrow \text{Bool} \\ \text{has_ntVP} & : \text{Nat, SPF } \alpha \rightarrow \text{Bool} \\ \text{Assets} & : \text{SPF } \alpha \rightarrow \text{Set SPF } \alpha \\ \text{modify} & : \text{SPF } \alpha, \text{SPF } \alpha, \text{SPF } \alpha \rightarrow \text{SPF } \alpha \end{array}$$

2.2. Axiomatization of Software Product Family Concepts

Let $\Sigma_{\text{SPFC}} = (\{\alpha, \text{SPF } \alpha\}, \{\text{asset}, \text{ntrl}, \parallel, \oplus\})$ denote the sub-signature of Σ_{SPF} that comprises only the constructor function symbols. With $\text{Terms}(\Sigma_{\text{SPFC}})$ (respectively $\text{Terms}(\Sigma_{\text{SPF}})$) we denote the set of ground terms that can be constructed over the signature Σ_{SPFC} (respectively Σ_{SPF}). The way in which the terms of $\text{Terms}(\Sigma_{\text{SPFC}})$ (respectively $\text{Terms}(\Sigma_{\text{SPF}})$) are constructed is characterized algebraically by the term algebra $T(\Sigma_{\text{SPFC}})$ (respectively $T(\Sigma_{\text{SPF}})$), that provides an inductive definition of the set $\text{Terms}(\Sigma_{\text{SPFC}})$ (respectively $\text{Terms}(\Sigma_{\text{SPF}})$). For details we refer to Section 2.2.1, Page 31.

The Axioms A-1–A-5 have a direct impact on the way in which we represent elements of sort $\text{SPF } \alpha$. All of these axioms are equalities and thus imply that there exist several different ways to assemble a software product family—being an element of sort $\text{SPF } \alpha$ —from the set of constructors of sort $\text{SPF } \alpha$. For example, due to the commutative law (Axiom A-1) and the existence of a neutral element (Axiom A-4), the following two ways of assembly

$$\text{asset}(a) \parallel \text{asset}(b) \quad \text{and} \quad \text{asset}(b) \parallel (\text{asset}(a) \parallel \text{ntrl})$$

are equivalent, and result in the same element of the underlying computation structure/algebra.

Each of these ways of assembly can be represented by a term $t \in \text{Terms}(\Sigma_{\text{SPF}})$ which is constructed in a corresponding way over the signature Σ_{SPF} according to the term-algebra $T(\Sigma_{\text{SPF}})$. In consequence, this implies that regarding the term representation of elements of sort $\text{SPF } \alpha$ there are different terms that actually represent the same underlying element of sort $\text{SPF } \alpha$. In this light, the axioms A-1–A-5 induce an equivalence relation on the set of (constructor) terms, which partitions the set of terms in subsets of equivalent terms.

For the term algebra $T(\Sigma_{\text{SPF}})$ we can represent each of these subsets of equivalent terms in a unique way by a distinguished element of that subset. For software product families this means that we can represent any element of sort $\text{SPF } \alpha$ by a unique term. Such a unique term is called a term normal form. The function NF returns for any element of sort $\text{SPF } \alpha$ its unique term normal form. It has the signature

$$\text{NF} \quad : \quad \text{SPF } \alpha \rightarrow \text{Terms}(\Sigma_{\text{SPFC}}) \quad (2.5)$$

Due to the unique term normal form that is realized by the function NF we can represent every element of the underlying computation structure in a unique way by a unique term. Consequently, the term normal form is one possible basis (and also a practically relevant one) to define the notion of equivalence of elements of sort

2. Formalization of Characteristic Software Product Family Concepts

SPF α . We consider two product families to be *equivalent* if their corresponding term normal forms are identical (with respect to *term equivalence*). We write $=_{\text{NF}}$ to denote the equality of elements of sort SPF α modulo the term normal form. For our algebraic considerations, and for the remainder of this chapter, we consider the notion of equality of elements of sort SPF α always modulo the equivalence relation induced by the normal form, unless otherwise noted.

The function NF itself is composed of the two functions *term* and *norm* in the following way

$$\text{NF}(x) = \text{norm}(\text{term}(x)) \quad (2.6)$$

where *term* and *norm* have the signatures

$$\text{term} : \text{SPF } \alpha \rightarrow \text{Terms}(\Sigma_{\text{SPFC}}) \quad (2.7)$$

$$\text{norm} : \text{Terms}(\Sigma_{\text{SPFC}}) \rightarrow \text{Terms}(\Sigma_{\text{SPFC}}) \quad (2.8)$$

The function *term* constructs for a given element $x \in \text{SPF } \alpha$ a structure equivalent term $t \in \text{Terms}(\Sigma_{\text{SPFC}})$ by simply traversing the structure of x and reproducing the same structure in t by applying corresponding functions from the ground term algebra $T(\Sigma_{\text{SPFC}})$, respectively. We use the same function symbols to denote the functions in $T(\Sigma_{\text{SPFC}})$ and in SPF α . Since this is a standard procedure for obtaining the term representation of an element of a Σ -algebra we do not explain it in more detail here. Since every element of sort SPF α can be built using the constructors only, it is sufficient to define the term normal form on the subset $\text{Terms}(\Sigma_{\text{SPFC}}) \subset \text{Terms}(\Sigma_{\text{SPF}})$ of constructor terms for the sub-signature Σ_{SPFC} and the corresponding term algebra $T(\Sigma_{\text{SPFC}})$ (instead of defining it on the set $\text{Terms}(\Sigma_{\text{SPF}})$ of all terms and the corresponding algebra $T(\Sigma_{\text{SPF}})$).

The function *norm* realizes the reduction of a term $t \in \text{Terms}(\Sigma_{\text{SPFC}})$ to its unique normal form, and thus represents the crucial part of NF. It is specified in detail by the equations 2.10–2.13 on Page 50. Before we give a precise, algebraic definition of the term normal form realized by the function NF and its sub-function *norm*, we will describe it informally. Intuitively, the term normal form respects the following properties.

1. Regarding Axiom A-4 (and the corresponding Theorem T-2), all “unnecessary” (with respect to Axioms A-4 and A-2) neutral elements are “stripped off”. This means, that the element *ntrl* appears in *no* normal form as a direct argument to any composition operator. It only appears as a direct argument of a variants operator, or in the ground term *ntrl*, representing the neutral element itself. For example, the term $P \parallel \text{ntrl}$ is reduced to its normal form $\text{norm}(P \parallel \text{ntrl}) = P$, and the term $P \parallel (\text{ntrl} \parallel Q)$ to $\text{norm}(P \parallel (\text{ntrl} \parallel Q)) = P \parallel Q$.

2.2. Axiomatization of Software Product Family Concepts

2. Regarding Axiom A-5, all trivial variation points of the kind $P \oplus_i P$ are reduced to the normal form $\text{norm}(P \oplus_i P) = P$ according to the application of Axiom A-5 from right to left. Thus, the normal form is free of such trivial variation points, and contains only variation points which offer a true alternative choice between different variants.

3. Regarding Axiom A-3, the distributive laws are always applied from right to left. For the term normal form this means that common parts are always factored out and variation points are pushed inwards the term hierarchy as far as possible. For example, the term $(P \parallel Q) \oplus_i (P \parallel R)$ is normalized to $\text{norm}((P \parallel Q) \oplus_i (P \parallel R)) = P \parallel (Q \oplus_i R)$.

4. Due to the commutativity (Axiom A-1) and associativity (Axiom A-2), the composition operation becomes independent of the order and the nesting structure in which its arguments are specified, and the composition merely characterizes a flat set of assets, variation points, or neutral elements. This means that the way how we (hierarchically) compose assets, variation points and neutral elements is irrelevant. For a unique normal form, all elements which are combined by the same nested structure of composition operators are ordered, based on a *lexicographical comparison* of their corresponding tree representation. The lexicographical comparison itself is based on a linear order relation $<_{\text{Symb}_\alpha}$ of the operator symbols, which we informally describe by (a formal definition will be given on Page 55):

$$\begin{aligned} \text{ntrl} &<_{\text{Symb}_\alpha} \text{asset}(a_1) <_{\text{Symb}_\alpha} \dots <_{\text{Symb}_\alpha} \text{asset}(a_n) & (2.9) \\ &<_{\text{Symb}_\alpha} \oplus_1 <_{\text{Symb}_\alpha} \dots <_{\text{Symb}_\alpha} \oplus_m <_{\text{Symb}_\alpha} \parallel \end{aligned}$$

Using this lexicographical order allows to arrange all elements which are at the same level of a (nested) composition in a unique way, which is represented as a unique term representation (cf. Figure 2.5b).

Formally, the function *norm* is characterized by the following laws, which specify the transformation of any term into its unique term normal form. In order to improve the comprehensibility we have encapsulated the individual steps in the auxiliary functions *coll*, *sort*, *reconv*, *cmn*, and *diff*, which are specified after the definition of *norm*. Note that the equals signs in the case differentiations actually mean term equality.

2. Formalization of Characteristic Software Product Family Concepts

$$\text{norm}(\text{ntrl}) = \text{ntrl} \quad (2.10)$$

$$\text{norm}(\text{asset}(a)) = \text{asset}(a) \quad (2.11)$$

$$\text{norm}(P \parallel Q) = \begin{cases} \text{ntrl} & , \text{ (norm}(P) = \text{ntrl}) \wedge \text{ (norm}(Q) = \text{ntrl}) \\ \text{reconv}(\text{sort}(\text{coll}(P))) & , \text{ (norm}(Q) = \text{ntrl}) \wedge \text{ (norm}(P) \neq \text{ntrl}) \\ \text{reconv}(\text{sort}(\text{coll}(Q))) & , \text{ (norm}(P) = \text{ntrl}) \wedge \text{ (norm}(Q) \neq \text{ntrl}) \\ \text{reconv}(\text{sort}(\text{coll}(P \parallel Q))) & , \text{ else} \end{cases} \quad (2.12)$$

$$\text{norm}(P \oplus_i Q) = \begin{cases} \text{norm}(P) & , \text{ norm}(P) = \text{norm}(Q) \\ \text{norm}(P) \oplus_i \text{norm}(Q) & , \text{ (norm}(P) \neq \text{norm}(Q)) \wedge \text{ (cmn}(\text{coll}(P), \text{coll}(Q)) = \emptyset) \\ \text{reconv}(\text{sort}(X \cup Y)) & , \text{ (norm}(P) \neq \text{norm}(Q)) \wedge \text{ (cmn}(\text{coll}(P), \text{coll}(Q)) \neq \emptyset) \end{cases} \quad (2.13)$$

where X and Y (appearing in Law 2.13) are abbreviations for the sake of readability

$$X := \text{cmn}(\text{coll}(P), \text{coll}(Q))$$

$$Y := \left\{ \begin{array}{l} \text{reconv}(\text{sort}(\text{diff}(\text{coll}(P), \text{coll}(Q)))) \\ \text{reconv}(\text{sort}(\text{diff}(\text{coll}(Q), \text{coll}(P)))) \end{array} \right\} \oplus_i$$

The Laws 2.10 and 2.11 state that terms representing the neutral element and atomic assets are already in term normal form.

Regarding Law 2.12, for a compound element it is necessary to collect all sub-elements which belong to this composition, and transform them into an ordered term structure. The different cases are necessary to strip off all possible combinations of trivial neutral elements. The collecting, sorting, and transformation of these arguments is encapsulated by the three auxiliary functions `coll`, `sort` and `reconv`, where `coll` collects the relevant elements in a list, `sort` sorts the list, and `reconv` reconverts the list into a term. The order in which these auxiliary functions are composed by the function `norm`, and the effects of the functions themselves guarantee that the resulting term is a composed term which respects the sorting order, and whose arguments are normalized terms, themselves.

2.2. Axiomatization of Software Product Family Concepts

Regarding Law 2.13, the term representation of a variation point is in normal form if (i) it is not a trivial variation point (item 3 in the property list on Page 48), and (ii) the common parts of its two variants are factored out according to Axiom A-3, and (iii) its variants are in normal form, themselves. Trivial variation points are reduced directly. If both variants actually have common factors ($\text{cmn}(\text{coll}(P), \text{coll}(Q)) \neq \langle \rangle$), these common parts are factored out by combining the auxiliary functions `coll`, `cmn`, and `diff`, which we will introduce in the following. The factorization is achieved by collecting all relevant elements of both variants (function `coll`) in two lists, determining their common elements (function `cmn`), and factoring them out. The remaining variation point is reconstructed from the variant specific elements of each variant, which are additionally sorted (function `diff` in combination with `sort`), respectively. The common elements of both variants P and Q are represented as a multiset abbreviated by the variable X , while the variable Y abbreviates the multiset which contains the "reduced" variation point as its single element. Here, "reduced" means that the common parts of its variants are factored out and only the variant-specific parts are preserved in the respective variants. Since the common parts and the remaining variation point will form a compound element in the end, the set of common parts together with the "reduced" variation point itself has to be ordered (function `sort`) prior to reconstructing them into a compound term. This guarantees that the factors of the resulting compound element are ordered (according to property 4 in the property list on Page 48), and that the compound element is in term normal form, itself. If both variants of a variation point do not have common parts, the entire variation point is in term normal form if both its variants are in normal form, respectively. We introduce all auxiliary functions in the following. Note that the recursive call of `norm` in Law 2.12 is made indirectly within the function `coll`. The remaining operations are all primitive recursive, and in particular not involved in a mutual recursion with `norm` or other functions.

Collecting the Relevant Assets and Variation Points

The function

$$\text{coll} : \text{Terms}(\Sigma_{\text{SPFC}}) \rightarrow \text{MSet } \text{Terms}(\Sigma_{\text{SPFC}}) \quad (2.14)$$

is initially called by the function `norm` on the terms of compound elements (Law 2.12), or the terms of variants of variation points (Law 2.13). It traverses recursively the term structure of a *compound element* along the respective composition operators (in pre-order), collects all assets and variation points which are combined in this (possibly nested) compound element, and returns them as a multiset of sort $\text{Seq } \text{Terms}(\Sigma_{\text{SPFC}})$. Thus, for compound elements it returns the set of all assets and variation points which are at the same level with respect to a nested structure of composition operators. In particular, these represent those elements which have to be sorted (in Law 2.12) in the subsequent step by the function `sort`, or the elements from which common elements have to be factored out (Law 2.12). The function `coll` is formally characterized by the following laws.

2. Formalization of Characteristic Software Product Family Concepts

$$\text{coll}(\text{ntrl}) = \emptyset \quad (2.15)$$

$$\text{coll}(\text{asset}(a)) = \{\text{asset}(a)\} \quad (2.16)$$

$$\text{coll}(P \parallel Q) = \begin{cases} \text{coll}(P) & , Q = \text{ntrl} \\ \text{coll}(Q) & , P = \text{ntrl} \\ \text{coll}(P) \cup \text{coll}(Q) & , \text{else} \end{cases} \quad (2.17)$$

$$\text{coll}(P \oplus_i Q) = \begin{cases} \text{coll}(P) & , \text{norm}(P) = \text{norm}(Q) \\ \{\text{norm}(P) \oplus_i \text{norm}(Q)\} & , \left(\text{norm}(P) \neq \text{norm}(Q) \right) \wedge \\ & \left(\text{cmn}(\text{coll}(P), \text{coll}(Q)) = \emptyset \right) \\ \text{coll}(\text{norm}(P \oplus_i Q)) & , \left(\text{norm}(P) \neq \text{norm}(Q) \right) \wedge \\ & \left(\text{cmn}(\text{coll}(P), \text{coll}(Q)) \neq \emptyset \right) \end{cases} \quad (2.18)$$

The symbols \emptyset , $\{a\}$, and \cup denote the *empty multiset*, the *multiset containing the single element a* , and the *set union* of multisets. For a precise definition of the sort $\text{MSet } \alpha$ of multisets see Figure A.6 in the Appendix A on Page 236.

Regarding Laws 2.15 and 2.16, since a neutral element and an atomic asset are already in normal form, their corresponding terms can simply be collected. Law 2.17 traverses along the arguments of a composition, and collects them, unless they are trivial neutral elements. Neutral elements are discarded directly (cf. the first two cases of the axiom). Since the arguments can represent compound elements, themselves, the function coll traverses the entire \parallel -structure recursively, until it reaches a term representing an asset or a variation point. These two kinds of elements are collected in the multiset which is returned.

Law 2.18 specifies the situation of collecting variation points. Due to the distributive law (Axiom A-3), a variation point itself is equivalent to a compound element, if its variants contain common parts that can be factored out. In such a case the representation of a variation point as a compound element is essential for coll . Thus, if the variants of a variation point still contain common parts (i.e. if $\text{NF}(P \oplus_i Q)$ has *not* the general term structure $A \oplus_i B$, where A is the term normal form of P , and B is one of Q), we represent the variation point as a compound element by applying the distributive law. This is done in the second case of the axiom by applying NF on the entire variation point. Then, after normalization in a subsequent recursive call of coll the common parts and the remaining variation point are added as separate elements to the multiset that is returned by coll . Otherwise, if no common parts exist (first case of the axiom), the term representation of the variation point can be added to the multiset directly, where its variants are further normalized.

Termination of norm: Mutual Recursion between the Functions norm and coll

The functions `coll` and `norm` are *mutually recursive*, i.e. `coll`, which is called within `norm` (in the case where its argument represents a compound element, cf. Law 2.12), calls `norm`, itself. With this mutual recursion, the actual recursion of the function `norm` is realized, which guarantees that all elements (assets and variation points) which are collected by `coll` are also transformed into normal forms, themselves. However, applying the function `norm` to any term of sort $\text{Terms}(\Sigma_{\text{SPFC}})$ always terminates yielding the unique term normal form. The following theorem makes this more precise.

Theorem 2.1 (Termination of the Function `norm`). *The reduction system which is obtained by applying the axioms defined for the function `norm` from left to right always terminates for any term of sort $\text{Terms}(\Sigma_{\text{SPFC}})$.*

Proof. The mutual recursion between `norm` and `coll` follows the construction scheme of *primitive recursion*, which basically means that recursive calls are always performed on subterms of the original term. This means that:

- In the specification of function `norm` (Laws 2.10–2.13), all arguments—except for one—of the direct recursive calls of `norm`, and the mutually recursive calls of `coll` on the right-hand sides of the equations are subterms of the arguments of calls of `norm` and `coll` on the left-hand sides of the equations. The only exception is the last case where `norm`($P \parallel Q$) triggers a call `coll`($P \parallel Q$) with the same arguments. However, the argument ($P \parallel Q$) is always taken apart in its subterms P and Q in the successive call of `coll` (cf. Law 2.17).
- For the specification of function `coll` the arguments get also broken down in its sub-elements in recursive calls, except for Law 2.18. Here, the call `coll`($P \oplus_i Q$) triggers mutually recursive calls of the kind `norm`($P \oplus_i Q$), which have the same arguments ($P \oplus_i Q$). However, in the following mutually recursive calls of `coll` and `norm`, the arguments are again sub-elements of the recursively preceding calls of `coll`, i.e. the call `norm`($P \oplus_i Q$) triggers only calls of `norm` and `coll` with the arguments P and Q (cf. Equation 2.13). Thus, considering the entire mutual recursion, mutually recursive calls of `norm` and `coll` are performed on sub-elements of the initial arguments, only.

Regarding the auxiliary functions `reconv`, `cmn`, and `diff`, which are used within the laws of `norm` and `coll`, all of them are also defined according to the primitive recursion scheme, as we will still see in the remainder of this section. Consequently, these auxiliary functions always terminate, too. Certainly, we can also assume termination for the auxiliary function `sort`. In summary, this means that the arguments of `norm` and `coll` are always broken up into its sub-elements the latest after an entire mutually recursive cycle `norm` \rightarrow `coll` \rightarrow `norm`, respectively `coll` \rightarrow `norm` \rightarrow `coll`. Thus, the mutual recursion between `norm` and `coll` follows the construction scheme of primitive recursion, and is consequently guaranteed to terminate. \square

2. Formalization of Characteristic Software Product Family Concepts

Auxiliary Operations `cmn` and `diff`: Determining the Common and the Variant-specific Parts of the Variants of a Variation Point

Law 2.13 factors out common parts of variants, which requires to determine the common and the variant-specific parts of the two variants of a variation point, before. For this purpose the relevant elements (assets and normalized variation points) of both variants are collected by preceding calls of function `coll`. This results in two multisets, for which the common elements are determined by means of the function

$$\text{cmn} : \text{MSet } \alpha, \text{MSet } \alpha \rightarrow \text{MSet } \alpha \quad (2.19)$$

The function `cmn` (common) returns a multiset of those elements which appear in both of its arguments. Being a multiset, multiple occurrences of elements are respected, e.g. the result of `cmn` ($\{a, b, a, c\}$, $\{b, a, a\}$) is the multiset $\{a, b, a\}$. As the function `cmn` is a standard operation on multisets, and not specific for our normal form, we have deferred its specification to Figure A.7 (Page 237) in Appendix A.

While the function `cmn` determines the common elements of two multisets, we also have to determine the different elements of two multisets, i.e. those elements which are in one multiset, but not in another. This is realized by the function

$$\text{diff} : \text{MSet } \alpha, \text{MSet } \alpha \rightarrow \text{MSet } \alpha \quad (2.20)$$

The function `diff`(A, B) determines those elements of the multiset A which are not in the multiset B . For example, `diff`($\{a, b, d, a\}$, $\{b, a, c\}$) yields the set $\{d, a\}$. Thus, for every element of B one instance of an identical element is removed from A , if A contains an instance of such an element at all. The complete specification of `diff` is shown in Figure A.7 (Page 237) in Appendix A.

In Law 2.13, the functions `cmn` and `diff` are combined in order to determine the common elements and the variant specific elements of both variants. More precisely, the variables X and Y which we use in Law 2.13 are just abbreviations, where X represents the multiset of common elements of the variants A and B , and Y represents the multiset of “remaining” variation point whose variants are free of common parts and only comprise the sorted variant specific parts anymore. Since the common part and the “remaining” variation point itself form a compound element, the normal form requires that these elements have to be sorted, which is why the multisets X and Y are combined and sorted, before they are finally re-transformed into a term representing the normal form.

2.2. Axiomatization of Software Product Family Concepts

Sorting the Collected Elements

Regarding Law 2.12, the elements—only assets and variation points—which have been collected by `coll` have to be sorted. This requires to compare assets with assets, variation points with variation points, but also assets and variation points with each other. The function `sort` realizes this by comparing the pre-order sequences of the corresponding operator trees of each element in the collected sequence *lexicographically*. For the lexicographical order we introduce (for every sort α) the set Symb_α of operator symbols.

$$\text{Symb}_\alpha := \{\text{asset}(a) \mid a \in \alpha\} \cup \{\text{ntrl}, \|\} \cup \{\oplus_i \mid i \in \text{Nat}\}$$

Each symbol denotes the element of sort $\text{SPF } \alpha$ with the same name. We define a linear order $<_{\text{Symb}_\alpha}$ on the elements of Symb_α in the following way:

$$\forall a \in \alpha, i \in \text{Nat} : \text{ntrl} <_{\text{Symb}_\alpha} \text{asset}(a) <_{\text{Symb}_\alpha} \oplus_i <_{\text{Symb}_\alpha} \|\quad (2.21)$$

$$\forall i, j \in \text{Nat} : (\oplus_i <_{\text{Symb}_\alpha} \oplus_j) \Leftrightarrow (i < j) \quad (2.22)$$

$$\forall a, b \in \alpha : (\text{asset}(a) <_{\text{Symb}_\alpha} \text{asset}(a')) \Leftrightarrow (a <_\alpha a') \quad (2.23)$$

where $<_\alpha$ realizes a linear order on the elements of sort α , which has to be defined separately for each sort α .

The elements which we have collected are compared based on their operator symbols. The function `Ops` traverses a term of sort $\text{Terms}(\Sigma_{\text{SPFC}})$ in pre-order and constructs the corresponding sequence of operator symbols. The symbols $\langle \rangle$, and \circ denote the *empty sequence*, and the *concatenation* of sequences. For a precise definition of the sort $\text{Seq } \alpha$ of sequences see Appendix A, Page 234.

$$\text{Ops} : \text{Terms}(\Sigma_{\text{SPFC}}) \rightarrow \text{Seq } \text{Symb}_\alpha \quad (2.24)$$

$$\text{Ops}(\text{ntrl}) = \langle \text{ntrl} \rangle \quad (2.25)$$

$$\text{Ops}(\text{asset}(a)) = \langle \text{asset}(a) \rangle \quad (2.26)$$

$$\text{Ops}(P \|\ Q) = \langle \|\rangle \circ \text{Ops}(P) \circ \text{Ops}(Q) \quad (2.27)$$

$$\text{Ops}(P \oplus_i Q) = \langle \oplus_i \rangle \circ \text{Ops}(P) \circ \text{Ops}(Q) \quad (2.28)$$

2. Formalization of Characteristic Software Product Family Concepts

Based on the operation Ops which returns the sequence of operator symbols and the order $<_{\text{Symb}_\alpha}$ defined on the set of symbols, we can define an order on entire terms of sort $\text{Terms}(\Sigma_{\text{SPFC}})$ by comparing their operator sequences lexicographically. We represent the lexicographical comparison by the function

$$<_{\text{lex}} : \text{Seq Symb}_\alpha, \text{Seq Symb}_\alpha \rightarrow \text{Bool} \quad (\text{infix}) \quad (2.29)$$

with the following laws

$$\langle \rangle <_{\text{lex}} X = \text{true} \quad (2.30)$$

$$\langle s_1 \rangle \circ r_1 <_{\text{lex}} \langle s_2 \rangle \circ r_2 \Leftrightarrow ((s_1) <_{\text{Symb}_\alpha} (s_2)) \vee (r_1 <_{\text{lex}} r_2) \quad (2.31)$$

Based on such a lexicographical order we define an order relation \sqsubset on terms of sort $\text{Terms}(\Sigma_{\text{SPFC}})$.

$$\sqsubset : \text{Terms}(\Sigma_{\text{SPFC}}), \text{Terms}(\Sigma_{\text{SPFC}}) \rightarrow \text{Bool} \quad (\text{infix}) \quad (2.32)$$

with the following law

$$(s \sqsubset s') \Leftrightarrow \text{Ops}(s) <_{\text{lex}} \text{Ops}(s') \quad (2.33)$$

With the linear order \sqsubset on terms representing elements of sort $\text{SPF } \alpha$ we can sort a sequence of assets and variation points in a unique way, using a standard sorting algorithm. We represent the sorting of a sequence of terms representing elements of sort $\text{SPF } \alpha$ by the function

$$\text{sort} : \text{MSet Terms}(\Sigma_{\text{SPFC}}) \rightarrow \text{Seq Terms}(\Sigma_{\text{SPFC}}) \quad (2.34)$$

For the scope of this thesis we do not specify the function sort in more detail, as we can use any standard sorting algorithm which is based on the *comparison of two elements*, e.g. *Quicksort* [Hoa62].

Reconstructing a Compound Element from a Sorted Sequence of Individual Elements

After the arguments of a compound element have been sorted, or the common elements of two variants have been factored out, the corresponding representation as a sequence of elements has to be transformed back into a single compound term of sort $\text{Terms}(\Sigma_{\text{SPFC}})$. For this purpose we use the function reconv

2.2. Axiomatization of Software Product Family Concepts

$$\text{reconv} : \text{Seq } Terms(\Sigma_{\text{SPFC}}) \rightarrow Terms(\Sigma_{\text{SPFC}}) \quad (2.35)$$

which reconverts a sequence of terms of sort $Terms(\Sigma_{\text{SPFC}})$ into a degenerated compound term. The function reconv is specified by the following laws

$$\text{reconv}(\langle \rangle) = \text{ntrl} \quad (2.36)$$

$$\text{reconv}(\langle a \rangle) = a \quad (2.37)$$

$$\text{reconv}(\langle a \rangle \circ s) = a \parallel \text{reconv}(s) \quad (2.38)$$

It iterates through the given sequence and appends every element on the right side of a nested term. Thereby, it constructs a degenerated tree representation (of sort $Terms(\Sigma_{\text{SPFC}})$) from the elements of the sequence supplied as its argument.

Example

Figure 2.5a shows an example of a term $Ex \in Terms(\Sigma_{\text{SPFC}})$ and its corresponding unique term normal form (Figure 2.5b). Since Ex is a compound element, the transformation into its normal form results in applying Rule 2.12

$$\text{norm}(Ex) = \text{reconv}(\text{sort}(\text{coll}(Ex)))$$

The innermost operation $\text{coll}(Ex)$ is equivalent to the following multiset

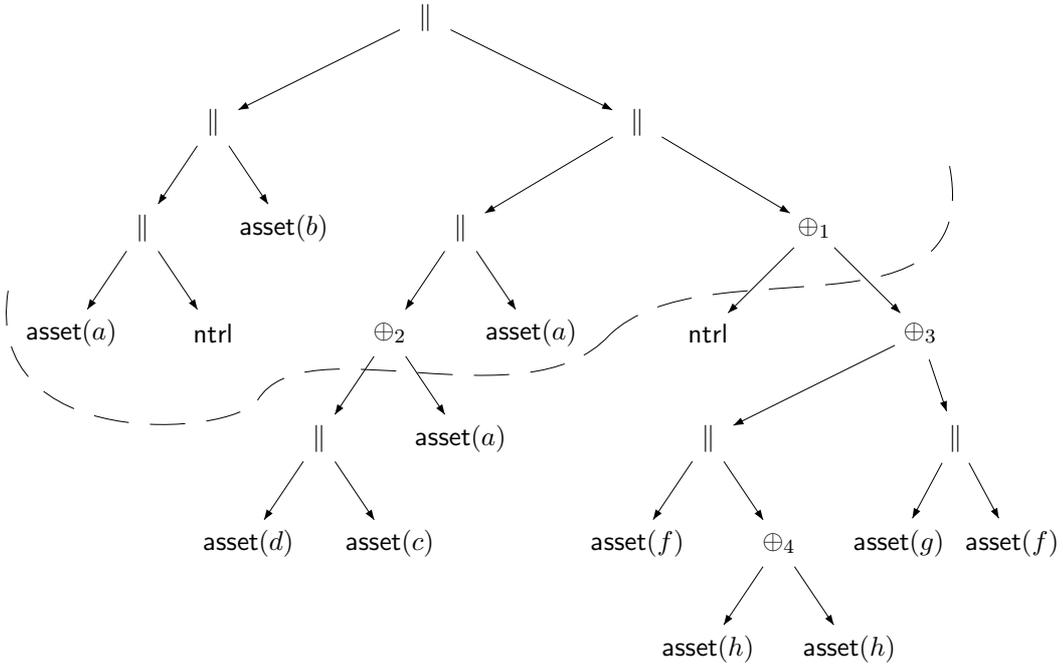
$$\begin{aligned} \text{coll}(Ex) &= \dots \\ &= \left\{ \text{asset}(a), \text{asset}(b), \oplus_2(\text{norm}(\dots), \text{norm}(\dots)), \text{asset}(a), \right. \\ &\quad \left. \oplus_1(\text{norm}(\dots), \text{norm}(\dots)) \right\} \end{aligned}$$

Sorting this set by comparing the elements by means of \sqsubset brings the assets to the front, and the two variation points to the end, and results in the following sequence:

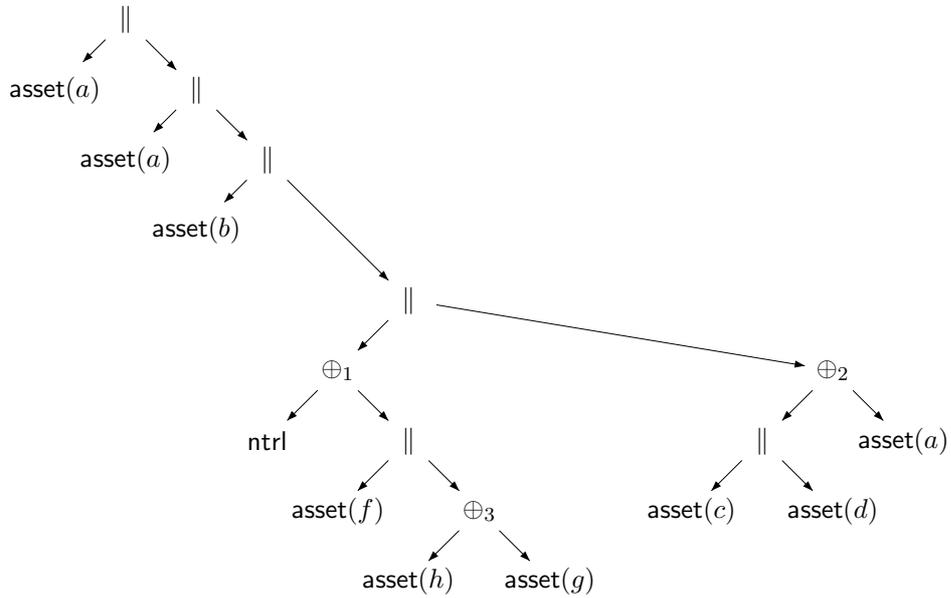
$$\begin{aligned} \text{sort}(\text{coll}(Ex)) &= \dots \\ &= \left\langle \text{asset}(a), \text{asset}(a), \text{asset}(b), \oplus_1(\text{norm}(\dots), \text{norm}(\dots)), \right. \\ &\quad \left. \oplus_2(\text{norm}(\dots), \text{norm}(\dots)) \right\rangle \end{aligned}$$

Finally, applying reconv on this sorted list, i.e. calling $\text{reconv}(\text{sort}(\text{coll}(Ex)))$ results in the desired term normal form, whose tree representation is shown in Fig. 2.5b.

2. Formalization of Characteristic Software Product Family Concepts



(a) Tree representation of a term of sort $Terms(\Sigma_{SPFC})$. The $asset$, $ntrl$ and \oplus nodes above the dashed line represent those elements which are combined by the nested composition operators starting in the root. In particular, these elements are composed in a unique order in the normal form shown in Figure (b).



(b) Tree representation of the unique term normal form of the term shown in (a).

Figure 2.5.: The tree representations of a term of sort $Terms(\Sigma_{SPFC})$ and its normal form. We assume that a, b, c, \dots are elements of sort α equipped with an alphabetical order.

2.2. Axiomatization of Software Product Family Concepts

Uniqueness of the Term Normal Form

The term normal form represented by the function NF is unique. This means that any software product family $P \in \text{SPF } \alpha$ which fulfills the axiomatization can be represented by a unique term of sort $\text{Terms}(\Sigma_{\text{SPFC}})$. More precisely, for any two software product families $P, Q \in \text{SPF } \alpha$ that are equivalent according to the constructor Axioms A-1–A-5, their corresponding term representations $\text{NF}(P), \text{NF}(Q) \in \text{Terms}(\Sigma_{\text{SPFC}})$ are identical, i.e. $\text{NF}(P) = \text{NF}(Q)$. Regarding the constructor axioms this means that the application of NF to the left-hand and right-hand side of each of the constructor Axioms A-1–A-5 yields the same (with respect to term equivalence) result. The following Theorem 2.2 states this more precisely.

Theorem 2.2 (Uniqueness of the Normal Form). *The normal form realized by the function NF is unique, i.e. it reduces any two elements of sort $\text{SPF } \alpha$ to the same unique term iff they can be transformed into one another using the constructor Axioms A-1–A-5. In particular, we observe the following identities, where $P, Q, R \in \text{Terms}(\Sigma_{\text{SPFC}})$, and each equation reflects the identity represented by one constructor axiom.*

$$\begin{aligned}
 \text{NF}(P \parallel \text{ntrl}) &= \text{NF}(P) \\
 \text{NF}(P \oplus_i P) &= \text{NF}(P) \\
 \text{NF}(P \parallel Q) &= \text{NF}(Q \parallel P) \\
 \text{NF}(P \parallel (Q \parallel R)) &= \text{NF}((P \parallel Q) \parallel R) \\
 \text{NF}((P \parallel Q) \oplus_i (P \parallel R)) &= \text{NF}(P \parallel (Q \oplus_i R))
 \end{aligned}$$

Proof. The only ways in which two elements of sort $\text{Terms}(\Sigma_{\text{SPFC}})$ can be transformed in each other is by applying one of the constructor Axioms A-1–A-5. For each of these axioms the application of the function $\text{NF}(x) = \text{norm}(\text{term}(x))$ to the left-hand (lhs) and right-hand (rhs) side of each constructor axiom yields the same term, since $\text{norm}(lhs) = \text{norm}(rhs)$ for each of the constructor axioms holds. We show this in five corresponding proofs (one for each axiom, respectively) by establishing the equalities for norm . As these five proofs are long standard structural inductions we have sourced them out to the Appendix B (Pages 239–266). The five equalities for the constructor axioms imply that the reduction system which bases on the constructor axioms and the function norm has the Church-Rosser property [BN98]. As a consequence of the Church-Rosser property the term normal form (of terms build from constructors only) realized by the function norm is unique. In particular, the function norm yields already this unique normal form since a second application of norm does not change the initial result anymore. In consequence, this implies that also the function NF yields a unique term representation of its argument, as the function term only implements a structure-preserving transformation from elements of sort $\text{SPF } \alpha$ to their corresponding term representations of sort $\text{Terms}(\Sigma_{\text{SPFC}})$. \square

2. Formalization of Characteristic Software Product Family Concepts

2.2.3.3. Configuration: Derivation of Products

A fundamental construction concept of a software product family is that the (specifications of the) products can be derived from the corresponding (specification of the) product family. This process is called *configuration process*. It requires to select for every variation point in the representation of the product family one of its variants. This particular variant will take the place of the corresponding variation point in the resulting product. We call such a selection a *configuration*, i.e. a configuration assigns to some variation points one of their variants, respectively. The following kinds of configurations are of particular interest.

Definition 2.4 (Complete and Atomic Configuration). *With respect to a concrete product family we call a configuration complete if it associates to every variation point that exists in the normal form of the product family one of its variants. A configuration is called atomic if it configures a single variation point only.*

Atomic configurations affect single variation points. Since the variants operator (as introduced so far) does only allow to model the choice between two variants, we represent the act of selecting one of either variants of a specific variation point by two functions

$$\text{selL} : \text{Nat}, \text{SPF } \alpha \rightarrow \text{SPF } \alpha \quad (2.39)$$

$$\text{selR} : \text{Nat}, \text{SPF } \alpha \rightarrow \text{SPF } \alpha \quad (2.40)$$

We call the functions *selection operators*. The arguments of both functions represent the identifier (number) of the variation point for which the respective variant will be selected, and an element of sort $\text{SPF } \alpha$ which represents the entire software product family in which the configuration has to be performed. The function selL (abbreviation for “select Left variant”) selects the left variant for the variants operator with the corresponding identifier, and the function selR (abbreviation for “select Right variant”) selects the right variant, respectively. The corresponding axioms which characterize the way in which a configuration takes place in detail are given in the following on Page 62. The single application of a selection operator corresponds to performing an atomic configuration.

Configurations can be denoted in many ways, e.g. as a function associating variants to variation points, or in a set-like form consisting of tuples of identifiers for variation points and variants. In our setting—in order to match the selection operators—we denote the *configuration choice* for a single variation point as a tuple

$$(VP, Var) \in \text{Nat} \times \{L, R\}$$

2.2. Axiomatization of Software Product Family Concepts

which consists of an identifier $VP \in \mathbf{Nat}$ for the variation point and an identifier $Var \in \{L, R\}$ for the concrete variant, where L denotes the left and R the right variant, respectively. We call the configuration choice for a single variation point *elementary* or *atomic*. Accordingly, we denote an (arbitrary) configuration as an (unordered) set

$$\{(VP_i, Var_i), \dots, (VP_n, Var_n)\}$$

of elementary configuration choices. With *CONFIGS* we denote the set of all (arbitrary) configurations. With respect to a product family with a fixed number n of non-trivial variation points, we denote the set of all *complete* configurations for such a product family by $CONFS^n \subset CONFIGS$. In particular, every configuration in $CONFS^n$ contains exactly n atomic configuration choices, a single one for each (non-trivial) variation point $1, \dots, n$. For the sake of simplicity we usually omit the curly set brackets around an atomic configuration and denote it as a tuple (VP_i, Var_i) instead of the precise notation $\{(VP_i, Var_i)\}$.

In order to denote the application of an entire configuration in our setting we use the function

$$configuration : CONFIGS \times \mathbf{SPF} \alpha \rightarrow \mathbf{SPF} \alpha \quad (2.41)$$

It simply performs an entire configuration on the representation of a product family, i.e. it applies the selection operators **selL** and **selR** according to the corresponding configuration $\{(i, Var_i) : i \leq n\} \in CONFS^n$ in the following way:

$$configuration\left(\{(i, Var_i) : i \leq n\}, PF\right) = f_1\left(1, f_2(2, \dots f_n(n, PF) \dots)\right)$$

$$\text{where } f_i = \begin{cases} \mathbf{selL} & , Var_i = L \\ \mathbf{selR} & , Var_i = R \end{cases}$$

The result of applying a configuration and in particular the corresponding selection operators to a software product family is again an element of sort $\mathbf{SPF} \alpha$, representing the software product family where the respective variation points are replaced by the selected variants, respectively, while the remaining structure of the product family is not changed. The way in which the configuration takes place is well-defined. A single selection operator affects only the corresponding variation point, but leaves the remaining variation points and the higher-level structure of the product family unchanged. The laws which characterize the behavior of the selection operators are given by the following axioms:

2. Formalization of Characteristic Software Product Family Concepts

$$\text{selR}(n, \text{ntrl}) = \text{ntrl} \quad (\text{A-6})$$

$$\text{selL}(n, \text{ntrl}) = \text{ntrl} \quad (\text{A-7})$$

$$\text{selR}(n, \text{asset}(a)) = \text{asset}(a) \quad (\text{A-8})$$

$$\text{selL}(n, \text{asset}(a)) = \text{asset}(a) \quad (\text{A-9})$$

$$\text{selR}(n, P \parallel Q) = \text{selR}(n, P) \parallel \text{selR}(n, Q) \quad (\text{A-10})$$

$$\text{selL}(n, P \parallel Q) = \text{selL}(n, P) \parallel \text{selL}(n, Q) \quad (\text{A-11})$$

$$\text{selR}(n, P \oplus_m Q) = \begin{cases} \text{selR}(n, Q) & , m = n \\ \text{selR}(n, P) \oplus_m \text{selR}(n, Q) & , \text{else} \end{cases} \quad (\text{A-12})$$

$$\text{selL}(n, P \oplus_m Q) = \begin{cases} \text{selL}(n, P) & , m = n \\ \text{selL}(n, P) \oplus_m \text{selL}(n, Q) & , \text{else} \end{cases} \quad (\text{A-13})$$

The Axioms A-12 and A-13 express that all intended variations points (where the number of the variation point matches the first argument of the selection operator) are replaced by the respective variant (the left one in the case of `selL`, and the right one in the case of `selR`), while all variation points with non-matching numbers are ignored in the sense that these variation points are preserved and the configuration is continued recursively on its variants. Since due to the distributivity and the idempotence (Axioms A-3 and A-5) variation point identifiers can (legally) exist several times in an element of sort `SPF` α , the selection operators are still recursively applied to the variants in the case where a variation point is actually replaced by one of its variants. In particular this means that the upper line in the case differentiation of Axiom A-12 (resp. A-13) is `selR`(n, Q) (resp. `selL`(n, P)) and not only Q (resp. P). Thus, any configuration decision represented by a corresponding selection operator only affects the variation point for which it was intended, i.e. the one labeled with the same number as the first argument of the selection operator.

The remaining axioms realize an recursive traversal preserving the structure of the element as a result of the selection operator. In particular, the Axioms A-10–A-11 guarantee that performing a configuration does not affect the structure of compound elements. If the product family consists of atomic assets or the neutral element only, the application of any configuration will always yield these assets/neutral element (cf. A-8–A-7). In summary, the Axioms A-6–A-13 guarantee that the effect of the selection operators is to replace all occurrences of variation points with a matching number by the selected variant.

The Equivalence Relation Induced by the Constructor Axioms is a Congruence for the Variants Operators

2.2. Axiomatization of Software Product Family Concepts

The selection operators have input and result arguments of sort $\text{SPF } \alpha$. In the light of the preceding Sections 2.2.3.1 and 2.2.3.2, we have to guarantee that for equivalent elements of sort $\text{SPF } \alpha$, i.e. elements which have the same term normal form and thus represent the same element of the underlying computation structure, the respective results of the selection operators are again equivalent and also have the same term normal form. This means that the equivalence relation induced by the constructor axioms (cf. Section 2.2.3.1) is indeed a congruence with respect to the selection operators. Thus, we have to show for both selection operators and for every constructor axiom (Axioms A-1–A-5) laws of the kind

$$(\text{NF}(Y) = \text{NF}(Z)) \Rightarrow \left(\text{NF}(\text{selX}(n, Y)) = \text{NF}(\text{selX}(n, Z)) \right) \quad (2.42)$$

where (i) selX represents one of selection operators selR , selL , respectively, and (ii) Y and Z represent the left-hand and the right-hand side of one of the Constructor Axioms A-1–A-5, respectively.

Proof Sketch. We check for every constructor axiom separately whether it specifies a congruence relation with respect to both selection operators. For the constructor Axioms A-1, A-2, and A-4, this is easy to see: Since the application of the selection operators results in term-equivalent elements in these cases (cf. Axioms A-6–A-11), the conclusion of Equation 2.42 also holds since term-equivalent elements always have the same term normal form.

For the constructor axioms involving the variants operator (Axioms A-3 and A-5) this also holds but is not directly obvious, as the corresponding laws of the selection operators (Axioms A-12 and A-13) not necessarily result in equivalent terms. As a representative for both of these axioms we consider the situation for the distributive law (Axiom A-3) in more detail. In the case of the distributive law the left-hand and the right-hand side of the equality in the conclusion of Equation 2.42 are $\text{NF}(\text{selL}(n, (P \parallel Q) \oplus_i (P \parallel R)))$ and $\text{NF}(\text{selL}(n, P \parallel (Q \oplus_i R)))$, where we exemplarily consider only the selection operator selL . In the case of $n = i$ both sides actually represent the same term, as the following equations show:

$$\begin{aligned} \text{selL}\left(n, (P \parallel Q) \oplus_i (P \parallel R)\right) &\stackrel{A-13}{=} \text{selL}(n, P \parallel Q) \\ &\stackrel{A-11}{=} \text{selL}(n, P) \parallel \text{selL}(n, Q) \end{aligned} \quad (2.43)$$

$$\begin{aligned} \text{selL}\left(n, P \parallel (Q \oplus_i R)\right) &\stackrel{A-11}{=} \text{selL}(n, P) \parallel \text{selL}\left(n, (Q \oplus_i R)\right) \\ &\stackrel{A-13}{=} \text{selL}(n, P) \parallel \text{selL}(n, Q) \end{aligned} \quad (2.44)$$

2. Formalization of Characteristic Software Product Family Concepts

Since both results are term-equivalent, they also have the same normal form, which makes the conclusion of Equation 2.42 true. The case for $n \neq i$ can be shown analogously. Also, the situation for the selection operator `selR` is similar. Thus, the equivalence induced by the distributive law is a congruence with respect to the selection operators. The situation for the idempotence law (Axiom A-5) is analogous. \square

In summary, this means that the equivalence relation induced by the constructor Axioms A-1–A-5 is indeed a congruence for the selection operators.

Order of the application of the variants operators

The configuration processes of *different* variation points are independent. More precisely, the (temporal) order in which selection operations concerning *different* variation points are performed is irrelevant for the configured product. This property of the configuration process is expressed by the following laws:

$$\text{selL}(m, \text{selL}(n, P)) = \text{selL}(n, \text{selL}(m, P)) \quad (\text{T-5})$$

$$\text{selR}(m, \text{selR}(n, P)) = \text{selR}(n, \text{selR}(m, P)) \quad (\text{T-6})$$

$$\left(\text{selL}(m, \text{selR}(n, P)) = \text{selR}(n, \text{selL}(m, P)) \right) \Leftrightarrow (m \neq n) \quad (\text{T-7})$$

These laws do not have to be required as axioms as we can derive them from the existing laws.

Proof. Since the functions `selL` and `selR` are axiomized according to *primitive recursive* scheme, we prove the respective Theorems T-5–T-7 via induction over the term structure. The base cases for the constructors `ntrl` and `asset(a)` follow directly from the respective definitions of the functions `selL` and `selR` (Axioms A-8–A-9 and A-6–A-7). The case for the composition operator follows from the definition (Axioms A-10–A-11) and the induction hypothesis. The same holds for the variation operation, which follows from the definition (Axioms A-12–A-13; where only the case $m \neq n$ has to be considered) and the induction hypothesis. \square

The laws T-5–T-7 allow to interchange the application of atomic selection operators in an arbitrary way. In particular, this is also the reason why a (non-atomic) configuration can be denoted as a *set* which does not imply an order of the elements, and does not have to be denoted as e.g. a sequence which comes with a clear order of its elements.

2.2. Axiomatization of Software Product Family Concepts

In contrast to the configuration of different variation points, multiple occurrences of selection operators concerning the *same* variation point *must not* be interchanged. More precisely, the (temporal) order in which selection operators referring to the *same* variation point are applied is essential for the resulting product. This means that in general we have the following inequality:

$$\text{selL}(i, \text{selR}(i, P)) \neq \text{selR}(i, \text{selL}(i, P))$$

The reason for this is that once we have selected a certain variant—i.e. once we have applied either `selL` or `selR` for a variation point—the resulting product family is fixed with respect to this variation point. In particular, we cannot undo our configuration selection for this particular variation point once it is performed. This reflects our intuitive understanding of how a (realistic) configuration process takes place: Once we have configured a variation point, any preceding configuration decision for the *same* variation points is irrelevant. Seen from a more theoretical point of view, if we would allow that the order of selection operations could be changed, i.e. if $\text{selL}(i, \text{selR}(i, P)) = \text{selR}(i, \text{selL}(i, P))$ would hold, then the process of configuration would not yield a unique product anymore, as we could prove the existence of at least two different products by applying the equality.

After applying a complete configuration to a product family, the product family contains no variation points anymore since every variation point is replaced by the selected variant. In particular, configuring a product family with a complete configuration always yields a product in a unique way.

2.2.3.4. Properties of the Variants Operator

(Non-) Associativity

An essential property for the configuration process of a software product family is that a specific configuration always yields the same product or sub-family. In order to guarantee this we cannot allow that the variants operator is associative since an associative variants operator would introduce an ambiguity in the mapping between variants and variation points. More precisely, consider the product family, where we assume that $i \neq j$.

$$P \oplus_i (Q \oplus_j R)$$

Here, the variants of each variation point are precisely specified: The variants of the variation point i are the product families P and $(Q \oplus_j R)$, respectively, while the variants of variation point j are the product families Q and R . Now, imagine an associative variants operator, i.e. where the law $P \oplus_i (Q \oplus_j R) = (P \oplus_i Q) \oplus_j R$ would hold. For such an associative variants operator, by applying the associative

2. Formalization of Characteristic Software Product Family Concepts

law, the association between the variants and the variation points is altered: After applying the law, the variants of variation point i would be P and Q , while the variants of the variation point j would then be $(P \oplus_i Q)$ and R . In particular, the same configuration $\{(i, L), (j, R)\}$ would result in different products, depending on whether we apply the associative law or not:

$$\begin{aligned} & \text{selL}\left(i, \text{selR}(j, P \oplus_i (Q \oplus_j R))\right) & \text{selL}\left(i, \text{selR}(j, (P \oplus_i Q) \oplus_j R)\right) & = \\ = & \text{selL}(i, \text{selR}(j, P)) & \neq & \text{selL}(i, \text{selR}(j, R)) \end{aligned}$$

The last line is an inequality since in general we cannot assume that $P = R$. Certainly, the property that the same configuration can result in different products is not desired, which is why we do not allow the variants operator to be associative.

Due to the non-existence of an associative law for the variants operator, the nesting of variation points induces a hierarchical structure, i.e. variants may again contain variation points, themselves. In particular, without associativity, it is precisely defined to which variation point a variant belongs. The hierarchy induced by the variants operator causes that the configuration decision of some variation points has no effect on the resulting product. To give an example, revisit the product family Spf shown in Figure 2.6: Here, the left variant of the variation point 2 contains the variation point 3. This implies, that for every configuration which does not select the left variant, e.g. the configuration $\{(2, R)\}$, also the variation point 3 will not be included in the final product. In particular, in such a situation the atomic configuration of the (contained) variation point 3 is irrelevant for the resulting product. We can see this easily if we consider every possible atomic configuration of the variation point 3 in combination with selecting the atomic configuration $\{(2, R)\}$. For the product family Spf , both configurations, $\{(2, R), (3, L)\}$ and $\{(2, R), (3, R)\}$, result in the same product family $SpfFem$, i.e.

$$\text{configure}\left(\{(2, R), (3, L)\}, Spf\right) = SpfFem = \text{configure}\left(\{(2, R), (3, R)\}, Spf\right)$$

which follows from applying the respective laws A-12–A-7.

This gives reasons for an equivalence relation \sim on the set of all possible complete configurations of a product family: two complete configurations $c_1, c_2 \in CONFS^n$ are equivalent for a given product family if they result in an identical product (Recall, that $CONFS^n$ denotes the set of all complete configurations for a product family with n variation points).

Definition 2.5 (Equivalence of Complete Configurations). *Let $PF \in SPF$ α be a product family with $n \in \text{Nat}$ non-trivial variation points, and $c_1, c_2 \in CONFS^n$ be complete configurations for PF . We call c_1 and c_2 equivalent iff they result in the same product when applied to PF , i.e. if*

$$\text{configure}(c_1, PF) = \text{configure}(c_2, PF)$$

2.2. Axiomatization of Software Product Family Concepts

All (complete) configurations which are equivalent according to Definition 2.5 form an equivalence class, which we denote by

$$[c] := \{c' \in \text{CONFS}^n \mid c' \sim c\}$$

Let us summarize the connection between products and configurations. We observe that (i) any complete configuration corresponds to exactly one product, but (ii) a product can be represented by many (complete) configurations. Thus any product corresponds to an equivalence class of complete configurations.

(Non-) Commutativity

Since the selection operators `selL` and `selR` identify the variants according to the side of the respective variants operator on which they appear, (only) our variants operator is *not* commutative. If we would allow commutativity for our version of the variants operator, i.e. if $P \oplus_i Q = Q \oplus_i P$ would hold, then the selection operators `selL` and `selR` would not yield the desired result—the desired variant in a unique way—as the following example illustrates:

$$\text{selL}(i, P \oplus_i Q) = \text{selL}(i, P) \neq \text{selL}(i, Q) = \text{selL}(i, Q \oplus_i P)$$

Certainly, in general the sequence in which the variants of a variation point are specified, is not important. Thus, in general a variants operator *is* commutative, as it only models the choice between variants, and not an ordered one on them. However, a commutative version of a variants operator comes with some overhead concerning the identification of the respective variants. Thus—for the sake of simplicity—we have preferred to abandon commutativity and decided to introduce a variants operator which identifies its variants according to the side of the operator on which they appear, being aware about the non-commutativity.

2.2.3.5. Sub-Families

The selection operators allow to configure an entire software product family successively by configuring variation points independently in an arbitrary order. This gives reasons for the notion of a *sub-family*. Intuitively, a sub-family is a product family—and thus an element of sort `SPF` α —which was derived from another product family that contains a superset of (non-trivial) variation points, by configuring some of the variation points of the original product family. More precisely, we say that a product family PF' is a *sub-family* of another product family PF if there exists a composition of selection operators—represented by a configuration—which yields the product family PF' when applied to the original product family PF .

2. Formalization of Characteristic Software Product Family Concepts

Definition 2.6 (Sub-Family). *Let $PF', PF \in \text{SPF } \alpha$ be two product families. We call PF' a sub-family of PF if*

$$\exists c \in \text{CONFIGS} : \text{NF}(\text{configure}(c, PF)) = \text{NF}(PF')$$

We say that the sub-family PF' is a partially configured version of PF . A sub-family that still contains non-trivial variation points is called a true sub-family.

Compared to the “original” family PF (where not all variation points have been configured yet), a sub-family PF' represents only a certain subset of products, as not all products which could be derived from the original one can still be derived from the sub-family product family. By comparing the normal forms of both product families we abstract from their concrete term representation, respectively. According to Definition 2.6 a sub-family itself does not necessarily have to contain variation points, anymore. In particular, completely configured products also fulfill the definition of sub-families. However, usually we assume that a sub-family still contains (non-trivial) variation points, and is not yet a product.

Consider for example the product family Spf given in Figure 2.3 together with its realization in the computation structure of stickmen drawings (cf. Figure 2.4). Imagine we are only interested in stickman drawings with female torsos. This requires to select the variant `Female_Torso` for the variation point 2, and results in a product family $SpfFem$ of stickmen with female torsos. We can represent this family easily by deriving it as a sub-family of the original product family Spf . The corresponding configuration consists of the single atomic configuration $(R, 2)$ only. The derivation of the sub-family $SpfFem$ is shown in Figure 2.6.

Figure 2.7 shows the actualization of $SpfFem$ in the computation structure of stickmen drawings. The product family $SpfFem$ has only one variation point and thus represents only two products. Note that $SpfFem$ is a self-contained product family which can be considered independently from other product families. However, due to deriving it as a sub-family from Spf the formal connection to Spf is precisely given, which allows to reason about $SpfFem$ also in the context of Spf .

For the application of software product family concepts in a realistic context the concept of a sub-family is very important. Although it is essential that the software product family “covers” the set of all derivable products in their entirety, sometimes the full model of a software product family is too complex to be considered in its full extent for specific situations (e.g. in an automotive OEM/supplier scenario). In order to cope with the complexity of real-life software product families, the capability to consider only relevant parts of a software product family is essential. In a software product family, this manifests very often in the situation that only a subset of all derivable products has to be considered. Here, the notion of a sub-family gives the necessary conceptual foundation in order to define an abstraction mechanism as a methodological concept.

2.2. Axiomatization of Software Product Family Concepts

$$\begin{aligned}
 SpfFem & := \text{configure}(\{(R, 2)\}, Spf) \\
 & = \text{selR}(2, Spf) \\
 & \stackrel{A-10}{=} \text{selR}\left(2, \left(\text{asset}(\text{Coffee}) \oplus_3 \text{asset}(\text{Left_Hand})\right) \parallel \text{Male_Torso}\right) \\
 & \quad \oplus_2 \text{asset}(\text{Female_Torso}) \\
 & \quad \parallel \text{selR}\left(2, \left(\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})\right) \parallel \text{asset}(\text{Face})\right) \\
 & \quad \parallel \text{selR}(2, \text{asset}(\text{Legs})) \\
 & \stackrel{A-8, A-10, A-12}{=} \text{selR}(2, \text{asset}(\text{Female_Torso})) \\
 & \quad \parallel \left(\text{selR}(2, \text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})) \parallel \text{selR}(2, \text{asset}(\text{Face}))\right) \\
 & \quad \parallel \text{asset}(\text{Legs}) \\
 & \stackrel{A-8, A-12}{=} \text{asset}(\text{Female_Torso}) \\
 & \quad \parallel \left(\text{selR}(2, \text{asset}(\text{Smiling})) \oplus_1 \text{selR}(2, \text{asset}(\text{Sad}))\right) \parallel \text{asset}(\text{Face}) \\
 & \quad \parallel \text{asset}(\text{Legs}) \\
 & \stackrel{A-8}{=} \text{asset}(\text{Female_Torso}) \\
 & \quad \parallel \left(\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})\right) \parallel \text{asset}(\text{Face}) \\
 & \quad \parallel \text{asset}(\text{Legs})
 \end{aligned}$$

Figure 2.6.: The product family $SpfFem$ is derived as a sub-family of Spf .

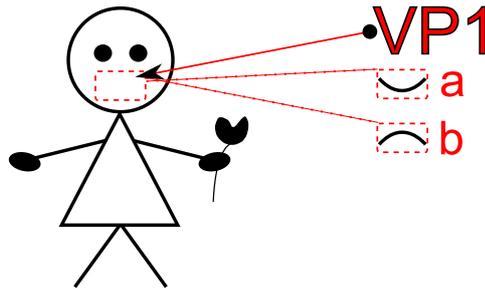


Figure 2.7.: The actualization of the product family $SpfFem$ (cf. Figure 2.6) in the concrete computation structure of stickmen drawings. The product family $SpfFem$ is a true sub-family of the original product family Spf . It contains only stickman drawings with female torsos.

2. Formalization of Characteristic Software Product Family Concepts

2.2.3.6. Products

The representations of individual products can be derived from the representation of a software product family by configuration. Intuitively, products are those configured instances of a software product family which have no variable parts and which thus can exist in reality. Compared to an arbitrary element of sort $\text{SPF } \alpha$, the distinctive property of a *product* is that it does not contain any more choices between (non-trivial) variants.

However, not only complete configurations yield products. For example, consider the incomplete configuration $\{(2, R), (1, L)\}$ for the product family Spf :

$$\begin{aligned} & \text{configure}\left(\{(2, R), (1, L)\}, \text{Spf}\right) \\ = & \text{selR}(2, \text{selL}(1, \text{Spf})) \\ = & \text{asset}(\text{Female_Torso}) \parallel (\text{asset}(\text{Smiling}) \parallel \text{asset}(\text{Face})) \parallel \text{asset}(\text{Legs}) \end{aligned}$$

Although being incomplete—the atomic configuration for the variation point 3 is missing—the configuration still yields a product which does not contain any variation points anymore. This is possible since the variation operators induce a hierarchic structure (due to the missing of an associative law for \oplus). In particular, a variant P of a variation point $P \oplus_i Q$ may again contain variation points, itself. This means that by *not* selecting P , the contained variation point will not be included in the final product, either, and does therefore also not require to be configured. Thus, while a complete configuration always yields a product, there are also incomplete configurations which already yield products.

Another issue is that some variation points do not offer a true choice, i.e. if an element of sort $\text{SPF } \alpha$ contains trivial variation points of the kind $P \oplus_i P$, according to the idempotence law (Axiom A-4) configuring this particular variation point always results in the same variant P . In this light, trivial variation points actually do not represent variability. Thus, a product can contain variation points—as long as all of them are trivial—and still fulfill our initial property of not containing variability any more. Note that performing the configuration on the normal form of an element avoids such trivial variation points, as we have mentioned before in Section 2.2.3.2.

In a realistic application scenario the ability to determine whether a (partially) configured product family is already a product or whether it still contains unconfigured, non-trivial variation points, is essential. For such a situation we provide a function `is_product` which characterizes products in general, i.e. it returns the answer whether a given product family of sort $\text{SPF } \alpha$ is actually a product or not by testing for variable parts.

$$\text{is_product} : \text{SPF } \alpha \rightarrow \text{Bool} \quad (2.45)$$

The result of the function `is_product` is a boolean value *true* or *false*. We represent the Boolean values by the sort `Bool`, which is defined by means of an algebraic specification that can be found in Appendix A.1 on Page 232. The function `is_product` precisely defines our notion of a product: While every product itself is also an element of sort `SPF` α , it still is a special kind of a product family since a product does not contain any non-trivial variation points anymore. This is expressed by the following axioms:

$$\text{is_product}(\text{ntrl}) = \text{true} \quad (\text{A-14})$$

$$\text{is_product}(\text{asset}(a)) = \text{true} \quad (\text{A-15})$$

$$\text{is_product}(P \parallel Q) = \text{is_product}(P) \wedge \text{is_product}(Q) \quad (\text{A-16})$$

$$\text{is_product}(P \oplus_i Q) = \begin{cases} \text{false} & , \text{NF}(P) \neq \text{NF}(Q) \\ \text{is_product}(P) & , \text{NF}(P) = \text{NF}(Q) \end{cases} \quad (\text{A-17})$$

The function recursively checks if an element of sort `SPF` α , e.g. a sub-family, contains non-trivial variation points. Any element, which contains such non-trivial variation points is *not* a product. Regarding Axiom A-14, although representing the concept of the *empty product family*, the element `ntrl` in its own rights is still a product in the sense that it does not contain any variability and requires no further configuration decisions. Regarding Axiom A-17, we distinguish between trivial and non-trivial variation points by checking for every variation point if its variants are equivalent or not. For this check we use the function `NF`, which represents the normal form of an element of sort `SPF` α . Thus, an element $P \oplus_i Q$ is actually a product if the normal form of both variants is the same, and if the variants themselves, w.l.o.g. represented by the left variant P , are products, too.

In summary, the function `is_product` checks whether an element of sort `SPF` α contains variability. Only those elements which do not contain any variable parts are called *products*. Technically, the check is performed by testing for non-trivial variation points. In particular, this means that we consider an element such as for example $P \oplus_1 P$ to be a product, even though its term representation contains a (trivial) variation point.

Set of Derivable Products

Characteristic for any software product family is its set of derivable products.

2. Formalization of Characteristic Software Product Family Concepts

Definition 2.7 (Set of Derivable Products). *The set $PRODUCTS_{PF}$ is the set of all products which are derivable from a software product family PF containing n non-trivial variation points, i.e.*

$$PRODUCTS_{PF} := \{NF(P) \in SPF \ \alpha \mid \exists c \in CONFS^n : configure(c, PF) = P\}$$

A set $PRODUCTS_{PF}$ only contains completely configured products of PF , since we only consider complete configurations in $CONFS^n$ which (by construction) always yield products. Since some of the possible products may be equivalent with respect to the equivalence relation (cf. Section 2.2.3.2) induced by the constructor axioms (cf. Section 2.2.3.1), we consider the set of normal forms of all possible configurations in Definition 2.7. Further, the set $PRODUCTS_{PF}$ is constructed by applying all combinatorially possible configurations to the respective product family PF . Since by definition there are only finitely many non-trivial variation points (with finitely many variants, respectively) the set $PRODUCTS_{PF}$ is actually constructible for any product family PF in finite time.

Beside the notion of equivalence of software product families of sort $SPF \ \alpha$ which is based on their normal form (cf. Section 2.2.3.2), we define another notion of equivalence based on the set of derivable products. For a given element of sort $SPF \ \alpha$, we consider two elements P, Q of sort $SPF \ \alpha$ *equal* if both allow to derive the same set of products.

Definition 2.8 (Product Equivalence of Software Product Families). *Let $P, Q \in SPF \ \alpha$ be two software product families. We consider both software product families equal iff both represent the same set of derivable products, i.e.*

$$PRODUCTS_P = PRODUCTS_Q$$

2.2.3.7. Common Parts

The explicit representation of common parts is at the heart of a software product family. The distributivity of the composition over the variants operator (Axiom A-3) is the conceptual key to deal with commonalities within a software product family, and in particular to define the notion of a *common part* of two variants. Recall the distributive law

$$(P \parallel Q) \oplus_i (P \parallel R) = P \parallel (Q \oplus_i R) \tag{A-3}$$

The distributive laws give reason to the concept of *common parts* of variants.

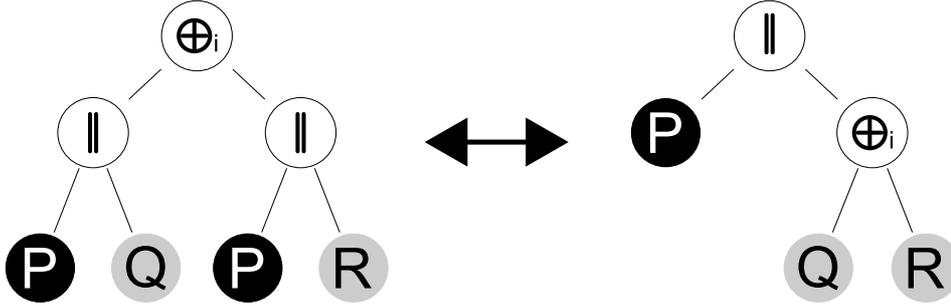


Figure 2.8.: Graphical illustration of the distributive law of \oplus over \parallel . It shows the effect of Axiom A-3 for the tree representation. The element P represents the direct common part of both variants.

Definition 2.9 (Common Part of Variants). *Let $Q, R \in \text{SPF } \alpha$ be arbitrary product families. We call any element $P \in \text{SPF } \alpha$, $P \neq \text{ntrl}$, a common part of the variants of variation point $A \oplus_i B$, if P exists in both variants A and B in such a way that P can be factored out by means of the distributive laws (Axiom A-3, Thm. T-1–T-4).*

Figure 2.8 illustrates the distributive law and the idea of common parts. It shows the effect of applying the distributive law for the tree representation of Axiom A-3. After applying the law, the common element P is factored out and the configuration selection has to be performed only between the variant-specific parts Q and R . Note that the common part is an arbitrary element (except for the neutral element) of sort $\text{SPF } \alpha$, which means that it does not necessarily have to be an asset. Thus, even sub-families can be common parts of two variants.

Although it is not a special law from an algebraic point of view, the distributive law is of immense importance for the realistic applicability of software product line engineering: It describes (i) how we can represent two variants as a composition of their common and their variant-specific parts, and (ii) gives a restructuring/rewriting rule how to factor out their common part. In order to emphasize the practical importance of this law we want to make one step back from our technical treatment and recall Dijkstra’s vision [DDH72], we have already given in the introduction:

If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say, different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common.

If we understand two similar systems (programs) as variants of a common variation point, the distributive law represents a restructuring rule that allows us to factor out

2. Formalization of Characteristic Software Product Family Concepts

the common part and push the variation point deeper (towards the leaves) in the term hierarchy. The resulting representation (right-hand side of the distributivity axiom) represents Dijkstra’s idea of an “abstract program”, where the common part is explicitly represented and the configuration selection has only to be performed between the different parts of both variants.

Example: Common Parts of the Stickmen Drawings

In order to illustrate the extraction of common parts by applying the distributive law we consider the product family *SpfII*.

$$SpfII := \left(\begin{array}{l} ((\text{asset}(\text{Coffee}) \oplus_3 \text{asset}(\text{Left_Hand})) \\ \parallel \text{asset}(\text{Male_Torso_II}) \parallel \text{asset}(\text{Right_Hand})) \\ \oplus_2 (\text{asset}(\text{Female_Torso_II}) \parallel \text{asset}(\text{Right_Hand})) \end{array} \right) \\ \parallel (\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})) \parallel \text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs})$$

The product family *SpfII* is a slight modification of the product family *Spf*. It uses most of the assets known from *Spf*, and the three new assets named *Female_Torso_II*, *Male_Torso_II*, and *Right_Hand*, for which we assume the sort *Stickman* to contain the respective constants. Note that compared to *Spf*, the asset *asset(Right_Hand)* is now an independent asset and forms no longer an atomic asset together with the torsos. Since the asset *asset(Right_Hand)* is contained in both variants of the variation point 2, we can apply the distributive law to factor out this common part as shown in the following calculation. This results in a restructured version of the product family *SpfII* which we denote with *SpfIII*.

$$SpfII = \left(\begin{array}{l} ((\text{asset}(\text{Coffee}) \oplus_3 \text{asset}(\text{Left_Hand})) \\ \parallel \text{asset}(\text{Male_Torso_II}) \parallel \text{asset}(\text{Right_Hand})) \\ \oplus_2 (\text{asset}(\text{Female_Torso_II}) \parallel \text{asset}(\text{Right_Hand})) \end{array} \right) \\ \parallel (\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})) \parallel \text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs}) \\ \stackrel{A-3}{=} \left(\begin{array}{l} ((\text{asset}(\text{Coffee}) \oplus_3 \text{asset}(\text{Left_Hand})) \parallel \text{asset}(\text{Male_Torso_II})) \\ \oplus_2 \text{asset}(\text{Female_Torso_II}) \\ \parallel \text{asset}(\text{Right_Hand}) \end{array} \right) \\ \parallel (\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})) \parallel \text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs}) \\ =: SpfIII$$

Let *StickmanShapesII* be a concrete computation structure that realizes the three new shapes and the remaining constants from the sort *Stickman* as concrete graphical

shapes as shown in Figure 2.9. Figure 2.10a shows the product family $SpfII$ as realized in `StickmanShapesII`. We see the effect of the distributive law in Figure 2.10b, which shows the product family $SpfII$ in its restructured form $SpfIII$, as it is realized in `StickmanShapesII`. After the restructuring (Figure 2.10b) the common part `Right_Hand` is now explicitly included in every product, since it is composed to the variation point 2 directly, and does not appear as part of both variants anymore.

2.2.3.8. Optional and Mandatory Parts

Typical for a software product family is the concept of *optional* and *mandatory* parts. Intuitively, a part is called *optional* if there are at least two variants that themselves differ only by the optional part, i.e. one variant comprises the entire other variant and additionally the optional part. The idea is that an optional part is not essentially required for every variant and can be added if desired. In contrast, we call a part *mandatory* if its existence is not depending on any configuration choice, i.e. if it is present in every possible variant.

Optional Parts

Optionality of variants is a special case of alternative existence of variants. We model an optional part, i.e. an optional variant, as a special case of a variation point where one variant represents an “empty” product family. For this purpose we use the *neutral element*, which allows to represent an optional element $O \in \text{SPF } \alpha$ by means of a variation point of the following kind

$$O \oplus_i \text{ntrl}$$

Depending on the atomic configuration of the corresponding variation point i this means that either (i) the element O is selected and thus present in the resulting product family, or (ii) the neutral element `ntrl` is selected which means that the resulting product family effectively remains unchanged as described by Axiom A-4. Consider for example the following product family (We assume that P and O themselves do not contain a variation point with label i).

$$P \parallel (O \oplus_i \text{ntrl})$$

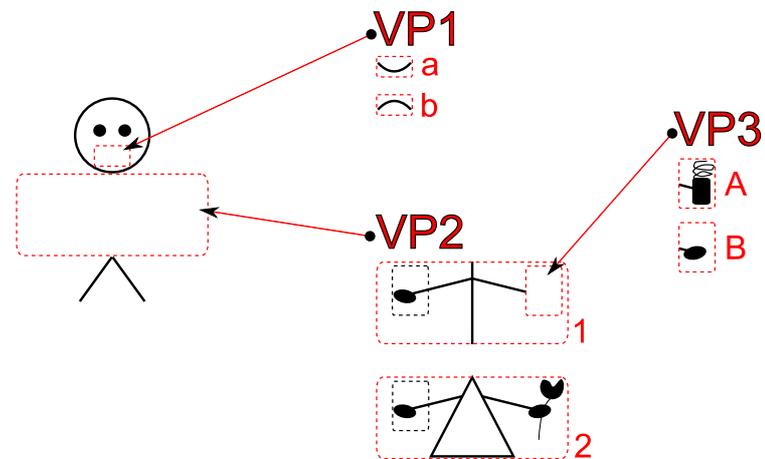
For the atomic configuration (i, L) the product family becomes

$$\begin{aligned} \text{selL}(i, (P \parallel (O \oplus_i \text{ntrl}))) &\stackrel{A-11}{=} \text{selL}(i, P) \parallel \text{selL}(i, (O \oplus_i \text{ntrl})) \\ &\stackrel{A-13}{=} \text{selL}(i, P) \parallel \text{selL}(i, O) \\ &\stackrel{T-8}{=} P \parallel O \end{aligned}$$

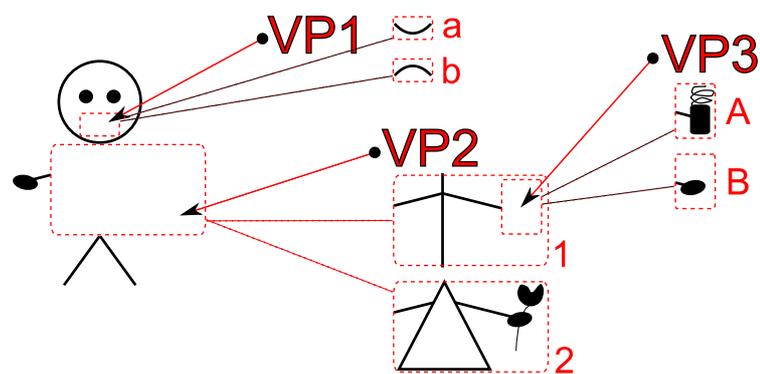
2. Formalization of Characteristic Software Product Family Concepts

Face	Legs	Smiling	Sad	Normal
Female_Torso_II	Male_Torso_II	Coffee	Left_Hand	Right_Hand

Figure 2.9.: The actualization of core assets of sort SPF Stickman in the computation structure StickmanShapesII.



(a) The family *SpfII* as realized in the computation structure StickmanShapesII.



(b) The product family *SpfII*. Compared to (a), the common part (asset Right_Hand) was factored out by means of the distributive law.

Figure 2.10.: Illustration of the product family *SpfII* before and after the application of the distributive law.

2.2. Axiomatization of Software Product Family Concepts

while for the atomic configuration (i, R) it becomes

$$\begin{aligned} \text{selR}(i, (P \parallel (O \oplus_i \text{ntrl}))) &\stackrel{A-10}{=} \text{selR}(i, P) \parallel \text{selR}(i, (O \oplus_i \text{ntrl})) \\ &\stackrel{A-12}{=} \text{selR}(i, P) \parallel \text{selR}(i, \text{ntrl}) \\ &\stackrel{A-6, T-9}{=} P \parallel \text{ntrl} \\ &\stackrel{A-4}{=} P \end{aligned}$$

Thus, the difference between both versions is only the element O , which is either completely included or not. Regarding the variants, the variant $P \parallel O$ comprises exactly the other variant P and additionally the optional part O .

Note that defining the optional part as the left-hand side argument of the variation point is an arbitrary choice. Switching the positions of the optional part and the element ntrl , i.e. specifying an optional part O as $\text{ntrl} \oplus_i O$ instead of $O \oplus_i \text{ntrl}$, is equally possible. Whatever version is used, the configuration of corresponding variation point i has to be adjusted accordingly. In order to abstract from this irrelevant detail—and as a kind of syntactic sugar—we introduce the function `optional` which is just a more comfortable way of denoting the corresponding variation point:

$$\text{optional} : \text{SPF } \alpha, \text{ Nat} \rightarrow \text{SPF } \alpha \quad (2.46)$$

Like a variation point, an optional part has also a unique name (represented by the argument of sort `Nat`). Since this name is basically a reference to the name of a variation point, this name has to be unique in the set of identifiers for all variation points in the respective product family. The function `optional` simply wraps an optional part O , essentially abbreviating the specification of an variation point as the corresponding axiom points out:

$$\text{optional}(O, i) = (O \oplus_i \text{ntrl}) \quad (2.47)$$

For example, the following product family has the optional element O :

$$P \parallel \text{optional}(O, i)$$

According to Equation 2.47 this is just an abbreviation for the following specification representing the optionality of the part O by means of a variation point:

$$P \parallel (O \oplus_i \text{ntrl})$$

2. Formalization of Characteristic Software Product Family Concepts

For technical reasons, the optional operator requires a special configuration operator which takes the product family to be configured, the number of the corresponding optional asset, and a boolean value which states whether the optional element will be included (indicated by *true*) or not (indicated by *false*):

$$\text{select} : \text{SPF } \alpha, \text{Nat}, \text{Bool} \rightarrow \text{SPF } \alpha$$

The axiom of this special configuration operator expresses the translation into the configuration for the corresponding variation point:

$$\text{select}(P, i, \text{choice}) = \begin{cases} \text{selL}(i, P) & , \text{choice} = \text{true} \\ \text{selR}(i, P) & , \text{choice} = \text{false} \end{cases}$$

Since the function `optional` and the associated selection operator `select` are merely an abbreviatory notation for the special case of an alternative variation point, we do not include them in the algebraic specification of a software product family (cf. Page 90).

Mandatory Assets and Mandatory Parts

Some core assets are included in every derivable product of a given product family. We call them *mandatory*. Mandatory assets represent the commonality among the products of a product family on the level of basic construction units, and can be used as a measure for reuse. The more mandatory assets exist in a product family, the higher is the degree of reuse. Other assets are not necessarily present in every product, i.e. they exist only in some products. In particular, they are not mandatory. We call such assets *variable*. Note that an optional asset is a special case of a variable one.

Consider for example Figure 2.10b (Page 76) which shows the actualization of the product family *SpfIII*: The asset `asset(Right_Hand)` actually is a mandatory asset of *SpfIII*, since it is part of any derivable product, as the configuration of *SpfIII* according to all six possible configurations shows.

So far, we pursued the idea of being mandatory only for assets. Now, we extend this idea to a more general setting. We consider the question whether arbitrary elements of sort `SPF α` are part of all products which can be derived from a certain software product family. This allows to express that compound objects, or even entire subfamilies are mandatory parts with respect to a larger product family. For this purpose we introduce the function

$$\text{is_mand} : \text{SPF } \alpha, \text{SPF } \alpha \rightarrow \text{Bool} \quad (2.48)$$

2.2. Axiomatization of Software Product Family Concepts

The operation $\text{is_mand}(Z, X)$ expresses that a general element $X \in \text{SPF } \alpha$ is contained as a mandatory part/sub-family in the product family $Z \in \text{SPF } \alpha$. Analogously to the notion of mandatory and variable assets we call an element X a *mandatory part* of Z , if X is part of every product that can be derived from Z . Otherwise, we call X a *variable part* with respect to Z .

Using our running example of stickman drawings, we might for example not only be interested in whether a concrete asset such as a smiling or sad mouth occurs in every product, but rather whether every product actually has a mouth. In the example of the product family *SpfIII* this is equivalent to the question whether every product actually contains the variation point $\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})$ as a mandatory part. The function is_mand provides the means to specify such properties. Note that we express such properties without an explicit *type-mechanism*. A type mechanism is orthogonal to software product family concepts and can be added additionally.

An element $X \in \text{SPF } \alpha$, $X \neq \text{ntrl}$, is mandatory in a product family Z if the product family Z is actually the part X itself, or if X is composed to every product independently from any configuration choice, or if X appears as a mandatory part of both variants, in the case where Z itself is a direct variation point. These ideas are reflected by the following axioms. Note that in order to check whether two elements are actually equal, we compare them on basis of their normal forms (cf. Section 2.2.3.2). In this way, the specification of is_mand becomes independent of the concrete term representation which is used to denote the elements.

$$\text{is_mand}(\text{ntrl}, X) = \text{false} \quad (\text{A-18})$$

$$\text{is_mand}(\text{asset}(a), X) = (\text{NF}(X) = \text{asset}(a)) \quad (\text{A-19})$$

$$\text{is_mand}(P \parallel Q, X) = \begin{cases} \text{false} & , \text{NF}(P \parallel Q) = \text{ntrl} \\ \begin{cases} (\text{NF}(P \parallel Q) = \text{NF}(X)) \\ \vee \text{is_mand}(P, X) \\ \vee \text{is_mand}(Q, X) \end{cases} & , \text{else} \end{cases} \quad (\text{A-20})$$

$$\text{is_mand}(P \oplus_i Q, X) = \begin{cases} \text{false} & , \text{NF}(P \oplus_i Q) = \text{ntrl} \\ \begin{cases} (\text{NF}(P \oplus_i Q) = \text{NF}(X)) \\ \vee \left(\begin{array}{l} \text{is_mand}(P, X) \\ \wedge \text{is_mand}(Q, X) \end{array} \right) \end{cases} & , \text{else} \end{cases} \quad (\text{A-21})$$

For example, by applying the Axioms A-18–A-21 from left to right we can conclude that the compound element $(\text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs}))$ is a mandatory part of the software product family *SpfIII*, i.e.

2. Formalization of Characteristic Software Product Family Concepts

$$\text{is_mand}(SpfIII, \text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs})) = \dots = \text{true}$$

Examining the example of the software product family $SpfIII$ more closely we see that the compound element $(\text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs}))$ is not the “largest” mandatory part of the product family $SpfIII$, yet. We can still add the elements Right_Hand and $(\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad}))$, thus considering the compound element

$$\text{asset}(\text{Right_Hand}) \parallel (\text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad})) \parallel \text{asset}(\text{Face}) \parallel \text{asset}(\text{Legs})$$

This element is also a mandatory part of the product family $SpfIII$. However, it is different to the former one as it represents a *true sub-family*, i.e. it not only contains assets but also a variation point. However, also such general elements are recognized as mandatory parts by the function `is_mand`.

Expressiveness of the Function is_mand

The previous example has shown that the function `is_mand` allows to check whether arbitrary elements of sort $\text{SPF } \alpha$ occur as mandatory parts of a product family of sort $\text{SPF } \alpha$. For example, by evaluating

$$\text{is_mand}(Z, \text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad}))$$

we can check whether every stickman drawing which is derivable from the family Z has either a smiling or a sad face. However, we cannot check whether every stickman drawing actually has a “face”. Simply, because the concept “having a face” is not expressible in the axiomatization. The reason for this is the lack of a *type system* which allows to specify concepts such as “having a face”, for example by grouping the three assets $\text{asset}(\text{Smiling})$, $\text{asset}(\text{Sad})$ and $\text{asset}(\text{Normal})$ together to form the subtype *Faces*. Certainly, the theory of type systems is well understood in the context of programming languages [CW85, LZ74, LW94], and we can easily extend our axiomatization with a type system, that allows to specify such questions.

However, we introduced the axiomatization without a type system since in our opinion a type system represents an orthogonal concept that can be additionally added to the axiomatization if desired, but that is not an essential part which is necessary to define the conceptual construction principle behind a software product family. Therefore, we have not added a type system on top of the axiomatization, being aware that questions such as the one before are not directly expressible within our theory. We will pick up this discussion again at the end of this chapter in Section 2.4.3.

2.2.3.9. Evolution of Software Product Families

As experience from realistic scenarios shows, a software product family constantly evolves [Bos00, TBK09, AGM⁺06] due to various reasons such as changing customer's needs, market development, or emerging technology trends. Often new assets are added, existing assets are modified, or the structure of the product family is changed. We also speak of *software product line evolution* [Bos00]. In our axiomatization we can also represent evolution of software product families, which facilitates further reasoning about the change within software product families.

Uniqueness of Variation Point Identifiers

In the situation where new assets are added by extending the specification of a product family P with new variation points, these variation points have not to be confused with existing variation points of P . In particular, there must not be a name clash with the identifiers of new variation and the identifiers of existing ones, since the selection choices represented by existing variation points have conceptually be made independently from the choices offered by new variation points. Therefore, we name new variation points with identifiers which are fresh in the set of identifiers of P . In order to realize this requirement, and to reason about the existence of certain variation points, a product family has to provide a mechanism that allows to keep track of the variation points included in the product family so far. This is realized by the function `has_ntVP` (has non-trivial variation point):

$$\text{has_ntVP} : \text{NAT, SPF } \alpha \rightarrow \text{Bool} \quad (2.49)$$

$$\text{has_ntVP}(n, \text{ntrl}) = \text{false} \quad (\text{A-22})$$

$$\text{has_ntVP}(n, \text{asset}(a)) = \text{false} \quad (\text{A-23})$$

$$\text{has_ntVP}(n, P \parallel Q) = \text{has_ntVP}(n, P) \vee \text{has_ntVP}(n, Q) \quad (\text{A-24})$$

$$\text{has_ntVP}(n, P \oplus_i Q) = \left(\begin{array}{l} ((n =_{\text{Nat}} i) \wedge (\text{NF}(P) \neq \text{NF}(Q))) \\ \vee \text{has_ntVP}(n, P) \\ \vee \text{has_ntVP}(n, Q) \end{array} \right) \quad (\text{A-25})$$

The operation `has_ntVP(i, P)` returns the result whether a given software product family P contains a variation point with the given label i . For core assets and the neutral element the result is always *false* since both are atomic units which can not contain any other elements of sort $\text{SPF } \alpha$. For composed elements the result depends on whether the composition units contain a variation point with such a

2. Formalization of Characteristic Software Product Family Concepts

label. Regarding Axiom A-25, for variation points, either the variation point itself has such a label, or the result depends on the result of its variants. Thereby, trivial variation points are ignored, i.e. the function `has_ntVP` only respects identifiers of existing *non-trivial* variation points (realized by the check $\text{NF}(P) \neq \text{NF}(Q)$ in the side condition of Axiom A-25).

Knowledge of the variation points which exist in a software product family can be exploited to simplify the configuration process. The following two laws express the fact that any application of a selection operator for a variation point i on a product family which does not contain a variation point with such an identifier, has no effect. In particular when reasoning about the effects of configurations for sub-families such laws are useful.

$$\neg \text{has_ntVP}(n, P) \Rightarrow (\text{selL}(n, P) = P) \quad (\text{T-8})$$

$$\neg \text{has_ntVP}(n, P) \Rightarrow (\text{selR}(n, P) = P) \quad (\text{T-9})$$

Both laws can be derived from the specifications of `has_ntVP` and the selection operators by means of term induction. Exemplarily we consider the proof for Law T-8, as the proof for Law T-9 is analogous.

Proof Sketch. The inductive base cases for the constructors `ntrl` and `asset(a)` follow directly from the respective definitions of the function `selL` (Axioms A-9 and A-7), and function `has_ntVP` (Axioms A-22 and A-23). Regarding the composition operator, the case $\neg \text{has_ntVP}(n, A \parallel B) \Rightarrow (\text{selL}(n, A \parallel B) = A \parallel B)$ follows when applying the definition (Axioms A-24 and A-11) and the respective induction hypotheses. For the case of the variation operation, we apply the definition for the respective cases (Axioms A-25 and A-13). Since we only have to consider the case ($n \neq i$), the definition of `selL` (Axiom A-13) reduces to the second case. The resulting proposition

$$\left(\begin{array}{l} ((n \neq_{\text{Nat}} i) \vee (\text{NF}(A) = \text{NF}(B))) \\ \wedge \neg \text{has_ntVP}(n, A) \wedge \neg \text{has_ntVP}(n, B) \end{array} \right) \Rightarrow \text{selL}(n, A) \oplus_i \text{selL}(n, B)$$

follows from the respective inductive hypotheses. \square

Basically, there are two ways to add a new variation point \oplus_i to an existing product family P of sort `SPF` α . Firstly, the variation point introduces an alternative to the entire existing product family P , resulting for example in a new product family of the kind $P' \oplus_i P$, where the alternative variant P' has to be provided as well. In

2.2. Axiomatization of Software Product Family Concepts

such a case the variation point identifier n must be fresh in the set of identifiers of P , i.e. $\text{has_ntVP}(n, P) = \text{false}$, and the set of variation point identifiers of P and P' must be disjoint. A second way of introducing a new variation point ($S \oplus_i T$) is by adding it to an existing product family $P \in \text{SPF } \alpha$ by composition, e.g. by composing it directly to P :

$$P \parallel (S \oplus_i T)$$

This can be done only if we require for the composition ($P \parallel Q$) of two arbitrary elements P, Q of sort $\text{SPF } \alpha$, that they do not contain variation points with identical identifiers.

Identifiers are not necessarily unique within the term representing a product family. The reason is that the distributive law (Axiom A-3)

$$(P \parallel Q) \oplus_i (P \parallel R) = P \parallel (Q \oplus_i R)$$

allows to “duplicate” variation points. When applying it from right to left, if the common part P either is a variation point itself or contains variation points, these variation points appear twice in the resulting term of the product family, since the element P gets “duplicated” by the law. Note, this is just a consequence of applying an axiom on an existing product family, and does not affect the consistency of our axiomatization as we have shown in Section 2.2.3.3 in the context of the selection operators. In particular, it is essential to realize that the situation of producing duplicated variation point identifiers by applying the distributive law should not be mistaken with the (forbidden) situation of labeling a variation point with an (already) existing identifier when adding it to a product family.

Modification of Core Assets

The entire philosophy of a software product family is based on the idea to assemble all products from the same set of core assets. Thus, the core assets (resp. the elements of sort α which are wrapped by the core assets) represent those artifacts which actually have to be implemented. Once all assets exist, any product can be assembled following the “blue print” which is given by the corresponding configured product family. Thus, in a realistic scenario it is interesting to know what assets are actually used in the products of a software product family, in order to reason about a product family based on the information of its assets. The set of all core assets which are used for the products of a specific software product family is characterized by the function

2. Formalization of Characteristic Software Product Family Concepts

$$\text{Assets} : \text{SPF } \alpha \rightarrow \text{Set SPF } \alpha \quad (2.50)$$

Given a concrete element of sort $\text{SPF } \alpha$, any core asset which appears in at least one derivable product of the product family PF is added to the set $\text{Assets}(PF)$ of all core assets. This idea is precisely expressed by the following axioms:

$$\text{Assets}(\text{ntrl}) = \emptyset \quad (\text{A-26})$$

$$\text{Assets}(\text{asset}(a)) = \{\text{asset}(a)\} \quad (\text{A-27})$$

$$\text{Assets}(P \parallel Q) = \text{Assets}(P) \cup \text{Assets}(Q) \quad (\text{A-28})$$

$$\text{Assets}(P \oplus_m Q) = \text{Assets}(P) \cup \text{Assets}(Q) \quad (\text{A-29})$$

Since the assets are the atomic units for the composition operator, the set $\text{Assets}(PF)$ is necessarily complete in the sense that it contains *exactly* the required core assets. More precisely, it does neither contain more nor fewer assets, since (i) except for the assets (and the special element ntrl) no other parts are contained in PF on the one hand, while on the other hand, (ii) every asset which appears in the representation of a product family is actually used in at least one product.

We have already seen that the fact that all products of the same product family are constructed from a common set of core assets is an essential prerequisite in order to achieve an efficient kind of reuse. In particular, with respect to modifications of core assets this means that whenever one core asset is modified, the change is propagated to all products of the software product family. Thus, the set of core assets can only evolve for all products simultaneously and not for individual products differently. From this point of view, this is one of the major differences between software product line engineering and single systems engineering with unplanned reuse. In [New05], Paul Clements has summarized this difference nicely:

...single-system development with reuse usually begins by taking the latest version of whatever other systems happen to exist, copying the code, and starting to make product-specific changes to it. The result is that each version of each product launches on its own maintenance and change trajectory, which soon overwhelms the organization trying to keep all of the versions under control and staffing the effort. In a software product line, the core assets follow a single evolution path, and all versions of all products are built from those.

2.2. Axiomatization of Software Product Family Concepts

Due to *software product line evolution* [Bos00] it is very often the situation that assets evolve over time, i.e. after a software product family has been constructed some of its core assets may be modified. This means that all products which contain this asset have to be modified, too. In our setting this can simply be done by modifying the assets directly in the representation of the product family itself. We represent the act of modifying an asset which is part of the universe of assets $\text{Assets}(PF)$ of an existing product family PF by the function

$$\text{modify} : \text{SPF } \alpha, \text{SPF } \alpha, \text{SPF } \alpha \rightarrow \text{SPF } \alpha \quad (2.51)$$

Applying the function $\text{modify}(PF, A, A')$ yields a modified product family which is identical to the product family PF , except that all assets A are replaced by a variation point $A \oplus_n A'$ which has the original asset A and the modified asset A' as its variants. Thereby, the identifier of the new variation point has to be fresh in the set of existing identifiers, i.e. $\text{has_ntVP}(n, PF) = \text{false}$. More precisely, this is specified by the following axioms

$$\text{modify}(\text{ntrl}, A, A') = \text{ntrl} \quad (\text{A-30})$$

$$\text{modify}(\text{asset}(a), A, A') = \begin{cases} \text{asset}(a) \oplus_i A' & , \text{asset}(a) = \text{NF}(A) \\ \text{asset}(a) & , \text{else} \end{cases} \quad (\text{A-31})$$

$$\text{modify}(P \parallel Q, A, A') = \text{modify}(P, A, A') \parallel \text{modify}(Q, A, A') \quad (\text{A-32})$$

$$\text{modify}(P \oplus_i Q, A, A') = \text{modify}(P, A, A') \oplus_i \text{modify}(Q, A, A') \quad (\text{A-33})$$

Note that it is essential to apply a modification of core assets directly to the product family and not only in some of the products after they have been derived from the product family.

2.2.3.10. A General Variants Operator for n Variants

The variants operator which we have introduced so far (cf. Section 2.2.3.3) offers only a choice between two variants. With an appropriate nesting of binary variation points and a corresponding configuration we can simulate a general n -ary choice between n variants. For example, a choice between three variants can be simulated by nesting two binary variation points as in

$$P \oplus_i (Q \oplus_j R) \quad , i \neq j$$

2. Formalization of Characteristic Software Product Family Concepts

For such a nesting we can obtain the element P with the configuration $\{(i, L)\}$, while the elements Q and R are the result of the configurations $\{(i, R), (j, L)\}$ and $\{(i, R), (j, R)\}$, respectively. Nevertheless, nesting binary variation points is just a work-around for the application in a realistic scenario. In particular, without requiring an associative law for the variants operator, the way in which we nest the variation points in order to simulate a general variants operator is crucial, since different ways of nesting represent different product families.

Thus—especially for a realistic scenario—a general variants operator offering multiple variants is a much more attractive and conceptually clean solution. We introduce such a general variants operator by means of the function

$$\oplus : \text{Seq SPF } \alpha, \text{Nat} \rightarrow \text{SPF } \alpha \quad (2.52)$$

It represents a variation point which offers the choice between multiple (finitely many) variants. In contrast to the variants operator introduced so far, we call such an n -ary variation point a *general* variation point, and an n -ary variants operator a *general* variants operator, accordingly. For a better differentiation we speak of a *binary*² variants operator when referring to the regular variants operator as introduced in Section 2.2.2.3. We “overload” the symbol \oplus and represent both kinds of variants operators by \oplus . However, since both versions of the variants operator can easily be distinguished by the number and the types of its arguments, it is always clear which variants operator is meant. Similarly to the binary variants operator, we also use an infix notation for the general variants operator and usually write $\oplus_i(\text{Vars})$ to denote the variation point with the name i offering the variants Vars .

The general variants operator represents a natural generalization of the binary variants operator. The general variants operator is a more comfortable version, but does not give more expressiveness. Thus, we decided to include only the binary version of the variants operator in the algebraic specification of sort $\text{SPF } \alpha$ and restrict ourselves with the presentation of the general variants operator to this section.

The general variants operator is also of sort $\text{SPF } \alpha$, which means that it can be included like the binary variants operator as a regular argument for the composition operator. Like the binary variants operator, also every general variants operator is identified by a (unique) name, which is represented by the second argument of sort Nat . The first argument of sort $\text{Seq SPF } \alpha$ is a sequence of elements which represent the variants offered by this variation point. The corresponding algebraic

²Although the variants operator (cf. Equation 2.4) is not a binary function—it takes three arguments: an identifier and two variants—we call it *binary* emphasizing the fact that it offers a choice between *two* variants.

2.2. Axiomatization of Software Product Family Concepts

specification of the sort $\text{Seq } \alpha$ —representing arbitrary sequences of elements of the sort α —can be found in the Appendix A.4 on Page 234. For the sake of simplicity we identify the variants according to their position in the sequence, similarly to the indexed access of elements in an array.

Usually, associativity and commutativity do not really apply for a non-binary operator as our general variants operator. However, the general variants operator has a commutative character since the order in which the variants are specified is in principle not relevant for the result of the configuration. Since we identify the variants according to the position in the sequence, this is not true for our general variants operator, however, when identifying the variants in a different way—not according to their position in the sequence—the general variants operator is indeed commutative. Regarding the associativity, similarly to the binary variants operator, the general variants operator has a non-associative character. Thus, nesting the general variants operator induces a hierarchy in which every variant is precisely associated to one specific general variation point.

Since the general variants operator uses the data structure $\text{Seq } \alpha$ to manage its variants, it requires a special (configuration) selection operation. The selection operation for the general variants operator is represented by the function

$$\text{select} : \text{SPF } \alpha, \text{Nat}, \text{Nat} \rightarrow \text{SPF } \alpha \quad (2.53)$$

Its first argument represents the product family which has to be configured, the second argument represents the identifier of the affected variation point, and the third argument is the identifier for the chosen variant. More precisely, the function $\text{select}(PF, i, m)$ returns the configured version of the product family PF where the variation point i has been replaced by its variant identified by the number m . Since we identify the variants according to the position where they appear in the sequence, the identifiers for the variants are of sort Nat . The variant identifiers are of sort Nat for the sake of simplicity, only. In general, this is not compulsory.

The behavior of the configuration process using the function select is characterized by the following axioms.

$$\text{select}(\text{ntrl}, i, j) = \text{ntrl} \quad (2.54)$$

$$\text{select}(\text{asset}(a), i, j) = \text{asset}(a) \quad (2.55)$$

$$\text{select}(P \parallel Q, i, j) = \text{select}(P, i, j) \parallel \text{select}(Q, i, j) \quad (2.56)$$

$$\text{select}(\oplus_i(\text{Vars}), j, m) = \begin{cases} \text{select}(\text{get}(m, \text{Vars}), j, m) & , i = j \\ \oplus_i(\text{map_select}(\text{Vars}, j, m)) & , \text{else} \end{cases} \quad (2.57)$$

2. Formalization of Characteristic Software Product Family Concepts

Similarly to the selection operators `selR` and `selL` for the binary variants operator, if the product family consists of a single asset or the neutral element, no configuration selection has any effect. For a composed element, selecting a certain variant means to perform the selection in both parts of the compound element. Performing a selection on a variation point either means to configure the variation point to the selected variant and continue the selection recursively in this variant, if the selection operation is intended for this particular variation point, or to push the configuration through all its variants. For the latter situation, in order to apply a configuration to all variants of a general variation point, we use the following auxiliary function

$$\text{map_select} : \text{Seq SPF } \alpha, \text{Nat}, \text{Nat} \rightarrow \text{Seq SPF } \alpha \quad (2.58)$$

The function `map_select` processes a sequence of elements of sort `SPF` α and applies the respective configuration to each element. Finally, this results in a sequence of configured elements, as the following axioms describe

$$\begin{aligned} \text{map_select}(Vars, j, m) &= \text{select}(\text{head}(Vars), j, m) & (2.59) \\ &\quad \circ \text{map_select}(\text{rest}(Vars), j, m) \end{aligned}$$

$$\text{map_select}(\langle \rangle, j, m) = \langle \rangle \quad (2.60)$$

Note that the functions `head`, `rest` and `get` return the first element, the remainder of a sequence without the first element, and the n^{th} element of a given sequence. The exact behavior of these functions is specified in the algebraic specification of sort `Seq` α which can be found in Appendix A.4 (Page 234). Further note that instead of specifying an auxiliary function like `map_select` we could also have introduced the operation `select` as an higher-order function.

2.2.4. Complete Algebraic Specification of the Sort `SPF` α

In the preceding Sections 2.2.2 and 2.2.3 we have collected and described the essential properties and concepts which constitute a software product family. These properties and concepts are expressed using a set of sorts, many-sorted functions operating on these sorts, and a set of axioms which are stated as equalities between these functions. Together, this forms an axiomatization which describes a software product family in a general, implementation independent, though formal way.

2.2. Axiomatization of Software Product Family Concepts

Figure 2.11 shows the entire axiomatization, denoted as an *algebraic specification* [Wir90] defining the sort $\text{SPF } \alpha$ of software product families. By means of this algebraic specification we can now precisely define what a software product family is. In particular, we can now finally render the characterizing, rather general definition of a software product family (Definition 2.3), which we have provided in the beginning of this Chapter, more precisely.

Definition 2.10 (Software Product Family). *We call any computation structure for the algebraic specification of sort $\text{SPF } \alpha$ (cf. Figure 2.11) a software product family.*

More illustratively, Definition 2.10 states that every concrete computation structure which (i) provides a concrete sort for every sort defined in the algebraic specification of sort $\text{SPF } \alpha$, and (ii) which provides concrete realizations of the specified functions in a way that all axioms are respected represents a realization of a software product family according to our definition. From this point of view, a computation structure is any realization of the axiomatization using an arbitrary mechanism of the real world, which we happen to know and understand, already. With the computation structure of stickmen drawings for sort $\text{SPF } \text{Stickman}$, we have provided a first toy-example of such a concrete computation structure, although this was just to illustrate the concepts and axioms. In the following chapter we provide a specification framework called PF-CCS which allows to model a very specific kind of product families (in a process algebra context) that are realizations of the algebraic specification for sort $\text{SPF } \alpha$, too.

Note that in the axiomatization, the equality between elements of sort $\text{SPF } \alpha$ is based on the normal form which we have introduced in Section 2.2.3.2. In the side conditions of some axioms we have to check whether elements of sort $\text{SPF } \alpha$ are equal. Such checks are based on the normal form of the elements, where two elements of sort $\text{SPF } \alpha$ are considered equal, if their corresponding normal forms are equal (with respect to term equivalence). Due to the number of additional axioms specifying the normal form, we have not included these axioms in the following algebraic specification. The specification of the function NF can be found in Section 2.2.3.2, starting on Page 50.

2. Formalization of Characteristic Software Product Family Concepts

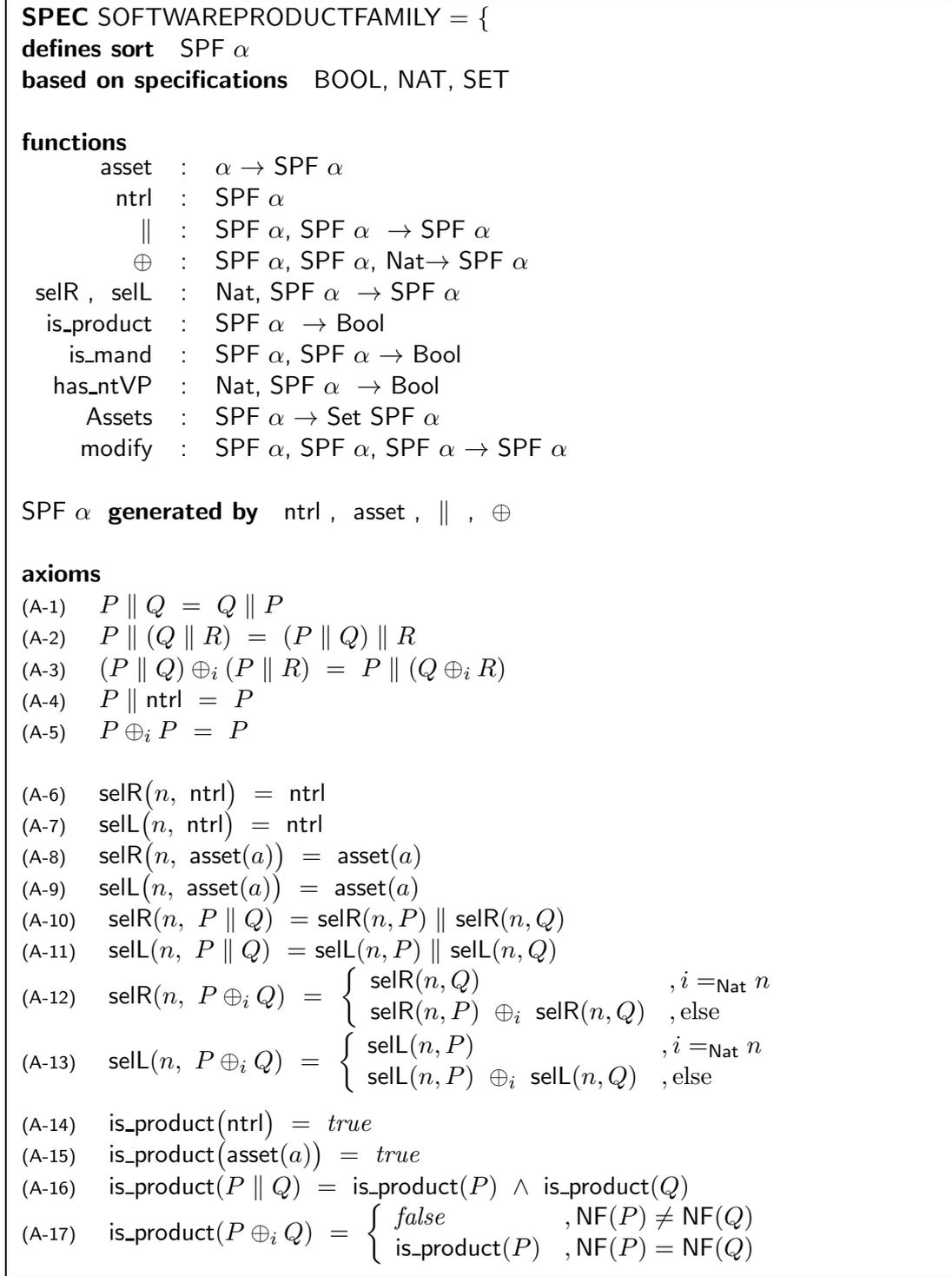


Figure 2.11.: Algebraic specification of sort $\text{SPF } \alpha$ representing a software product family. To be continued on the next page. Note that the function NF realizes a unique normal form for elements of sort $\text{SPF } \alpha$. Its specification can be found in Section 2.2.3.2 on Page 50.

(A-18)	$\text{is_mand}(\text{ntrl}, X) = \text{false}$	
(A-19)	$\text{is_mand}(\text{asset}(a), X) = (\text{NF}(X) = \text{asset}(a))$	
(A-20)	$\text{is_mand}(P \parallel Q, X) = \begin{cases} \text{false} & , \text{NF}(P \parallel Q) = \text{ntrl} \\ (\text{NF}(P \parallel Q) = \text{NF}(X)) \\ \vee \text{is_mand}(P, X) & , \text{else} \\ \vee \text{is_mand}(Q, X) \end{cases}$	
(A-21)	$\text{is_mand}(P \oplus_i Q, X) = \begin{cases} \text{false} & , \text{NF}(P \oplus_i Q) = \text{ntrl} \\ (\text{NF}(P \oplus_i Q) = \text{NF}(X)) \\ \vee \left(\begin{array}{l} \text{is_mand}(P, X) \\ \wedge \text{is_mand}(Q, X) \end{array} \right) & , \text{else} \end{cases}$	
(A-22)	$\text{has_ntVP}(n, \text{ntrl}) = \text{false}$	
(A-23)	$\text{has_ntVP}(n, \text{asset}(a)) = \text{false}$	
(A-24)	$\text{has_ntVP}(n, P \parallel Q) = \text{has_ntVP}(n, P) \vee \text{has_ntVP}(n, Q)$	
(A-25)	$\text{has_ntVP}(n, P \oplus_i Q) = \left(\begin{array}{l} ((n =_{\text{Nat}} i) \wedge (\text{NF}(P) \neq \text{NF}(Q))) \\ \vee \text{has_ntVP}(n, P) \\ \vee \text{has_ntVP}(n, Q) \end{array} \right)$	
(A-26)	$\text{Assets}(\text{ntrl}) = \emptyset$	
(A-27)	$\text{Assets}(\text{asset}(a)) = \{\text{asset}(a)\}$	
(A-28)	$\text{Assets}(P \parallel Q) = \text{Assets}(P) \cup \text{Assets}(Q)$	
(A-29)	$\text{Assets}(P \oplus_i Q) = \text{Assets}(P) \cup \text{Assets}(Q)$	
(A-30)	$\text{modify}(\text{ntrl}, A, A') = \text{ntrl}$	
(A-31)	$\text{modify}(\text{asset}(a), A, A') = \begin{cases} \text{asset}(a) \oplus_i A' & , \text{asset}(a) = \text{NF}(A) \\ \text{asset}(a) & , \text{else} \end{cases}$	
(A-32)	$\text{modify}(P \parallel Q, A, A') = \text{modify}(P, A, A') \parallel \text{modify}(Q, A, A')$	
(A-33)	$\text{modify}(P \oplus_i Q, A, A') = \text{modify}(P, A, A') \oplus_i \text{modify}(Q, A, A')$	

Figure 2.11. CONTINUED: Algebraic specification of a software product family. Note that the function NF realizes a unique normal form for elements of sort $\text{SPF } \alpha$. Its specification can be found in Section 2.2.3.2 on Page 50.

theorems	
(T-1)	$(Q \parallel P) \oplus_i (R \parallel P) = (Q \oplus_i R) \parallel P$
(T-2)	$\text{ntrl} \parallel P = P$
(T-3)	$P \oplus_i (P \parallel R) = P \parallel (\text{ntrl} \oplus_i R)$
(T-4)	$(P \parallel R) \oplus_i P = P \parallel (R \oplus_i \text{ntrl})$
(T-5)	$\text{selL}(m, \text{selL}(n, P)) = \text{selL}(n, \text{selL}(m, P))$
(T-6)	$\text{selR}(m, \text{selR}(n, P)) = \text{selR}(n, \text{selR}(m, P))$
(T-7)	$(\text{selL}(m, \text{selR}(n, P)) = \text{selR}(n, \text{selL}(m, P))) \Leftarrow \neg(m =_{\text{Nat}} n)$
(T-8)	$\neg \text{has_ntVP}(n, P) \Rightarrow (\text{selL}(n, P) = P)$
(T-9)	$\neg \text{has_ntVP}(n, P) \Rightarrow (\text{selR}(n, P) = P)$
	}

Figure 2.11. CONTINUED: Algebraic specification of a software product family.

2.3. Modeling Dependencies in Software Product Families

So far we have seen that a software product family is an essential part of a software product line, since the product family corresponds to the blueprint of how the assets have to be combined in order to construct the respective products. However, in order to establish a software product line some other purposes and models have to be taken into account. One of these purposes is to restrict the set of “constructible” products of a software product family, and to characterize a subset of products which actually are intended to exist as real systems. According to literature, so-called *feature models* [KHNP90] are very often used for this purpose. However, as feature models are also used for other purposes (cf. the discussion in Section 2.5), we will not take over this name and call our model (that restricts the set of possible configurations) a *dependency model*. The model is represented as a formula in propositional logic. We will motivate and introduce it in the following.

In general, a product family has a large number of configurations. In Section 2.2.3.4 we have seen that each complete configuration describes a single product. However, not all combinatorially possible configurations of a software product family are desired and shall be “allowed”. Speaking in terms of constructible products, not all products shall actually become constructed and exist in the field. For the purpose of controlling the set of possible configurations we introduce a *dependency model*, which has a similar intention as a so-called feature model [KHNP90]. In combination with a dependency model a software product family becomes applicable for realistic scenarios. While the model of the software product family represents the construction blueprint for each product—i.e. it specifies *how* every product is constructed from the set of core assets—a dependency model specifies *what* products are allowed to be constructed.

2.3. Modeling Dependencies in Software Product Families

Reasons for restricting the set of configurations are usually non-functional, e.g. implementation-specific reasons, platform restrictions, market decisions, but also country-specific requirements and statutory regulations. For example, due to marketing reasons selecting the keyless-entry option in a software product family for automobiles might require that also the comfort version of the seats has to be selected, although—from a functional point of view—the keyless-entry functionality would work with the standard version of the seats, too. An example for a statutory restriction is the *eCall* (emergency call) [Com09] system. It is compulsory for new vehicles in most European countries starting from 2010, while for configurations of cars for the US or Asian market such a system is not compulsory, yet.

In general, in the context of a software product family a *dependency* is a relation between *variants*. Some prominent examples of dependencies are the *requires* or *excludes* relations [Bat05, KHNP90], which express that the existence of some parts requires the existence or absence of some other parts. In general, a dependency expresses that the selection of a certain variant of one variation point determines also the configuration of other variation points.

The dependency model for a specific product family PF is given as a formula DEP_{PF} in propositional logic. More precisely, DEP_{PF} is a formula over atomic propositions which represent the existence or absence of single variants. In order to model individual variants in this propositional logic setting we define for every variation point $\oplus_i(P, Q)$ of a product family PF two atomic propositions L_i and R_i . The proposition L_i means that the left variant Q , the proposition R_i means the right variant P is selected to be present in the resulting system, respectively. For a product family PF with n non-trivial variation points we define

$$AP_{PF} = \{R_i \mid i \in \{1, \dots, n\}\} \cup \{L_i \mid i \in \{1, \dots, n\}\}$$

to be the set of all such atomic propositions. Usually we omit the subscript and write DEP and AP , if it is clear which product family we mean.

Definition 2.11 (Dependency Model). *Let PF be a product family with n variation points. We call any propositional formula DEP_{PF} with propositions only from the set AP_{PF} a dependency model for PF .*

In general, DEP can be an arbitrary propositional formula, but usually it will consist of an arbitrary conjunction or disjunction of sub-formulas representing the dependencies of individual variants. An example for a very simple dependency model is the following DEP -formula:

$$R_2 \Rightarrow L_{42}$$

It states that whenever the right variant of variation point 2 has been selected, then the left variant of variation point 42 has to be selected, too. In a realistic context this might for example mean that selecting the keyless-entry option (represented by

2. Formalization of Characteristic Software Product Family Concepts

R_2) always requires the comfort behavior of the seats (represented by the variant L_{42}). Combining several dependencies of this kind as a conjunction yields a slightly larger *DEP*-formula:

$$(R_4 \Rightarrow R_{17}) \wedge (R_2 \Rightarrow L_{42}) \wedge (L_2 \Rightarrow \neg L_8)$$

However, we are not restricted to such simple formulae. By allowing (full) propositional logic with the described atomic propositions in order to model the dependencies, we can express all kind of relevant dependencies, e.g. also more complex ones such as:

$$(R_4 \vee L_{42}) \wedge ((R_4 \wedge L_2) \Rightarrow (R_{17} \vee R_{18}))$$

This dependency states that a valid configuration has to select at least one of the variants R_4 or L_{42} in any case. In addition, if both variants R_4 and L_2 are selected simultaneously, the variants R_{17} or R_{18} or both have to be selected, too.

Normal Form for Dependencies

Since with a binary variants operator either the left or the right variant can be selected for each variation point we observe a natural duality for the variants of any variation point i :

$$L_i \equiv \neg R_i \tag{2.61}$$

$$R_i \equiv \neg L_i \tag{2.62}$$

These dualities allow to eliminate every negation in *DEP*. Together with the following tautologies 2.63 to 2.67

$$A \Rightarrow B \equiv (\neg A \vee B) \tag{2.63}$$

$$A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C) \tag{2.64}$$

$$A \Rightarrow (B \vee C) \equiv \neg A \vee B \vee C \tag{2.65}$$

$$(A \wedge B) \Rightarrow C \equiv A \Rightarrow (\neg B \vee C) \tag{2.66}$$

$$(A \vee B) \Rightarrow C \equiv (A \Rightarrow B) \wedge (A \Rightarrow C) \tag{2.67}$$

we can transform every *DEP* formula into a *normal form*.

Definition 2.12 (Normal Form of a Dependency Model). *Let $A_i, B_{i,j} \in AP$ be atomic propositions. We say that the dependency model DEP is in normal form if it has the following representation*

$$DEP \equiv \bigwedge_{i=1}^m \left(A_i \Rightarrow \bigvee_{j=1}^{n_i} B_{i,j} \right)$$

2.3. Modeling Dependencies in Software Product Families

The normal form reflects a natural understanding of a dependency: The propositions A_i represent all those variants whose existence or non-existence influences other variants. The way how the existence of other variants is effected is described by the respective disjunctions $\bigvee_{j=1}^{n_i} B_{i,j}$.

A normal form offers several benefits. The transformation into the normal form directly supports the dependency methodology, since it allows to specify dependencies in any initially free way while being able to transform them in a normal form, afterwards. Moreover, minimizing the normal form using standard minimization techniques for propositional logic, e.g. [Kar53], helps to determine the set of all variants ranging over AP which are involved into a dependency relation at all.

Connection between a Dependency Model and a Software Product Family

For the purpose of establishing the connection between a dependency model for a product family of sort \mathbf{SPF} α and its possible configurations, we introduce a representation of a single configuration as a set of propositions. For every complete configuration $c \in \mathit{CONFS}^n$ of a product family PF with n non-trivial variation points we define a function which yields an equivalent representation as a formula in propositional logic.

$$\kappa : \mathit{CONFS}^n \rightarrow \mathcal{P}(AP_{PF}) \quad (2.68)$$

The function κ represents the selected variants as a set of corresponding atomic propositions, i.e.

$$\kappa(\{(1, v_1), \dots, (n, v_n)\}) = \left\{ Prop_i \mid i \in \{1 \dots n\}, Prop_i = \begin{cases} L_i & , v_i = L \\ R_i & , v_i = R \end{cases} \right\}$$

For example, the configuration $c = \{(1, L), (2, L), (3, R)\}$ is represented by the set of propositions $\kappa(c) = \{L_1, L_2, R_3\}$.

By means of the representation of a configuration as a set of propositions we can introduce the notion of a *valid* configuration. Illustratively, a configuration of a product family is valid for a given dependency model if the variants selected by the configuration fulfill the restrictions imposed by the dependency model. The following definition makes this idea precise.

2. Formalization of Characteristic Software Product Family Concepts

Definition 2.13 (Valid Configuration). *A given configuration $c \in \text{CONFS}^n$ of a product family PF is said to be valid according to a given dependency model DEP_{PF} if the corresponding logical representation of the configuration is a model (in the logical sense) of the dependency model, i.e. iff*

$$\kappa(c) \models DEP_{PF}$$

Every product which is represented by a valid configuration is called valid.

We illustrate these concepts with our running example of the family *Spf* of Stickman shown in Table 2.3 on Page 41. Imagine that all stickman drawings with a coffee cup in their hand are happy and thus have a smiling facial expression. In addition, every female stickman drawing is happy and shall be always smiling. We can restrict the set of all combinatorially possible configurations (in this case stickman) for *Spf* by requiring the corresponding dependencies:

$$\begin{aligned} DEP_{Spf} &\equiv (L_3 \vee R_2) \Rightarrow L_1 \\ &\stackrel{2.61-2.67}{\equiv} (R_3 \wedge L_2) \vee L_1 \end{aligned}$$

It states that whenever we have selected the left variant (representing the asset Coffee) of the variation point 3, or the right variant of variation point 2 (representing the asset Female_Torso), we also have to select the left variant (representing the asset Smiling) of variation point 1 in order to have a valid product. For this particular dependency model, the three products shown in Figure 2.4a (cf. Page 42) which are represented by the configurations $c_1 = \{L_1, L_2, L_3\}$, $c_2 = \{R_1, L_2, R_3\}$ and $c_3 = \{L_1, R_2, L_3\}$ are valid since every configuration is a model of DEP_{Spf} . In contrast, the configuration $c_4 = \{R_1, L_2, L_3\}$ is not valid, since $R_1, L_2, L_3 \not\models (L_3 \vee R_2) \Rightarrow L_1$. This can be seen easily if we consider the second line of the equation which shows DEP_{Spf} in an equivalent representation.

Expressing dependencies as formulae in propositional logic has several advantages:

- Propositional logic offers an appropriate degree of expressiveness. In particular, it is sufficient to embody those kinds of dependency relations which are typical for product families [Man02, Bat05], i.e. relations which deal with the existence of variants such as *requires* or *excludes* relations.
- A logical formula is an appropriate representation for the application of formal methods, e.g. it can easily be fed into a model checker or theorem prover. This is especially important for the tool-supported verification of a software product family using e.g. model checking techniques, as we discuss them in Chapter 4.

In general, if no dependency model is explicitly given, we assume that no restrictions on the set of configurations are made, i.e. that all combinatorially possible configurations are actually allowed to be constructed. Technically, in such a situation we assume the dependency model to be equivalent to the logical value *true*, i.e.

$$DEP \equiv true$$

2.4. Discussion

In this chapter we have provided a formalization of software product family concepts that allows to reason about software product families in general, and that underpins the entire paradigm of software product line engineering. In the following we motivate our reasons to formalize software product family concepts by means of an axiomatization (using the technique of algebraic specification). In addition, we briefly discuss the structural relation between elements of sort *SPF* α and *AND/OR-trees*, and motivate why we have not combined the axiomatization with a type system for elements of sort *SPF* α .

2.4.1. On the Choice of an Algebraic Specification

In general, there are several ways of defining a software product family and its specific concepts in a mathematical, formal manner. Historically, two prominent research directions which made use of rigorous mathematical specification techniques are (i) approaches defining the semantics of programming languages (cf. Floyd and Hoare [Flo67, Hoa69, Dij76], and Scott and Strachey [SS71, Sto77]), and (ii) approaches for the definition of (abstract) data types (cf. [Gut75, LZ74, Zil74, BHK89, EM85, Wir90]).

For our purpose we have chosen to axiomatize software product family concepts using an *algebraic specification* [Wir90]. Our reasons for an axiomatization by means of an algebraic specification are:

- *An algebraic specification facilitates an implementation-platform independent, property-oriented formalization.* The software product line success stories (cf. Section 1.1) show that product family concepts are applied (i) in different domains and for different application areas, (ii) at many stages throughout the software development process, (iii) and for the modeling of different kinds of products. However, regardless of the concrete software product family, the basic concepts which are characteristic for a software product family are always

2. Formalization of Characteristic Software Product Family Concepts

the same. To this extent, the engineering of software-intensive systems as a software product family embodies a universal design concept which is orthogonal to other design concepts for the construction of systems, and which can be studied in its own right. For our purpose, we wanted to specify the characteristic software product family concepts in a self-contained, implementation-platform independent way which still allows to apply formal methods in order to reason about conceptual properties of software product families. The technique of algebraic specification is suitable for such a purpose. An algebraic specification allows to describe an (abstract) computation structure without prescribing *how* the required data structure is represented or *how* the respective functions are actually implemented. In this sense, an algebraic specification can be used to represent an entire class of concrete computation structures, and captures the conceptual essence which is common to the concrete implementation-specific software product family realizations of this class. With respect to the variety of different software product families, the algebraic specification of sort $\text{SPF } \alpha$ which we have given in this chapter characterizes a class of possible realizations [ST97]. The members of this class are all those concrete computation structures which exist as real software product families, and which exhibit the characteristic properties postulated by the axioms of the algebraic specification.

- *An axiomatization allows to reason about software product family concepts.* We use the axiomatization in order to reason about conceptual properties of software product families in general. A main motivation was not only to define the universal design concept of a software product family, but in particular to allow reasoning about fundamental properties which every product family exhibits. Only in such a way we can gain a better understanding of software product families, and verify our ideas and theorems in a formally valid way. Since a software product family deals with the constructional aspect of how to build systems based on their commonalities, an axiomatization was our vehicle of choice, as it allows to derive new properties from axioms, which characterize the interplay between the operations that are used to build software product families. In addition, an axiomatization that is given by means of an algebraic specification comes with standard ways of deriving new properties and proving the corresponding theorems, e.g. by *structural induction* or *logical transformation*.
- We use the axiomatization (i) to determine for existing approaches whether they are proper software product families according to our axiomatization, and (ii) to guide the construction of new software product family realizations by characterizing the fundamental operations. Here, an algebraic specification offers a very suitable vehicle as it describes software product family concepts from a constructive point of view: although an algebraic specification abstracts from implementation-specific details, it can still be constructive [Loe87] in the

sense that it guides the development of a concrete computation structure of a software product family for a specific purpose. In particular, if the algebraic structure itself fulfills certain *constructiveness constraints* [vHL89], it can basically directly be implemented by deriving term rewriting rules from the axioms. For example, due to defining all axioms according to the scheme of *primitive recursion*, the algebraic specification of sort $\text{SPF } \alpha$ can very easily be translated into an implementation in a functional programming language such as for example ML [MTH90]. In this way, an algebraic specification is more useful to guide and reason about the construction of a concrete software product family than for example a denotational approach.

- *Standardization of and Reasoning about Software Product Family Concepts*
The axiomatization allows to define concepts such as a *normal form*, *trivial variation points*, *optional parts*, *mandatory parts*, *common parts*, etc. in a formal way. Some of these notions and terms are frequently used in the software product line community, but lack a formal definition. For example, an apparently simple concept like a *variation point* has quite different meanings in different approaches [WG04], driven by different implementation paradigms, different perspectives on technological platforms, different levels of abstraction, or different stages in the development process. Here, an axiomatization describes existing concepts precisely, and provides a basis to define a precise terminology, too. For new concepts, the axiomatization provides the formal framework to define them in relation to the existing body of knowledge, and to reason about them formally. For example, the notion of an explicit *normal form* for software product families, or the concept of a *trivial variation point*, are defined in a precise formal setting within this thesis for the first time, as far as we are aware.

2.4.2. Structural Similarity to an AND/OR-Tree

With respect to the structure of a product family of sort $\text{SPF } \alpha$ which can be specified by means of the constructors, we observe that such a structure is basically equivalent to an *AND/OR-tree* [LS93]. The composition operation corresponds to *AND*-nodes, while the variants operator corresponds to *OR*-nodes. While an AND/OR-tree is a structure that is commonly known, we have not simply used this structure for the specification of software product families, since

- there is no set of commonly accepted laws for AND/OR-trees which allow a restructuring of such a tree, similarly to the way in which we use the axioms for the constructors to restructure the term representation of an element of sort $\text{SPF } \alpha$, and

2. Formalization of Characteristic Software Product Family Concepts

- since we are in particular interested in the operations such as selection, determination of mandatory parts, etc., which are not predefined in the context of an AND/OR-tree, either.

In this light, it was not very beneficial for us to base our specification of a software product family on the structure of an AND/OR-tree. In fact, it was more comfortable to specify such a structure anew by introducing the constructor functions with their corresponding postulates using the typical mechanism of algebraic specification, and thereby being able to benefit from the existing concepts, e.g. the notion of an abstract computation structure, which are already established in the context of algebraic specifications.

2.4.3. Combining the Axiomatization with a Type System

We have presented the axiomatization in this chapter without explicitly combining it with a *type system*, as it is for example commonly known [LW94] from the data types of object-oriented programming languages. This means that we cannot further classify the elements of sort $\text{SPF } \alpha$ into subtypes.

Having no type system immediately implies that questions which are typically addressed in the context of types and their subtypes are not directly expressible within the axiomatization. For example, as we have already seen in Section 2.2.3.8 (Page 80), while the function `is_mand` allows to specify questions of the kind “*Has every stickman drawing which is derivable from the product family Z of stickman drawings either a smiling or a sad face?*”, formally represented by

$$\text{is_mand}(Z, \text{asset}(\text{Smiling}) \oplus_1 \text{asset}(\text{Sad}))$$

the question “*Has every stickman drawing of the family Z actually a face?*” is not expressible in terms of the axiomatization. The reason for this is because the specification of a concept like “*having a face*” requires the existence of a type system and the notion of *subtypes*.

A type system represents an orthogonal concept that can be additionally added to the axiomatization if desired, but that is not necessary to define the conceptual construction principle behind a software product family. Therefore, we have not added a type system on top of the axiomatization, being aware that questions such as the one before are not directly expressible within our theory.

2.5. Related Work

Feature Oriented Software Development

Most of the related approaches which deal with the constructional aspect of software product families are found in the area of *Feature Oriented Software Development* (FOSD) [CE00]. FOSD deals with the construction of large-scale, variable software systems by structuring them into the features they provide. An overview of FOSD and the recent development in this area is given in [AK09]. *Features* [KHNP90, CE00] represent functional or non-functional, observable properties or characteristics of system components, assets, and/or products, which are relevant for a certain stakeholder. However, the definitions of what a feature is, are not consistent and vary from approach to approach.

A typical structure to capture variability on the level of features are so-called *feature models* [KHNP90]. They are used to model optional, mandatory and variable features, and their dependencies in a set of related products. A feature model is a combination of our concept of a software product family and a dependency model (cf. Section 2.3): On the one hand, a feature model contains information of mandatory and optional parts, i.e. it represents information of how products are hierarchically structured by modeling which parts are present (as sub-parts) in which product variants, e.g. in a product family of cars features such as *Engine* or *Radio* actually represent the idea of atomic, composable entities (of the same sort). However, since different features can represent incomparable entities, artifacts, or properties (*diesel engine* and *automatic mode* can both be features in the same feature model), this is not done strictly on the level of composable and fitting atomic assets or parts (of the same sort). On the other hand, a feature model contains dependencies, such as *requires* or *excludes* relations between features. Here, the aim is to forbid or to force certain combinations of features, serving the same purpose as our dependency model (cf. Section 2.3). However, a feature models mixes those two fundamentally different aspects (construction and configuration restriction) in a way where different concerns are not separated anymore. Moreover, feature models usually lack a precise semantics (which hinders to reason about features or feature combinations using formal methods).

Feature models are usually denoted by means of feature diagrams, which are essentially AND/OR trees [LS93]. However, feature models can likewise be denoted as formulae. Mannion [Man02] was the first one to use propositional formulae in order to reason about software product lines. In [Bat05], Batory relates the semantics of feature models with propositional logics and grammars. Another popular work is due to Czarnecki [CW07], where he relates feature models to BDDs (Binary Decision Diagrams) [Ake78], and shows how to extract a feature model from a given propositional formula. Approaches which use a logical representation usually deal

2. Formalization of Characteristic Software Product Family Concepts

with restricting the set of configurations, and with related questions, and thus serve the same purpose as our dependency model.

Algebraic Treatment of FOSD

In [ALMK10, ALMK08], Apel et al. introduce an algebraic approach for features, feature composition and feature-based program synthesis, which captures the key ideas of feature orientation in an algebraic setting. In accordance with most feature-oriented approaches, a feature is understood to be an increment in functionality that captures a stakeholder’s requirement. Features are associated to code fragments—where Java code is used to exemplify the concepts. A single feature corresponds to a hierarchical tree structure called *feature structure tree (FST)*. A FST consists of nodes representing code structures of the respective language, e.g. for Java such code structures are for example packages, files, classes, fields or methods. Features can be composed, where feature composition is understood and realized in two different ways: The first kind of feature composition allows to combine two entire features, respectively their FSTs. On the level of code, this kind of feature composition is realized by so-called *superimposition*. Superimposition combines the feature structure trees of two features by matching the nodes with the same relative position, names, and types in the tree hierarchy. The result is a combined feature structure tree—and thus again a feature—representing the composition of both features. Besides superimposition, a second kind of feature composition allows for a more fine grained extension of a feature by adding specific nodes to the FST that represents that feature. It is called *composition by quantification and weaving*. In contrast to superimposition, this second kind of composition does not combine two features, but rather combines a feature (its FST) and a so-called *modification*. In this context a modification is a tuple that consists (i) of a specification which characterizes a subset of nodes of an FST to which the modification shall be applied, and (ii) a specification of how these nodes are affected (add or altered). Applying a modification to a feature yields a modified feature.

Within the feature algebra, the operations for composing and modifying features are described in a way which abstracts from the concrete kind of feature structure trees which are specific for any individual language. In the *feature algebra* superimposition corresponds to an abstract operation called *introduction sum*. It combines two so-called *introductions*, which are abstract counterparts of basic features, i.e. feature tree structures, yielding a new introduction. The introduction sum operation constitutes an idempotent monoid over the set of introductions. Beside introduction sum, two other operations on features are introduced: *modification application* and *modification product*. These operations implement the idea of composition by quantification and weaving, where *modification application* takes a so-called *modification* and a introduction, and returns the modified introduction. *Modification application* distributes over the introduction sum operation. The third operation, *modification product* is a binary operation on modifications that returns a new modification which represents the combined effect of both modifications. From an algebraic perspective,

the modification product operation induces another monoid over the set of modifications. Modification application combines the two monoids induced by introduction sum and modification product, and together with the monoids allows to represent feature composition. In summary, the algebraic structure which results from the two monoids and the modification application operation is very similar to the structure of a vector space, where modification application takes the role of scalar product.

There are several options in which features, i.e. introductions and modifications, can be composed. In order to explore these options more closely the notion of *quarks* is introduced. A quark is a tuple that consists of an introduction and one (or more) modifications. Quarks can be composed to form new quarks. Different kinds of quark-composition—local and global quark composition—are discussed, which represent different ways of how and where modifications are applied while composing a sequence of features.

The focus of the feature algebra as it is introduced in [ALMK10, ALMK08] is on composing features, and describing effects of modifying features and feature compositions. In contrast to our axiomatization of software product family concepts, the feature algebra does not explicitly support the notion of variability, in the sense that optional, variable or mandatory parts as known from feature models can not be explicitly represented as variable features in the feature algebra. Consequently, a notion of configuration or alternative features in the sense of an alternative variation point does not exist, either. However, the feature algebra takes the notion of composition of features much further, than our axiomatization deals with the composition of assets or product families (being the counterparts to features). In particular, both algebras—our axiomatization of software product family concepts and the feature algebra—fit together in the sense that the feature composition mechanism of the feature algebra can be used to realize the composition function of our axiomatization. In that way, the concept of variability and that of feature composition can be combined in a conceptually clean, and formally based way, resulting in a new comprehensive algebra for feature-oriented software product families. In fact, in [AKGL10], Apel et al. go actually a similar way and introduce a feature-oriented language which implements the concepts introduced in their feature algebra, but which is combined with a very general kind of feature model in order to model the design and the restrictions of a product line. We consider their approach in the following.

Feature-oriented Product Lines

Based on the concepts introduced in their feature algebra, in [AKGL10] Apel et al. consider so-called *feature-oriented product lines*, which are families of programs that share a common set of features. Similarly to their concept of features in their feature algebra [ALMK10, ALMK08], a feature adds new program structures, or refines existing ones, when added to a program. In [AKGL10], product lines are specified

2. Formalization of Characteristic Software Product Family Concepts

using FFJ_{PL} , which extends the feature-oriented, Java-like language *Feature Featherweight Java (FFJ)* with product line concepts. FFJ_{PL} addresses the challenge of producing only type-safe family members, such that every derivable program is guaranteed to comprise only a type-safe combination of features.

Already the starting point, the authors' language FFJ , comes with a type system comprising a set of type rules and auxiliary operations, that facilitates to check whether a single FFJ program is well-formed according to the type concept. Compared to type-checks of single programs in FFJ , type-checking in FFJ_{PL} is performed on the code comprising the entire product line, and does no longer require to generate and check every program individually. In this light, the idea behind a FFJ_{PL} product line is the same as the one behind a software product family of sort $SPF \alpha$, since both formalisms represent a product family in an integrated way.

A feature-oriented product line consists of two ingredients: a set of so-called *feature modules* which take the role of atomic assets, and a *feature model* which describes how the feature modules are combined. For the purpose of FFJ_{PL} Apel et al. abstract from the concrete representation of the feature model. For accessing the feature model a general interface is provided, which comprises auxiliary operations that allow to check whether features exist *never*, *sometimes*, or *always together* in the same program. Since FFJ_{PL} does not introduce a special operation which realizes variation points, these interface to the feature model is the only way to determine optional and mandatory parts.

Well-formedness conditions (represented as relations) are introduced for classes, refinements, and methods, respectively. Based on (i) the well-formedness conditions, (ii) the well-typedness of the FFJ_{PL} term, and (iii) a valid class table, the concept of well-typed FFJ_{PL} product lines is defined. For such a well-typed FFJ_{PL} product line with the corresponding well-formed lookup tables, it is shown that any program which can be derived with a valid feature configuration is again well-typed. In addition, the authors show that a FFJ_{PL} product line is well-typed (meeting the well-formedness conditions of the corresponding lookup-tables) if all programs that can be derived from the product line are well-typed.

By abstracting from the concrete kind of feature model the focus of FFJ_{PL} is not on how to integrate variability concepts into the introduced feature-oriented language. The operations which are provided in the abstract interface to the feature model do not facilitate to take into account the relation between mandatory, alternative, and optional features on a level as we do it with our axiomatization. For example, the interface does not allow to determine the connection between optional and alternative features in such an explicit way as our axiomatization does. Beside, the concept of commonalities between programs is also not explicitly supported. This implies that the degree of efficiency in which the type-check can be performed on a

FFJ_{PL} product line strongly depends on the way in which commonalities are taken into account in the underlying feature algebra.

In summary, the FFJ_{PL} approach clearly targets at the practical challenge of producing type-safe product lines in a specific feature-oriented, Java-like language. FFJ_{PL} does not focus on the nature of variability, and the constructional concepts of how optional, common and alternative parts are related, which is the main interest of our axiomatization. This is a fundamental difference and classifies the feature-oriented product line approach by Apel et al. to a different application area. However, the operations of the FFJ_{PL} interface (to an underlying feature model) can also be realized on basis of our axiomatization. In particular for theoretical considerations, where properties of FFJ_{PL} programs have to be shown that require a more in-depth consideration of software product family concepts, both approaches can be combined in a beneficial way.

The work [AKGL10] of Apel et al. takes up many ideas from a preceding paper [KA08] of Kästner and Apel from 2008. In [KA08], Kästner and Apel introduce the formal calculus *Color Featherweight Java (CFJ)* together with a set of type rules. CFJ is an extension of *Featherweight Java (FJ)* [IPW01] and realizes an annotation-based implementation of a product line. Variability in CFJ is implemented with *#ifdef*-like directives on basis of the annotations. Individual products are derived from an annotated CFJ program by removing those code fragments which correspond to unselected features. CFJ defines a type system for annotation-based product lines. Similarly to [AKGL10], the authors prove for CFJ that the generation of products preserves typing, i.e. that all programs (products) that are derived from an annotation-based product line are well-typed if the product line is well-typed itself. This prove can be done without generating and compiling the individual products first. In general, CFJ provides a type system which is realized on the preprocessor level, and which can easily be integrated into existing tool environments (based on a FJ or Java type system) since CFJ does not introduce new language constructs. However, even though the type check considers all possibly derivable products, the connection between the individual products is not explicitly (in the sense of a product family of sort $\text{SPF } \alpha$) modeled, since the annotations for individual features are not explicitly related to one other.

In [DCB09], Delaware et al. address the challenge of type-safety for a product line of Java-like programs. Delaware et al. refer to this challenge as *safe composition*. As a basis for their considerations the authors present *Lightweight Feature Java (LFJ)*, which is an extension of *Lightweight Java (LJ)* [SSP07] with support for features. For LFJ the authors define a constraint-based type system and prove its soundness, which means that any composition of features in LFJ that satisfies the typing constraints will generate a well-formed LJ program. More precisely, the typing rules allow to generate a set of constraints for each feature. These constraints

2. Formalization of Characteristic Software Product Family Concepts

are encoded into a SAT-instance. The satisfaction of the formula built from these SAT-instances indicates whether the corresponding LJ program is well-typed.

The authors introduce the typing rules for LFJ and LJ for each LJ construct individually. Since features are typically depending on other features, the typing rules for a single LJ (and LFJ) construct consider the specific structure of the respective construct, but also the dependencies and relations due to the surrounding program structure. Converting these type aspects into constraints provides an explicit interface for a LJ construct. Given a LFJ program, the type rules allow to generate constraints for each feature. Based on the feature constraints, checking safe composition of a product line reduces to showing that the programs allowed by the corresponding feature model are contained within the set of type-safe products. In LFJ, an entire product line results in a set of constraints that remain static regardless of the product specification being checked. In this sense such a set of constraints is similar to a product family of sort $\text{SPF } \alpha$ since both represent a set of products in its entirety.

Algebraic Treatment of Software Product Family Concepts

Regarding the algebraic treatment of software product families, there are some approaches which also aim to unify the common concepts, techniques and methods of feature-oriented approaches by providing an abstract, common, formal basis. In this context, we consider especially the approaches [HKM06, HKM09, BO92] to be relevant.

The closest to our axiomatization of a software product family is an approach by Höfner et al. [HKM06, HKM09], introducing the notion of a *feature algebra*, and a *product family*, respectively, which describes the features of a family of products, and their typical operations from a mathematical, group-theoretic, algebraic perspective. More precisely, a feature algebra is represented as the algebraic structure of a commutative idempotent semiring.

Elements of a feature algebra are called product families. A product family corresponds to a set of products, where individual products are considered to be flat collections of features. In general, the structure of a feature algebra largely coincides with the structure of a software product family of sort $\text{SPF } \alpha$, as it can be built using the constructors (cf. Section 2.2.2) only. In contrast to our variants operator, which is labeled with a unique name, the corresponding operators in the approach of Höfner et al. are not numbered at all. Thus, different variation points can not be told apart, and the notion of configuration as it exists in our approach does not exist in the approach of Höfner et al., since the configuration of individual variation points cannot explicitly be specified or referred to. Apart from the constructors, for product families of sort $\text{SPF } \alpha$ in our axiomatization we additionally define functions that characterize how to manipulate and work with a product family, e.g. the

selection operators `selL` and `selR`, function `is_mand`, etc. Corresponding operations do not exist in the approach of Höfner et al. either.

For feature algebras, the notion of refinement of product families exists. Refinement relates several product families and is based on the subset relation of the features of products. Methodologically, refinement is applied when a new product family is constructed from an existing one by adding new features. While we have not explicitly entitled it refinement in our approach, we can express a similar concept with axioms that characterize the notion of configuration and composition.

Höfner et al. state that a general aim of the feature algebra approach is “*to underpin the ideas of family-based development with a formalism that allows a mathematically precise description and manipulation of product families*”, which is a similar motivation as for our approach. However, being an algebraic specification our approach tends more in the direction of characterizing the class of “valid” software product families, and checking that a concrete computation structure is actually an “instance” of the abstract sort `SPF` α .

In contrast to the work of Höfner et al.

- we provide a *unique normal form* (cf. Section 2.2.3.2) for software product families, which allows to represent a software product family in a unique way, and which is a basis for dealing with the equality of software product families. While a corresponding notion of a normal form is not explicitly given in [HKM06, HKM09], Höfner et al. also make use of distributive laws to change the representation of product families. However, in contrast to our normal form, in the approach of Höfner et al. uniqueness of such restructured representations is not explicitly investigated. The kind of representation which is encountered frequently in the feature algebra approach integrates the common parts as far as possible into the corresponding products, and thus aims at the quite opposite representation as we use it in our normal form, where common parts are factored out from alternative variants as far as possible.
- While the approach of Höfner et al. is done in a pure algebraic, mathematical setting, in our approach we identify the characteristic functions in a less group-theoretic way from a more “practically motivated” perspective, in the sense that we do not only introduce the constructors that are required to represent product families, but also formalize many operations which represent the interaction and the manipulation that is typical for a software product family, e.g. configuration selection or the check for common parts by means of the function `is_mand`. In particular, in the presence of these extra functions we focus on the properties and laws which can be derived for these functions, and characterize them to reason about software product families. This allows to reason about concepts like selection, common mandatory, and optional parts, which are motivated from a realistic point of view.

Product Family CCS—A Framework for the Specification of the Behaviors of a Set of Systems as a Product Family

In contrast to the general characterization of fundamental software product family concepts which we have undertaken in the preceding chapter by means of an axiomatization, we consider in this chapter product families for a specific purpose. Now, the operational functionality of a set of reactive systems is represented in an integrated way as a product family, and an individual product of the family corresponds to the operational behavior of a single system. In order to model such product families we introduce the process algebraic specification framework *Product Family CCS (PF-CCS)*. PF-CCS is based on Milner’s process algebra CCS, and essentially extends it with concepts that allow to model the variability within the behaviors of a set of systems in a way that respects the laws required by the axiomatization. We introduce the syntax of PF-CCS and develop a semantics in terms of multi-valued labeled transition systems.

Contents

3.1. Syntax of PF-CCS	111
3.2. Semantics of a PF-CCS Program	121
3.3. Design Decisions for PF-CCS	147
3.4. Practicability of PF-CCS	150
3.5. Related Work	155

3. PF-CCS: Product Family CCS

As we have seen in the preceding chapter, many different kinds of software product families are constructed in various application domains and for various purposes. The product families differ in their scope, the kind of products, the kind of commonalities, the kind of assets and the entire construction mechanism for assembling the products, etc. For example, while one company develops all software-relevant parts of an entire family of similar ship systems [BCKB03] using a software product line engineering approach, another company develops the application software for a set of mobile phones as a software product family (see [SEI]). Obviously, both families and their derivable products are completely different and not comparable. Still, both families incorporate the same fundamental construction principle, which is fundamental to every software product family. In the last chapter we have formally characterized these concepts in a realization and domain independent way by means of an axiomatization (cf. Figure 2.11 on Page 90).

Now, in this chapter we move away from the treatment of universally valid software product family concepts and consider software product families for a very specific scope and purpose. We model the *operational functionality* of a set of software-intensive, reactive systems as a product family in an integrated way which is independent of the implementation on a specific software/hardware platform and architecture. For this purpose we introduce the process algebraic framework *Product Family CCS* (PF-CCS). PF-CCS combines and adds software product family concepts—as we have introduced them in the preceding chapter—with established process algebraic techniques for the representation of the behavior of single systems in an operational though implementation-independent way.

Basically, PF-CCS is a process algebra based on Milner’s *Calculus of Communicating Systems* (CCS) [Mil80]. While the process calculus CCS itself can be used to represent the operational behavior of single systems, PF-CCS extends the capabilities of CCS as it allows to model the behavior of entire families of such systems. Thereby, the notion of *family* corresponds exactly to the one we have defined by means of the axiomatization in the preceding chapter. For example, with PF-CCS we can represent the operational behavior of families of coffee vending machines, the behavior embodied in a variable network protocol between a variety of senders and receivers, the behavior of a swarm of interacting insects [Tof92], and the behavior of a family of automotive screen wiper systems. Particularly with regard to behavioral variety, PF-CCS establishes the conceptual basis to deal with that kind of variety which we encounter in the operational functionality of families of reactive, software-intensive systems as they are typical for the automotive domain.

However, PF-CCS is not designed to be a specification framework that can directly be applied as it is for the practical development of families of software-intensive systems in the current industrial practice. In particular, PF-CCS does not aim at the specification of large industrial-size system families. Similarly to the underlying process algebra CCS that was never designed to specify the behavior of large reactive

systems, also the focus of the PF-CCS framework is not on providing a technically mature, perfectly scaling solution that allows to specify the operational behavior of industrial-size system families like for example entire automotive model series. In this light, the PF-CCS specification framework addresses rather the theoretician than the practitioner. Nevertheless, as the specification of the operational behavior of automotive system families is—at least to our knowledge—currently not undertaken in an integrated way as it is possible with PF-CCS, a framework like PF-CCS is a useful step to tackle the current challenges in automotive software engineering. In this light, we see PF-CCS as a conceptual solution to master the large variety in the operational behavior of similar systems, and as a fundamental technique which lays the ground for frameworks that aim at the application in the real-life practical context. Although being designed as a conceptual framework, in Section 3.4 we discuss some aspects of PF-CCS, e.g. the positioning of PF-CCS in an adjusted development process, that relativize its applicability in a practical context.

Technically, PF-CCS subsumes all concepts of CCS for the specification of non-variable behavior. In particular, PF-CCS provides all the operators known from (basic) CCS, and additionally introduces a variants operator. With respect to the original CCS operators PF-CCS is a “conservative” extension of CCS in the sense that the semantic of the original operators is preserved. The variants operator of PF-CCS allows to model variation points as characterized and introduced in the axiomatization in the preceding chapter, i.e. it allows to model a (deterministic) choice between alternative variants. Altogether, PF-CCS is designed in a way that fulfills the axioms introduced in the last chapter, i.e. any process algebraic structure which can be specified in PF-CCS is a software product family in the sense of the axiomatization given in the last chapter.

Beside the ability to capture the operational functionality of a family of systems as a software product family, PF-CCS (1) comes with a multi-valued logic that allows to reason about the integrated behavior of the members of a software product family and the product family itself, as we will discuss in detail in Chapter 4, (2) facilitates the application of automatic verification techniques like model checking, and (3) provides the theoretical basis to restructure a software product family in order to find commonalities in the operational behavior of its products (cf. Chapter 5).

3.1. Syntax of PF-CCS

PF-CCS is a process algebra based on Milner’s CCS [Mil80]. It allows to specify the behavior of a product family in a process algebraic fashion as a set of equations over processes, actions and operations between those. Similarly to other process algebras such as CCS, ACP [BK84], or CSP [Hoa85], we express the concept of behavior

3. PF-CCS: Product Family CCS

(functionality) in PF-CCS in terms of processes which perform actions, too. Let Id be a finite set of *process identifiers*. Usually, we use capital latin letters such P, Q, P_1, \dots to denote process identifiers. The identifier Nil is reserved for a special process, the so called atomic *idle process*. Nil is a process which can not perform any action, in particular we can understand Nil as the (successful) termination of a system. Further, let Σ be a finite set of *input actions*. Usually, we use lowercase latin letters such as a, b, \dots to range over input actions. Let

$$\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$$

be the set of *output actions*. As in CCS, let

$$\mathcal{A} = \Sigma \cup \bar{\Sigma} \cup \{\tau\}$$

represent the set of *communication actions*, where

$$\tau \notin \Sigma \cup \bar{\Sigma}$$

represents a special action, the so-called *silent action*, which is used to model (and to abstract from) the internal communication between concurrent processes. Usually, lowercase Greek letters α, β, \dots range over communication actions.

Definition 3.1 (Syntax of PF-CCS Process Expressions). *The set \mathcal{L}_{PFCCS} of all PF-CCS process expressions (also called processes) is generated by the following EBNF grammar*

$$\begin{array}{l}
 P = \quad Q \\
 \quad | \text{'Nil'} \\
 \quad | \alpha \text{'.'} P \\
 \quad | P \text{'+' } P \\
 \quad | P \text{'\oplus'} P \\
 \quad | P \text{'\parallel'} P \\
 \quad | P \text{'[f]'} \\
 \quad | P \text{'\lambda'} L ;
 \end{array}$$

where

- $Q \in Id$ is a process identifier
- $\alpha \in \mathcal{A}$ is an action
- $L \subseteq \mathcal{A}$ is a set of action labels
- $f : \mathcal{A} \rightarrow \mathcal{A}$ is an action renaming function, i.e. a function with the properties (i) $f(\bar{a}) = \overline{f(a)}$, and (ii) $f(\tau) = \tau$.

Except for \oplus —which represents the variants operator—the symbols represent the operations as known from CCS [Mil80]. The intuitive meaning of the operations is the following:

- The symbol $.$ represents action prefixing, i.e. the process $\alpha.P$ represents a process which can perform the action α and then behaves like process P . Action prefixing is a special case of sequential composition.
- The symbol $+$ represents non-deterministic choice, i.e. the term $P + Q$ represents a process which either behaves like the process P or Q . The choice between P or Q is made by the environment, i.e. depending on whether the action to be performed is the first one of P or Q . If P and Q have the same initial action, the choice is made nondeterministically. Since $+$ is associative and commutative¹, we write

$$\sum_{i \in \{1 \dots n\}} P_i$$

to represent the more general situation of choosing non-deterministically one process out of the set of n processes $\{P_1, \dots, P_n\}$. Per definition the sum

$$\sum_{i \in \emptyset} P_i := Nil$$

over an empty set of processes is equal to the idle process Nil .

- The symbol \parallel represents the parallel composition of processes, where processes P and Q operate concurrently (asynchronously) but can communicate only synchronously with each other. Such a kind of “internal” communication is abstracted by the silent action τ . Note that Milner usually [Mil95] uses the symbol $|$ instead of \parallel for parallel composition.
- The symbol $[f]$ represents the renaming operator which allows to rename the actions performed by a process P as specified by the function f . More precisely, the process $P[f]$ behaves like P , but with all performable actions α renamed to $f(\alpha)$.
- The symbol \backslash represents the restriction operator which restricts the set of actions that a process P can perform in a certain environment. For any action $\alpha \notin L$, the process $P \backslash L$ can interact with any environment in the same way as the process P , while it can not perform any actions $\alpha \in L$. Restriction is typically used to control the communication between parallel processes.
- The symbol \oplus represents the *variants operator*, which has no counterpart in CCS. Thus, syntactically, PF-CCS extends CCS [Mil80] only by the binary *variants operator* \oplus . The variants operator allows to specify a special form of alternative behavior, i.e. it represents a variation point (as defined in Equation 2.4 in Chapter 2.2.2). More precisely, the process $P \oplus Q$ represents a variation

¹For the PF-CCS $+$ operator we observe the algebraic laws: (i) $P + Q = Q + P$ (Commutativity) and (ii) $P + (Q + S) = (P + Q) + S$ (Associativity).

3. PF-CCS: Product Family CCS

point which behaves exactly like one of its two alternative processes P and Q . However, the choice between P or Q is made deterministically (in contrast to the nondeterministic choice operator $+$) according to a given configuration. In particular, the configuration of a variants operator \oplus is not affected by any other process or CCS construct and represents a conceptually different dimension as it describes the static structure of PF-CCS processes while the “original” CCS operators like action prefixing and non-deterministic choice model the dynamic (behavioral) aspect of processes. Note that—as described in the last chapter in Section 2.46—we can understand optional parts as a special case of alternative choices. Similarly to the optional operator `optional` (cf. Equation 2.46, Page 77), an optional-operator $\langle _ \rangle$ can be added to PF-CCS as the syntactical abbreviation:

$$\langle P \rangle := P \oplus Nil$$

In Section 3.2, where PF-CCS semantics is discussed, we will see that this abbreviation meets our intuition, allowing us to confine in this technical presentation of PF-CCS to the variants operator \oplus only.

Similarly to CCS, also in PF-CCS processes can be specified by means of (recursive) equations. A *process definition* (*constant definition*) is a defining equation of the form

$$P \stackrel{def}{=} t(P_1, \dots, P_n)$$

where $P \in Id$ is a process identifier and $t(P_1, \dots, P_n) \in \mathcal{L}_{PFCCS}$ is a PF-CCS process (term) which contains the process identifiers (P_1, \dots, P_n) .

Since process terms (right-hand side of the equations) can contain process identifiers on their part, processes can be defined by mutual recursion in terms of each other, e.g. as in

$$\begin{aligned} P &\stackrel{def}{=} \alpha.Q \\ Q &\stackrel{def}{=} \beta.P \end{aligned}$$

This gives reason to recursive specification schemes which can be used to specify the behavior of an entire product family. We call a recursive specification scheme *PF-CCS program*.

Definition 3.2 (PF-CCS Program). *A PF-CCS program is a tuple (\mathcal{E}, P_1) which consists of a finite set of (possibly recursive) process definitions*

$$\mathcal{E} = \left\{ \begin{array}{l} P_1 \stackrel{def}{=} t_1(P_1, \dots, P_n) \\ P_2 \stackrel{def}{=} t_2(P_1, \dots, P_n) \\ \vdots \\ P_n \stackrel{def}{=} t_n(P_1, \dots, P_n) \end{array} \right\}$$

and a distinguished main process identifier $P_1 \in Id$.

For the sake of simplicity, we typically denote a PF-CCS program by listing its equations only, assuming that the left-hand side of the first equation is the main process identifier.

In compliance with results from CCS and ACP [BK84] we only consider PF-CCS programs with a special kind of recursion, so-called *guarded recursion*, where all processes are (*action-*) *guarded*. According to Milner [Mil95] a CCS process X is *guarded* in a term E if each occurrence of X is within some subterm $\alpha.F$ of E , which performs initially an action α . For a precise definition for guarded recursion see [Mil95, Fok00]. For PF-CCS, this means that every process identifier P_i which appears in a process term $t_i(P_1, \dots, P_n)$ of a PF-CCS program has to be action guarded in the above sense. The main benefit of guarded recursive equations is that they always have unique solutions (see e.g. [Mil95, Fok00]).

Connection of the PF-CCS Operators to the Axiomatization

The PF-CCS operators \parallel and \oplus are of particular interest with respect to the axiomatization given in Chapter 2. In PF-CCS, the (CCS) parallel composition takes the role of the composition function (cf. Equation 2.3, Page 35) in the axiomatization: It allows to specify a (static) structure of CCS processes in a way which respects the axioms required for the composition function in the axiomatization. Letting the CCS parallel composition \parallel take the role of the composition function was a natural choice: Beside the restriction and re-labeling operators of CCS, the parallel composition is the only so-called *static operator* according to the classification of CCS operators of Milner [Mil95]. In CCS, static operators are those operators which define the (static) structure of a CCS process, i.e. how the sub-processes are linked. Thus, the parallel composition of PF-CCS fulfills the same purpose as the composition function in the axiomatization, which was used to compose different structural units, e.g. assets and compound objects.

Regarding the variants operator in PF-CCS, it takes the role of the function \oplus (cf. Equation 2.4, Page 36) which represents variation points in the axiomatization. The PF-CCS variants operator is designed to exhibit the same properties which are required from the variation points in the axiomatization. In particular, the variants operator of PF-CCS represents a conceptually new kind of deterministic alternative choice which cannot be modeled with the original operators of CCS.

For the sake of simplicity we use the same symbols \oplus and \parallel for denoting the PF-CCS operators which we have already used to represent the respective functions for composition and variation points in the axiomatization in the preceding chapter. Together, the PF-CCS operators \oplus and \parallel are used to extend CCS with software product family concepts as defined in the axiomatization of the preceding chapter. In particular, the distributive laws as propagated in the axiomatization also hold in PF-CCS. This is the basis for extracting common behavioral parts as described in the upcoming Chapter 5.

3. PF-CCS: Product Family CCS

3.1.1. Well-formed PF-CCS programs.

In the following we introduce some restrictions on the syntax and the (term) structure of PF-CCS programs. These restrictions serve two purposes: On the one hand, they realize some basic properties which we require from every rational (process algebraic) equational specification. On the other hand, they implement some of the properties required by the axiomatization (cf. Chapter 2.2, Page 90), especially with respect to variants operators.

The syntactical restrictions are achieved by three conditions: *completeness*, *finitely configurable*, and *fully expanded*. These conditions are used to derive the notion of *well-formed* systems. For well-formed PF-CCS programs we can define a *compositional* semantics (cf. Section 3.2.2) in the sense that if a PF-CCS program is well-formed, we can label the \oplus -operators in a unique way after having specified the entire program. This means that we can write down an entire program first without having to attach numbers to the \oplus -operators. If the final program fulfills our well-formedness constraints, we finally can simply number all occurring \oplus -operators consecutively in the order of their appearance while guaranteeing the properties required in the axiomatization of a general software product family. This procedure enhances the specification process. In the remainder of the section we successively introduce the syntactical restriction.

Complete PF-CCS Programs

The first condition is essential for any equational specification scheme, and not specific to PF-CCS. It states that the behavior of all process identifiers which are used in the program has to be specified as part of the program, and that a PF-CCS program contains no undefined process identifiers.

Definition 3.3 (Complete PF-CCS Program). *We call a PF-CCS program with the set of process definitions $\{P_1 \stackrel{\text{def}}{=} e_1, \dots, P_n \stackrel{\text{def}}{=} e_n\}$ complete, if all process identifiers P_i on the left-hand sides of the defining equations are pairwise distinct and the defining equations e_1, \dots, e_n contain only the process identifiers P_1, \dots, P_n .*

The remaining conditions are concerned with the questions of writing PF-CCS programs with only finitely many variants operators, and of numbering the variants operators in a PF-CCS program in a unique way. Recall that a unique numbering (labeling) of the variation points and the existence of only finitely many variation points are essential properties which we required in the axiomatization for every software product family. Since in PF-CCS we usually use the variants operators in a specification without associating unique identifiers to them initially, we have to make sure that PF-CCS programs facilitate a numbering afterwards. A PF-CCS program has to meet certain conditions in order that these requirements can be fulfilled.

PF-CCS Programs with Finitely many Variants Operators

Our goal is to model software product lines which require only an *a priori finite* number of decisions taken at variation points when deriving a specific system, which is the case for all product lines relevant in practice. So far, however, as in CCS, PF-CCS allows the creation of new processes by using the parallel operator \parallel within recursive process definitions. In combination with our \oplus -operator this may potentially result in an unbounded number of variation points. In order to cope with this issue we consider the way in which processes (and the defining equations respectively) depend on each other. For this purpose we now turn towards the definition of a dependency graph of a PF-CCS program, which—similarly to a control flow graph for programming languages—reflects the (recursive) dependencies of process definitions in a program. For a PF-CCS process term e , let $pt(e)$ denote the parse tree of e defined in the usual manner (see e.g. [ASU86]) as a tree labeled with operator symbols or process identifiers (in leafs).

Definition 3.4 (Program Dependency Graph). *Let $(\{P_1 \stackrel{def}{=} e_1, \dots, P_n \stackrel{def}{=} e_n\}, P_1)$ be a complete PF-CCS program. We define its program dependency graph as the directed labeled graph (V, E) given as follows:*

- *Its nodes V comprise those for left-hand sides of the equations labeled with P_1, \dots, P_n , together with the nodes of the parse trees $pt(e_1), \dots, pt(e_n)$ of the defining equations e_1, \dots, e_n .*
- *Its edges E comprise all edges of the parse trees $pt(e_1), \dots, pt(e_n)$, and edges connecting the nodes P_i corresponding to the left-hand sides of equations to the roots of parse trees $pt(e_i)$ of the corresponding defining equations e_i . In addition, it comprises all edges from leafs of the parse trees labeled with single process identifiers P_i to the nodes P_i representing the left-hand side of the defining equation $P_i \stackrel{def}{=} e_i$.*

A program dependency graph basically represents an enriched “parse tree” of a PF-CCS program with also contains its recursive call dependencies. As an example, consider the following PF-CCS program whose program dependency graph is shown in Figure 3.1.

$$\begin{aligned} P &\stackrel{def}{=} (\alpha.P) \oplus (Q \parallel Q) \\ Q &\stackrel{def}{=} \beta_1.Nil \oplus \beta_2.Nil \end{aligned}$$

We call a node labeled Q *reachable* from a node labeled P if there exists a path from P to Q in its program dependency graph.

3. PF-CCS: Product Family CCS

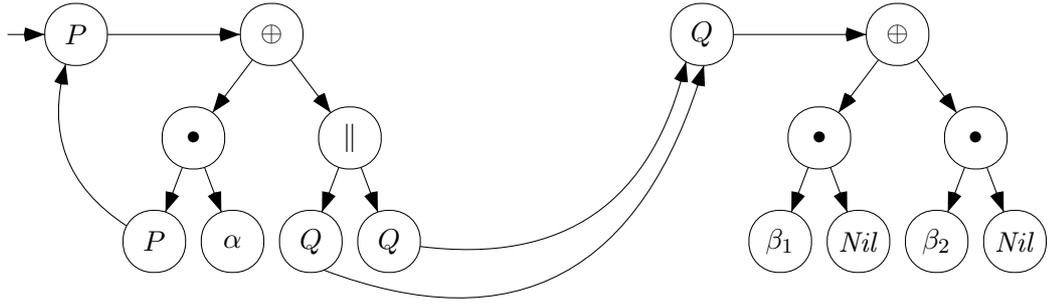


Figure 3.1.: A program dependency graph.

Intuitively, a program dependency graph reflects the dependencies between the process identifiers of a PF-CCS program with respect to their defining equations. A cycle in this graph that contains a node labeled with a parallel operator \parallel might represent a recursive process definition “spawning” an arbitrary number of copies of its own. Consider for example the following process which represents a set of arbitrary many parallel processes.

$$\begin{aligned}
 P &\stackrel{\text{def}}{=} \alpha.P \parallel \beta.Q \\
 &= (\alpha.P \parallel \beta.Q) \parallel \beta.Q \\
 &= ((\alpha.P \parallel \beta.Q) \parallel \beta.Q) \parallel \beta.Q \\
 &= \dots
 \end{aligned}$$

If in such a context, the variants operator \oplus comes into play (e.g. if Q contains an variants operator), an *unbounded* number of configuration selections would be possible, since with every newly spawned Q we would get a new variants operator, as well. In particular, if all variants operators are labeled differently, a realistic configuration is not performable any more. We therefore consider in the following PF-CCS programs which forbid such a situation and thus are configurable within finitely many configuration selections.

Definition 3.5 (Finitely Configurable PF-CCS Program). *We call a complete PF-CCS program finitely configurable, if its program dependency graph has no cycle containing a node labeled with \parallel from which a node labeled with \oplus is reachable.*

Consider Figure 3.1. While there is a cycle from P back to P from which a \oplus -operator is reachable, the program is finitely configurable as this cycle does not contain a node labeled \parallel . Thus, the variants operators are not part of a (recursive) process which gets spawned infinitely often. However, if we would specify instead $P \stackrel{\text{def}}{=} (\alpha.P) \parallel (Q \parallel Q)$, the program would not be finitely configurable, as the cycle from P to P would contain the parallel operator, and, still the \oplus -operator of the second equation is reachable.

$$\begin{array}{ll}
P \stackrel{\text{def}}{=} Q \parallel Q & R \stackrel{\text{def}}{=} Q_1 \oplus Q_2 \parallel Q_1 \oplus Q_2 \\
Q \stackrel{\text{def}}{=} Q_1 \oplus Q_2 &
\end{array}$$

(a) First possibility. (b) Second possibility: The program is fully expanded.

Figure 3.2.: Two ways of understanding alternative variants and variation points.

Note that the definition of finitely configurable does not *characterize* the programs that are configurable within finitely many configuration selections, but is just a sufficient condition. However, as it is (already) undecidable whether a CCS program yields a finite or infinite state system [Mil95], it is easy to see that it is also undecidable whether the transition system defined by a PF-CCS program would make use of only finitely many configuration selections. In the following, we therefore consider only finitely configurable PF-CCS programs.

Fully Expanded PF-CCS Programs

There is a further restriction we want to make. It is concerned with the intended meaning of substituting a process definition which contains a variants operator. Consider for example the two independent systems P and R shown in Figure 3.2. When considering the left system P one might understand its meaning as follows:

- (a) P consists of two “instances” of the same variation point Q . Hence, one selects once between Q_1 and Q_2 and follows this choice for any occurrence of Q in P . However, if we would specify P by substituting the process Q in P by its defining equation $Q_1 \oplus Q_2$, we would get a system like R , which represents a quite different intention:
- (b) In R , we now have two (independent) variation points, which—though offering the same variants Q_1 and Q_2 —might be configured differently from each other.

So far, the structural semantics rules, as we introduce them in Section 3.2.2, are only compositional for meaning (a). In particular, if we want to model two independent choices, we have to explicitly model them as shown in Figure 3.2b. Therefore—for the scope of this thesis and to simplify the technical treatment—we only consider systems like R , where every variants operator can be configured independently from the configuration of other variation points (with different names). Note that it is easy to extend our formalism to actually cope with both meanings by introducing a second kind of variants operator with a suitable semantics for the case of Figure 3.2a. However, as this does not match our understanding of a software product family, we refrain from giving this extension in this thesis. Moreover, the alternative to assign labels directly to variants operators at the time of writing down the PF-CCS program also gives the possibility to deal with such a situation in the current setting.

3. PF-CCS: Product Family CCS

We call PF-CCS programs that only contain variation points of the kind shown in Figure 3.2b *fully expanded*. The property of being fully expanded can be checked by inspecting the corresponding program dependency graph of a PF-CCS program. If each \oplus -node can only be reached on one path from any other node in the program dependency graph, then the corresponding program is fully expanded. Otherwise, if a \oplus -node can be reached on at least two different paths, we must have a situation like in Figure 3.2a, where the program is not yet fully expanded. Thereby, we have to take cycles in the program dependency graph into account, since the two paths must be “really” different, i.e. different runs through a path which contains a cycle, by looping through the cycle a different number of times, do not count as different paths. We achieve this by considering only cycle free paths, where we call a path (n_1, n_2, \dots, n_m) *cycle free*, iff it does not contain any node twice, i.e. iff $\forall i, j \in \{1, \dots, m\} : n_i \neq n_j$.

Definition 3.6 (Fully Expanded PF-CCS Program). *We call a complete and finitely configurable PF-CCS program fully expanded, if every \oplus -node in its program dependency graph can be reached from the start node by at most one cycle free path.*

The property of being reachable from any node is equivalent to being reachable from the start node, since in a complete program any node is reachable from the start node. Note that a finitely configurable PF-CCS program which is not fully expanded can be transformed into an equivalent fully expanded version. Certainly, this is not to be understood in a mathematical sense, as no semantics for non-fully expanded programs has and will be provided, which does not allow to define *equivalence* precisely. For example, the program whose program dependency graph is shown in Figure 3.1 is not fully expanded since the \oplus -node on the right is reachable from the start node by two different, cycle free paths. The Definitions 3.3 to 3.6 can be subsumed characterizing the set of *well-formed* PF-CCS programs.

Definition 3.7 (Well-formed PF-CCS Program). *We call a PF-CCS program well-formed, if it is complete, finitely configurable, and fully expanded.*

The rationale for the syntactical restrictions leading to Definition 3.7 is that in a well-formed PF-CCS program we can easily label each variants operator with a unique natural number by parsing over the PF-CCS program and attaching a fresh number to every occurrence of a variants operator. In contrast, for non well-formed programs, we have to number a variants operator immediately when we add it to the PF-CCS specification. The ability of numbering variants operators in a unique way allows us to precisely define the concept of a variation point with respect to PF-CCS programs, which conforms to the corresponding concept of a variation point as required in the axiomatization in Section 2.2.2.3 (Page 36).

Definition 3.8 (Variation Point in PF-CCS). *In PF-CCS, we call a uniquely labeled variants operator with number $i \in \mathbb{N}$, denoted by \oplus_i , a variation point.*

3.2. Semantics of a PF-CCS Program

In the following, we define the semantics of a PF-CCS program. We do this in an intuitive way by introducing three different, subsequent semantics, the *flat semantics*, the *unfolded semantics*, and the *configured-transitions semantics*. In particular we show how they are related. Basically, the first two semantics are only introduced to motivate and justify the final semantics, the configured-transitions semantics, which will be an appropriate basis for specifying and model checking properties of software product line as introduced in Chapter 4.

3.2.1. Flat Semantics

The *flat semantics* reflects the intuitive understanding of a PF-CCS program representing a product family: Every PF-CCS program can be understood as the set of all (plain) CCS programs that can be derived by a full configuration of the PF-CCS program. More precisely, given a well-formed PF-CCS program, we choose for every variants operator either the process term on its left- or right-hand side and remove all the unselected terms together with the respective \oplus symbols from the PF-CCS program. For every complete configuration, this procedure results in a plain CCS program, which can be understood in the usual way, e.g. with the SOS semantics described by Milner [Mil80].

Technically, in order to talk about configurations for certain variation points, we have to label every variants operator \oplus uniquely with a number in $\{1, \dots, n\}$. As we have seen in the preceding section, we can always do this for *well-formed* PF-CCS programs by simply parsing a PF-CCS program line by line and numbering every variants operator with the next fresh number when we parse it. For programs which are not well-formed we have to label the variants operators manually at the time when we add it to the PF-CCS program.

We denote a configuration by means of *configuration vector*, which stores the individual configuration selections for the corresponding variants operators.

Definition 3.9 (Configuration Vector). *A configuration vector $\theta \in \{R, L, ?\}^n$ is a vector of the form $\langle c_1 c_2 \dots c_n \rangle$ where each c_i represents the individual configuration for the i^{th} variants operator \oplus_i of the corresponding PF-CCS program. We call a configuration vector $\theta \in \{R, L, ?\}^n$ fitting to a PF-CCS program containing m variation points, if $n = m$.*

In order to work with configuration vectors we use the following operations and notation: Let $\nu \in \{R, L, ?\}^n$ be a configuration vector and $0 < i \leq n$ be an index

3. PF-CCS: Product Family CCS

number. By ν_i we denote the i^{th} element of ν . The construct $\nu|_{i/x}$ represents the updated vector ν in which the entry at the i^{th} position is replaced by the value $x \in \{R, L\}$. All other entries keep their values, i.e. $\forall j \neq i : (\nu|_{i/x})_j = \nu_j$. In order to separate the entries more explicitly, we sometimes denote a configuration vector using commas as in $\langle R, R, L \rangle$. With $\langle ?^n \rangle$ we denote the configuration vector consisting of n ?-entries.

In a configuration represented by a vector $\langle c_1, \dots, c_n \rangle$, the values of single entries c_i have the following meaning: The value R represents the selection of the right variant (of variation point i), and L represents the selection of the left variant. The entry $?$ represents the situation that none of the two alternative variants has been selected for the corresponding variation point. This means that the configuration of a variation point is not relevant for the resulting product, i.e. that products with both kinds of variants exist. For example, the vector $\langle R, R, L \rangle$ denotes the configuration for a software product family containing three variants operators, where for the first two variants operators \oplus_1 and \oplus_2 the right variant (R) is selected, and for the third variants operator \oplus_3 the left variant (L) is selected.

A special kind of configuration vectors are those which contain no ?-entries, as such configuration vectors always correspond directly to a product of the software product family.

Definition 3.10 (Fully Configured Configuration Vector). *We call a configuration vector $\theta \in \{R, L, ?\}^n$ fully configured if $\forall i \in \{1, \dots, n\} : \theta_i \neq ?$.*

Configuring a software product family according to the configuration represented by a fully configured configuration vector results in a concrete product, which contains no variability anymore. We represent the act of applying a configuration to a PF-CCS program by the function *config*. Let \mathcal{L}_{CCS} denote the set of all CCS programs. Given a well-formed PF-CCS program *Prog* with n variants operators and a fully configured configuration vector $\theta \in \{R, L, ?\}^n$ we define the function

$$\text{config} : \mathcal{L}_{PFCCS} \times \{R, L\}^n \rightarrow \mathcal{L}_{CCS}$$

which realizes a configuration on the term structure of a PF-CCS program. This means that *config* reduces *Prog* to a CCS program P_{CCS} , where P_{CCS} is constructed by removing all *subterms* in the PF-CCS program *Prog* which are not selected according to θ . Thus, *config* manipulates the actual PF-CCS process expression and removes the unselected variants together with the respective \oplus tokens from the PF-CCS equations. The function *config* is inductively defined as shown below. Let P, T be PF-CCS processes, $\alpha \in \mathcal{A}$ a communication action, $f : \mathcal{A} \rightarrow \mathcal{A}$ a renaming function, $L \subseteq \mathcal{A}$ a set of actions, $Q \in Id$ an atomic process identifier, and $\theta \in \{R, L\}^n$ an corresponding configuration vector.

3.2. Semantics of a PF-CCS Program

$$\mathit{config}(P \oplus_i T, \theta) = \begin{cases} \mathit{config}(P, \theta) & , \theta_i = L \\ \mathit{config}(T, \theta) & , \theta_i = R \end{cases} \quad (3.1)$$

$$\mathit{config}(Q, \theta) = Q \quad (3.2)$$

$$\mathit{config}(\mathit{Nil}, \theta) = \mathit{Nil} \quad (3.3)$$

$$\mathit{config}(\alpha.P, \theta) = \alpha.(\mathit{config}(P, \theta)) \quad (3.4)$$

$$\mathit{config}(P + T, \theta) = \mathit{config}(P, \theta) + \mathit{config}(T, \theta) \quad (3.5)$$

$$\mathit{config}(P \parallel T, \theta) = \mathit{config}(P, \theta) \parallel \mathit{config}(T, \theta) \quad (3.6)$$

$$\mathit{config}(P[f], \theta) = (\mathit{config}(P), \theta)[f] \quad (3.7)$$

$$\mathit{config}(P \setminus L, \theta) = (\mathit{config}(P, \theta)) \setminus L \quad (3.8)$$

In order to configure a PF-CCS program $\{P_1 \stackrel{\text{def}}{=} e_1, \dots, P_n \stackrel{\text{def}}{=} e_n\}$ according to a configuration θ we apply the function config to all right-hand sides e_i of the defining equations. This results in a CCS program given by the following set of equations:

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \mathit{config}(e_1, \theta) \\ &\vdots \\ &\vdots \\ P_n &\stackrel{\text{def}}{=} \mathit{config}(e_n, \theta) \end{aligned}$$

Consider for example the following PF-CCS-program:

$$\begin{aligned} P &\stackrel{\text{def}}{=} (\alpha.P) \oplus_1 T \\ T &\stackrel{\text{def}}{=} \beta_1.\mathit{Nil} \oplus_2 \beta_2.\mathit{Nil} \end{aligned} \quad (3.9)$$

By following the instructions given above, we get for the configuration $\theta = \langle R, L \rangle$ the plain CCS program:

$$\begin{aligned} P &\stackrel{\text{def}}{=} T \\ T &\stackrel{\text{def}}{=} \beta_1.\mathit{Nil} \end{aligned}$$

Note that due to the nested structure of a PF-CCS expression there are also not fully configured vectors which yet determine a single CCS system. For a more detailed discussion see Chapter 2.2.3.3. For example for the software product family specified in the preceding PF-CCS program 3.9 the configuration $\langle L, ? \rangle$ already yields the plain CCS program $P \stackrel{\text{def}}{=} \alpha.P$, even though $\langle L, ? \rangle$ is not fully configured.

By means of the function config we define the *flat semantics* of a PF-CCS program. The flat semantics is the set of all plain CCS programs (interpreted as labeled transitions systems in the conventional CCS SOS semantics) which can be derived by applying a configuration to the (term structure of a) PF-CCS program. Thus, it is the same as specifying a software product family by explicitly writing down the set of all CCS programs directly and interpreting them in the original CCS SOS semantics shown in Figure 3.3.

3. PF-CCS: Product Family CCS

$$\frac{P \xrightarrow{\alpha} P'}{C \xrightarrow{\alpha} P'} , C \stackrel{def}{=} P \quad (\text{constant definition}) \quad (3.10)$$

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad (\text{prefix}) \quad (3.11)$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad (\text{nondeterministic choice (1)}) \quad (3.12)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \quad (\text{nondeterministic choice (2)}) \quad (3.13)$$

$$\frac{P \xrightarrow{\alpha} P'}{(P \parallel Q) \xrightarrow{\alpha} (P' \parallel Q)} \quad (\text{parallel composition (1)}) \quad (3.14)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{(P \parallel Q) \xrightarrow{\alpha} (P \parallel Q')} \quad (\text{parallel composition (2)}) \quad (3.15)$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{(P \parallel Q) \xrightarrow{\tau} (P' \parallel Q')} \quad (\text{parallel composition (3)}) \quad (3.16)$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad (\text{relabeling}) \quad (3.17)$$

$$\frac{P \xrightarrow{\alpha} P'}{(P \setminus L) \xrightarrow{\alpha} (P' \setminus L)} , \alpha, \bar{\alpha} \notin L \quad (\text{restriction}) \quad (3.18)$$

Figure 3.3.: SOS rules for CCS as defined by Milner [Mil95].

Definition 3.11 (Flat Semantics of a PF-CCS Program). *Let $Prog \in \mathcal{L}_{PFCCS}$ be a PF-CCS program containing n variation points and $\theta \in \{R, L\}^n$ be a fitting configuration vector. The flat semantics of a product line $Prog$ is defined as*

$$\llbracket Prog \rrbracket_{Flat} = \{ \llbracket V \rrbracket_{CCS} \mid \exists \theta : (\text{config}(Prog, \theta) = V) \}$$

where $\llbracket V \rrbracket_{CCS}$ denotes the conventional CCS semantics of the CCS program V .

The conventional CCS semantics of a CCS program is given in terms of SOS (structured operational semantics) rules which precisely define how to construct a labeled transition system (LTS) from a CCS specification. Figure 3.3 shows the SOS rules for CCS as introduced by Milner in [Mil80]. We write $\llbracket Prog \rrbracket_{CCS.s}$ to denote the state s of the LTS $\llbracket Prog \rrbracket_{CCS}$.

Note that the flat semantics does not account for any feature constraints or feature dependencies. It simply consists of all combinatorially possible configurations of

a product line. However, such feature constraints can be incorporated in the flat semantics by further restricting the set of allowed configurations θ in the formula of Definition 3.11 by means of propositional logic, i.e. by means of the dependency model which have introduced in Chapter 2.3.

3.2.2. Unfolded Semantics

In the preceding section we have seen that in the flat semantics a PF-CCS program corresponds to a *set* of labeled transition systems (LTS), one for each fully configured configuration and thus for each product. Now, in the unfolded semantics, the meaning of a PF-CCS program is defined by a *single* labeled transition system representing an entire product family. This transition system actually is a product family in the sense of Chapter 2. In particular, by combining the behavior of all derivable systems within *one* labeled transition system, it represents the fundamental model for model checking—as we will show in Chapter 4—since now commonalities between systems can explicitly be considered and exploited. Before defining the unfolded semantics we introduce a specific form of a labeled transition system, a so-called *Product Line Labeled Transition System (PF-LTS)* which represents the semantical domain for the unfolded semantics.

Definition 3.12 (PF-LTS for the Unfolded Semantics). *A product family labeled transition system (PF-LTS) (representing the unfolded semantics) for a PF-CCS program containing n variation points is a tuple $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, \sigma)$, where*

- \mathcal{S} is a (countably, possibly infinite) set of states.
- \mathcal{A} is a finite set of communication actions,
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \{R, L, ?\}^n \times \mathcal{S}$ is a transition relation. If $(s, \alpha, \nu, s') \in \rightarrow$ we also write $s \xrightarrow{\alpha, \nu} s'$,
- and $\sigma \in \mathcal{S}$ is the start state.

With $\mathcal{T}.s$ we denote the state s of the PF-LTS \mathcal{T} . A PF-LTS for the unfolded semantics is basically a regular LTS which has a special kind of labels. More precisely, in a PF-LTS every transition from one state to another is labeled by an action α and an additional (possibly partial) configuration vector ν . We call an configuration vector attached to a transition also a *configuration label*. Usually, in this chapter we use the Greek letter ν to range over configuration labels, while θ ranges over configuration vectors which represent configurations.

The configuration label ν of a transition specifies in which configurations the transition exists. Thereby, a single entry ν_i at position i within ν means the following:

3. PF-CCS: Product Family CCS

- if $\nu_i = R$, then the transition is present only in those products (configurations) where the right variant for the variation point numbered with i is chosen,
- if $\nu_i = L$, then the transition is present only in those products (configurations) where the left variant for the variation point numbered with i is chosen,
- if $\nu_i = ?$, then the existence of the transition is not influenced by the specific configuration choice of the variation point labeled with number i , i.e. the transition is present in both cases where we chose either the right (R) or the left (L) variant for variation point i .

On basis of the individual entries we can define when an entire configuration matches a configuration label. More precisely, given a configuration θ representing a specific product (or subfamily), we can compare it with a configuration label ν of a transition in order to determine whether the corresponding transition exists in the system (family) represented by θ . Formally, for the comparison we introduce the following relation between configuration vectors.

Definition 3.13 (Conformance of Configuration Vectors). *Let $\nu, \nu' \in \{L, R, ?\}^n$ be configuration vectors. We say that ν' conforms to ν , denoted by $\nu' \sqsubseteq \nu$, if*

$$\forall i \in \{1, \dots, n\} : ((\nu_i = L) \Rightarrow \nu'_i \in \{L, ?\}) \quad \wedge \quad ((\nu_i = R) \Rightarrow \nu'_i \in \{R, ?\})$$

We will use the \sqsubseteq relation in order to precisely express in what configurations a certain transition of a PF-LTS is present. Every (possibly incomplete) configuration θ conforms to a configuration label ν , if θ either requires the same concrete values L or R at positions where ν holds these concrete values, or if θ even requires less, i.e. holds the entry $?$ at such positions (Note that if $\theta_i = ?$ then the premises of both conjuncts are false, making both conjuncts and the entire formula of Definition 3.13 true). For entries where $\nu_i = ?$, an arbitrary entry θ_i is allowed. For example, the (incomplete) configuration $\theta = \langle RL?? \rangle$ conforms to the label $\nu = \langle ?LR? \rangle$. For the derivation of concrete products, only complete configurations are relevant. Since complete configurations do not contain any $?$ -entires anymore, Definition 3.13 reduces to the following form in such a case:

$$\forall i \in \{1, \dots, n\} : ((\nu_i = L) \Rightarrow (\nu'_i = L)) \quad \vee \quad ((\nu_i = R) \Rightarrow (\nu'_i = R))$$

For *complete* configurations θ , a configuration label ν characterizes the set of all those configurations θ , whose entries either match the entries of ν , or have the values R or L on positions where ν carries an entry $?$ (and hence does not require any particular variant). Thus, a transition $s \xrightarrow{\alpha, \nu} s'$ in a PF-LTS exists in all configurations θ which *conform* to the transition label ν , i.e. where $\theta \sqsubseteq \nu$. For example, the configuration label $\langle LR? \rangle$ characterizes the complete configurations $\theta_1 = \langle LRL \rangle$ and $\theta_2 = \langle LRR \rangle$, for which the α -transition $\xrightarrow{\alpha, \langle LR? \rangle}$ is present. Beside the conformance relation we also need another relation that represents a kind of refinement for configurations (labels).

Definition 3.14 (Concretization of a Configuration Vector). *Let $\nu, \nu' \in \{L, R, ?\}^n$ be configuration vectors. We say that ν' is a concretization (or more concrete) of ν , denoted by $\nu' \sqsubseteq \nu$, if*

$$\forall i \in \{1, \dots, n\} : (\nu_i = \nu'_i) \vee ((\nu_i = ?) \Rightarrow (\nu'_i \in \{L, R\}))$$

In contrast to \sqsubseteq , we use the \sqsubset relation to characterize the evolution of configuration labels when constructing the transition relation according to the SOS rules. Illustratively, a configuration label ν' is a concretization of a label ν , if ν' agrees exactly with ν except for some positions where ν has an $?$ -entry while ν' has a more concrete entry in $\{L, R\}$. Note that for *complete* configuration vectors $\nu' \in \{L, R\}^n$, and arbitrary configuration vectors $\nu \in \{L, R, ?\}^n$ we have $\nu' \sqsubseteq \nu \Leftrightarrow \nu' \sqsubset \nu$, as the construction of the corresponding truth table shows.

Based on this prerequisites we can now elaborate the unfolded semantics of a PF-CCS program, which is given as a PF-LTS. In such a PF-LTS the states are pairs consisting of a PF-CCS process expression and a configuration label that specifies the configurations under which this state was reached. Thus, we denote states as tuples (t, ν) consisting of a PF-CCS process term $t \in \mathcal{L}_{PFCCS}$ and a configuration vector $\nu \in \{R, L, ?\}^n$. The transition relation of the PF-LTS is defined by means of SOS rules. The corresponding SOS rules for PF-CCS are shown in Figure 3.4. Essentially, the PF-CCS SOS rules are similar to the original CCS SOS rules. However, the PF-CCS SOS rules are enriched with configuration vectors which are used to construct the configuration labels for the transitions, respectively. This allows to keep track of the choices for the variants operators when building up the PF-LTS.

Except for the SOS rules for the variants operator \oplus , the remaining rules do not influence the construction of the configuration labels of the transitions. They are basically similar to their CCS SOS counterparts and only adjusted in order to be capable of handling configuration vectors. For example, rules (3.19) and (3.20) express that the execution of an action—specified either directly by action-prefixing as in (3.20) or indirectly by a constant definition as in rule (3.19)—can be done independently from and without affecting the current configuration label ν . More precisely, any state $(\alpha.P, \nu)$ affords a transition labeled with the action α to a successor state (P, ν) in every possible configuration ν . Note that the configuration label ν is not modified in this rule.

Essential for the SOS rules of the unfolded semantics is the treatment of the variants operator \oplus . Recall that it is a binary operator which allows to model a selection between two alternative processes where only one will be existing in the resulting system. Though looking similar to the ordinary CCS $+$ operator—which also models a kind of alternative choice—the variants operator has to be treated differently for two reasons: Firstly, when a configuration selection has been made, the same selection has to be taken when recursively revisiting the same \oplus_i -operator. Secondly, in

3. PF-CCS: Product Family CCS

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu}{C, \nu \xrightarrow{\alpha, \nu} P', \nu}, C \stackrel{\text{def}}{=} P \quad (\text{constant definition}) \quad (3.19)$$

$$\frac{}{\alpha.P, \nu \xrightarrow{\alpha, \nu} P, \nu}, \text{ for arbitrary } \nu \in \{R, L, ?\}^n \quad (\text{prefix}) \quad (3.20)$$

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu}{P + Q, \nu \xrightarrow{\alpha, \nu} P', \nu} \quad (\text{nondeterministic choice (1)}) \quad (3.21)$$

$$\frac{Q, \nu \xrightarrow{\alpha, \nu} Q', \nu}{P + Q, \nu \xrightarrow{\alpha, \nu} Q', \nu} \quad (\text{nondeterministic choice (2)}) \quad (3.22)$$

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha, \nu} (P' \parallel Q), \nu} \quad (\text{parallel composition (1)}) \quad (3.23)$$

$$\frac{Q, \nu \xrightarrow{\alpha, \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha, \nu} (P \parallel Q'), \nu} \quad (\text{parallel composition (2)}) \quad (3.24)$$

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu \quad Q, \nu \xrightarrow{\bar{\alpha}, \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\tau, \nu} (P' \parallel Q'), \nu} \quad (\text{parallel composition (3)}) \quad (3.25)$$

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu}{P[f], \nu \xrightarrow{f(\alpha), \nu} P'[f], \nu} \quad (\text{relabeling}) \quad (3.26)$$

$$\frac{P, \nu \xrightarrow{\alpha, \nu} P', \nu}{(P \setminus L), \nu \xrightarrow{\alpha, \nu} (P' \setminus L), \nu}, \alpha, \bar{\alpha} \notin L \quad (\text{restriction}) \quad (3.27)$$

(a) PF-CCS SOS rules for the unfolded semantics, except of the variants operator \oplus .

$$\frac{P, \nu|_{i/L} \xrightarrow{\alpha, \nu'|_{i/L}} P', \nu'|_{i/L}}{P \oplus_i Q, \nu \xrightarrow{\alpha, \nu'|_{i/L}} P', \nu'|_{i/L}}, \nu_i \neq R, \nu' \sqsubset \nu \quad (\text{configuration selection (1)}) \quad (3.28)$$

$$\frac{Q, \nu|_{i/R} \xrightarrow{\alpha, \nu'|_{i/R}} Q', \nu'|_{i/R}}{P \oplus_i Q, \nu \xrightarrow{\alpha, \nu'|_{i/R}} Q', \nu'|_{i/R}}, \nu_i \neq L, \nu' \sqsubset \nu \quad (\text{configuration selection (2)}) \quad (3.29)$$

(b) PF-CCS SOS rules for the treatment of the variants operator \oplus .

Figure 3.4.: Complete set of PF-CCS SOS rules for the unfolded semantics.

order to facilitate further reasoning about the configurations, e.g. by model checking, the choice has to be “made visible” within the transition relations. These two issues are captured by the two SOS rules for the variants operator \oplus shown in Figure 3.4b.

We illustrate their meaning informally taken the example of the first configuration selection rule (Equation 3.28) which corresponds to the selection of the left variant of the variants operator i . Recall that (1) ν_i yields the i^{th} element of the vector ν , and (2) $\nu|_{i/x}$ represents the updated vector ν in which the entry at the i^{th} position is replaced by the value $x \in \{R, L\}$, while all other entries keep their initial values. The SOS rule 3.28 states that from a state $(P \oplus_i Q, \nu)$, which represents the parse term $P \oplus_i Q$ and which was reached in configuration ν , we can perform an α transition to a state (P', ν') if (i) we decide to select the left variant (represented by the act of updating the initial configuration label ν to $\nu'|_{i/L}$ and labeling the current transition with this modified configuration vector), and (ii) the left variant P actually affords an α successor P' in the initial configuration ν , and (iii) the right variant has not been chosen for this variation point so far in a preceding recursive parsing pass.

The last requirement (iii) is realized by the side condition $\nu_i \neq R$ of rule 3.28. This side condition guarantees that also in recursive process definitions the SOS rule only allows to derive transitions which conform to the configuration choices taken in previous recursive (parsing) passes of the corresponding equation. Thus, it prevents the derivation of an α transition labeled with a configuration label where $\nu_i = L$ from a state which lies on a path on which the opposite variant R was already chosen for the respective variation point i . Consider for example the recursive program

$$\begin{aligned} P &\stackrel{\text{def}}{=} \alpha.T \oplus_1 \beta.T \\ T &\stackrel{\text{def}}{=} \gamma.P \end{aligned}$$

If we chose one variant, say the right variant $\beta.T$, the intention is that in any subsequent recursive pass of P we cannot undo a preceding configuration choice R and suddenly select the left variant $\alpha.T$. The SOS rule 3.28 guarantees that the resulting transition system is constructed in a way that respects this property.

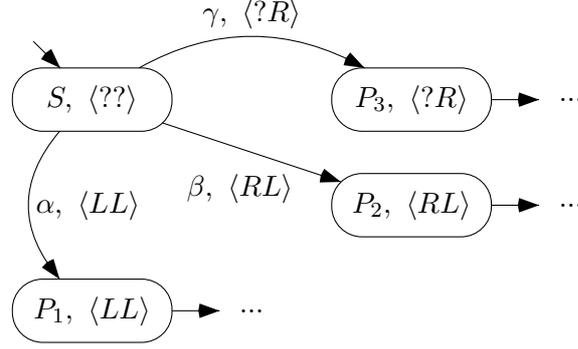
Together, these SOS rules allow to define the unfolded semantics of a product family specified in PF-CCS. As a prerequisite the variants operators in the respective PF-CCS program have to be numbered. However, for any well-formed PF-CCS program this can be done easily as we have motivated in the preceding section.

Definition 3.15 (Unfolded Semantics of a PF-CCS Program). *Let $Prog$ be a well-formed PF-CCS program $Prog = (\mathcal{E}, P_1)$ with n variants operators. We define the unfolded semantics of $Prog$, denoted by*

$$\llbracket Prog \rrbracket_{UF}$$

as the PF-LTS \mathcal{T} obtained by applying the SOS rules (3.19)-(3.29) to the main process identifier P_1 with an initial configuration label $\nu = \langle ?^n \rangle$. The start state of \mathcal{T} is $(P_1, \langle ?^n \rangle)$.

3. PF-CCS: Product Family CCS



(a) Initial part of the PF-LTS.

$$\begin{array}{c}
 \frac{\alpha.P_1, \langle LL \rangle \xrightarrow{\alpha, \langle LL \rangle} P_1, \langle LL \rangle}{\alpha.P_1 \oplus_1 \beta.P_2, \langle ?L \rangle \xrightarrow{\alpha, \langle LL \rangle} P_1, \langle LL \rangle} \\
 \frac{\alpha.P_1 \oplus_1 \beta.P_2, \langle ?L \rangle \xrightarrow{\alpha, \langle LL \rangle} P_1, \langle LL \rangle}{(\alpha.P_1 \oplus_1 \beta.P_2) \oplus_2 \gamma.P_3, \langle ?? \rangle \xrightarrow{\alpha, \langle LL \rangle} P_1, \langle LL \rangle}
 \end{array}$$

(b) Deduction of the transition $\xrightarrow{\alpha, \langle LL \rangle}$.

Figure 3.5.: PF-LTS for $S \stackrel{def}{=} (\alpha.P_1 \oplus_1 \beta.P_2) \oplus_2 \gamma.P_3$ and the deduction of a transition.

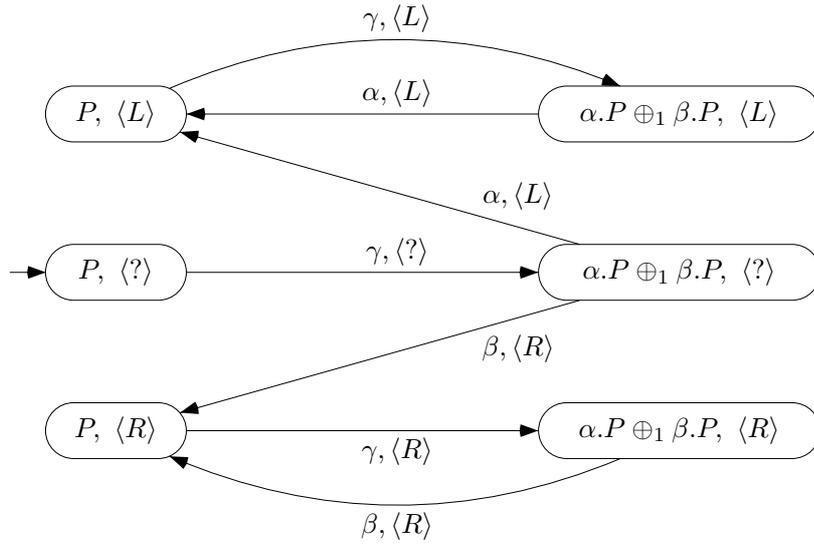
Let us demonstrate the unfolded semantics with two examples. As a first example, Figure 3.5a shows the PF-LTS when applying the configuration selection rules 3.28 and 3.29 to the (incomplete) PF-CCS program starting with the main process definition

$$S \stackrel{def}{=} (\alpha.P_1 \oplus_1 \beta.P_2) \oplus_2 \gamma.P_3$$

Since the presence of $\gamma.P_3$ in the final configuration only requires to select the right variant at the variation point \oplus_2 , the corresponding transition to state $(P_3, \langle ?R \rangle)$ only fixes the second entry of the configuration vector to the value R while leaving any choice for the first entry (?). In contrast to that, the selection of either $\alpha.P_1$ or $\beta.P_2$ requires to take two configuration decisions, reflected by the vectors $\langle LL \rangle$ and $\langle RL \rangle$ in the respective states $(P_1, \langle LL \rangle)$ and $(P_2, \langle RL \rangle)$. A corresponding deduction (applying twice Rule 3.28) for the selection of the variant $\alpha.P_1$ is given in Figure 3.5b. As the derivation demonstrates, the SOS semantics can require multiple configuration selections for deriving a single transition.

A second example illustrates the configuration selection rules for recursive process definitions. More specifically, Figure 3.6 shows the PF-LTS for the PF-CCS program

$$P \stackrel{def}{=} \gamma.(\alpha.P \oplus_1 \beta.P)$$


 Figure 3.6.: PF-LTS for the PF-CCS term $P \stackrel{\text{def}}{=} \gamma.(\alpha.P \oplus_1 \beta.P)$

Recall, that the state labels correspond to the process term together with the configuration under which they were reached. If the semantics would only depend on the current state's PF-CCS term (and not additionally on the configuration selected so far), the states at the left and the right column could not be told apart since the process term is the same for all three states in one column. But since the unfolded semantics keeps track of which configuration was chosen so far, identical PF-CCS terms yield different states in the PF-LTS, if the terms are parsed under different configurations. More precisely, this means that for example in the state $(\alpha.P \oplus_1 \beta.P, \langle L \rangle)$ the semantics does not allow to have an outgoing transition $\xrightarrow{\beta, \langle R \rangle}$ since the dual configuration $\langle L \rangle$ has already been selected for the configuration label of this state.

Given a PF-LTS (which we assume to represent an entire product family), we can derive the respective transitions systems which represent sub-families by means of projecting to a given configuration. Thereby, the projection yields a PF-LTS which comprises only those transitions where the configuration θ conforms to the respective configuration label. All other transitions, i.e. transitions whose configuration labels contain at least one contradictory entry, are discarded by the projection. Regarding states, the projected PF-LTS comprises only those states of the original PF-LTS which are reachable by the preserved transitions. Note that if the configuration θ is complete the projection always yields a full product which contains no variability anymore.

3. PF-CCS: Product Family CCS

Definition 3.16 (Projection of a PF-LTS (Unfolded Semantics)). *Let $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, \sigma)$ be a PF-LTS, and $\theta \in \{R, L, ?\}^n$ be a fitting configuration. The projection $\Pi_\theta(\mathcal{T})$ of \mathcal{T} according to θ is defined as the PF-LTS*

$$\Pi_\theta(\mathcal{T}) = (\mathcal{S}_\theta, \mathcal{A}, \rightarrow_\theta, \sigma)$$

where

- $\rightarrow_\theta = \{s \xrightarrow{\alpha, \nu} s' \in \rightarrow : \theta \sqsubseteq \nu\}$ is the relation which comprises exactly those transitions of \rightarrow which are allowed according to the configuration θ (Recall that \sqsubseteq represents the conformance relation as defined in Definition 3.13),
- $\mathcal{S}_\theta = \{(P, \nu) \in \mathcal{S} : \theta \sqsubseteq \nu\}$, i.e. $\mathcal{S}_\theta \subseteq \mathcal{S}$ is the set of all states which are reachable from σ with respect to the transition relation \rightarrow_θ .
- \mathcal{A} is a set of communication actions,
- and $\sigma \in \mathcal{S}$ is the start state.

Depending on the configuration, the concept of projection allows us to derive different kinds of transition systems: If the configuration is not complete the projection yields a PF-LTS which represents a sub-family that in general still contains variability. On the contrary, if the configuration is complete, the projection yields a PF-LTS which represents exactly the behavior of a single product. Since a complete configuration selects a variant for every occurring variation point, the projection according to a complete configuration only leaves those transitions (and respective states) in the resulting system for which $\nu \sqsubseteq \theta$, i.e. which exist exactly in this individual system.

In order to illustrate the concept of deriving products by projection consider the program *Prog* which is given by the following equations:

$$\begin{aligned} P &\stackrel{def}{=} \alpha.P_1 \oplus_1 \beta.P_2 \\ P_1 &\stackrel{def}{=} \alpha.P \oplus_2 \beta.P_2 \\ P_2 &\stackrel{def}{=} \gamma.P \end{aligned} \tag{3.30}$$

The PF-LTS representing the unfolded semantics of *Prog* is shown in Figure 3.7. The projections of the PF-LTS according to the four possible *complete* configurations are indicated by the line styles of the transitions:

- In configuration $\langle LL \rangle$ only the dotted,
- in configuration $\langle LR \rangle$ only the dashed
- in configuration $\langle RL \rangle$ only the disproportionately dashed,
- and in configuration $\langle RR \rangle$ only the solid transitions exist.

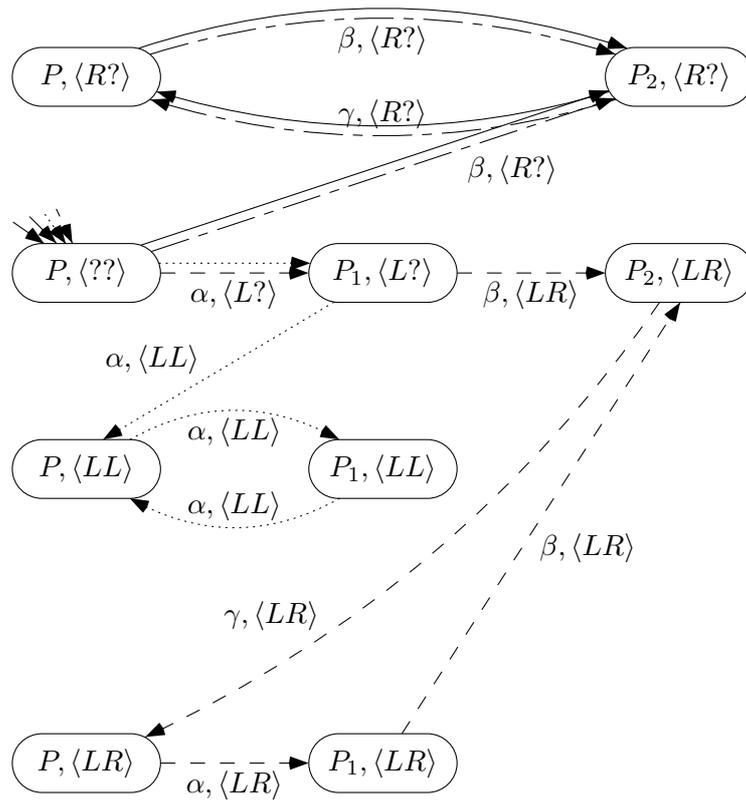


Figure 3.7.: The PF-LTS representing the unfolded semantics of process P defined in Equation 3.30 (on Page 132). The line styles of the transitions illustrate the projections to the four possible (complete) configurations: in a single configuration only the transitions with the same style are present.

3. PF-CCS: Product Family CCS

The corresponding states which exist in each single projection are those ones reachable under the respective subset of transitions (with the same line style), however, they are not explicitly marked in Figure 3.7. For example, the projection to configuration $\langle LL \rangle$ comprises the states $(P, \langle ?? \rangle)$, $(P_1, \langle L? \rangle)$, $(P, \langle LL \rangle)$, and $(P_1, \langle LL \rangle)$.

Let us now elaborate on the correctness of the unfolded semantics. The question is whether the PF-LTS constructed by the unfolded semantics indeed corresponds to the same set of products as specified by the flat semantics. More precisely, does every transition system which we can construct for any (complete) configuration θ from the unfolded semantics coincide with its corresponding counterpart which we obtain for the same configuration θ from the flat semantics? As we will show in the following, the answer is *yes, modulo bisimulation*.

To motivate this more clearly consider Figure 3.8. It shows the flat semantics of the PF-CCS program *Prog* specified in Equation 3.30, i.e. for all possible complete configurations θ it shows the corresponding transition systems which result from applying the CCS semantics to the respectively reduced PF-CCS program $\text{config}(\text{Prog}, \theta)$, which is shown next to each transition system. If we compare these transition systems with the corresponding transitions systems (for the same configuration) in Figure 3.7, we can see easily that they are not isomorphic. However, they are bisimilar, as Theorem 3.1 (Page 135) will show. Since bisimulation is the “natural” kind of equivalence relation for synchronously communicating systems—and thus also for PF-CCS—the wider meaning of Theorem 3.1 is that we can actually use PF-CCS to specify product families, since we can be sure that the derivable products are equivalent to specifying the same systems as standalone CCS programs. More precisely, Theorem 3.1 states that the two ways of constructing a system—via the flat semantics and the unfolded semantics, respectively—always coincide (modulo bisimulation) for any complete configuration. Thus, for a given product family represented as a PF-CCS program *Prog* and a complete configuration θ , the projection $\Pi_\theta(\text{Prog})$ from the respective PF-LTS for *Prog* (constructed using the SOS rules 3.19-3.29.) and the corresponding configuration $\text{config}(\text{Prog}, \theta)$ of the actual PF-CCS term *Prog* in fact yield bisimilar transition systems.

Before we introduce Theorem 3.1, we recall the concept of bisimulation as introduced by Park [Par81]. Certainly, the idea of bisimulation directly applies to processes, too, since processes (process expressions) semantically correspond to states. Milner [Mil95] presents the same concept of (strong) bisimulation in terms of processes.

Definition 3.17 (Bisimulation). *Let s_1, s'_1, s_2, s'_2 be states, and $\alpha \in \mathcal{A}$ be an action. A binary relation \mathcal{B} over a set of states of one (or many) labeled transition system(s) is a bisimulation iff whenever $s_1 \mathcal{B} s_2$:*

- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \mathcal{B} s'_2$, and
- if $s_2 \xrightarrow{\alpha} s'_2$, then there is a transition $s_1 \xrightarrow{\alpha} s'_1$ such that $s'_2 \mathcal{B} s'_1$.

3.2. Semantics of a PF-CCS Program

For any pair of states $(s_1, s_2) \in \mathcal{B}$ we say that s_1 and s_2 are bisimilar and write

$$s_1 \approx s_2$$

Two transition systems Sys_1 and Sys_2 are called bisimilar, denoted by $Sys_1 \approx Sys_2$, if their initial states are bisimilar.

Technically, in order to relate the transition systems which result from the flat semantics and unfolded semantics, respectively, we have to adjust the concept of bisimulation to the particular structure of a PF-LTS. In contrast to an LTS (in the flat semantics) where the transitions are only labeled with single actions α , e.g. as in $s \xrightarrow{\alpha} s'$, the transitions in a PF-LTS (in the unfolded semantics) additionally are labeled with a configuration label ν , such as in $s \xrightarrow{\alpha, \nu} s'$. However, since we only consider PF-LTSs which result from projection according to a *complete* configuration, the configuration labels in such a PF-LTS are meaningless, since every complete configuration corresponds to a full product which contains no variability anymore. Thus, we can ignore the configuration labels in such a case for the examination of a bisimulation between an LTS and a PF-LTS.

Theorem 3.1 (Correctness of Unfolded Semantics). *Let $Prog = (\mathcal{E}, P)$ be a well-formed PF-CCS program with n variation points. For all fitting and complete configurations $\theta \in \{R, L\}^n$ the (transition) system obtained from $Prog$ with the flat semantics according to θ is bisimilar to the system obtained by applying the unfolded semantics and subsequently projecting to the identical configuration θ .*

$$\llbracket config(Prog, \theta) \rrbracket_{CCS} \approx \Pi_\theta(\llbracket Prog \rrbracket_{UF})$$

Proof. Let

$$T_U := (\mathcal{S}_U, \mathcal{A}, \rightarrow, \sigma_U) = \llbracket Prog \rrbracket_{UF}$$

be the PF-LTS representing the unfolded semantics of $Prog$. We denote the states of T_U as tuples $(Proc, \nu)$ consisting of a PF-CCS process term $Proc \in \mathcal{L}_{PFCCS}$ and a configuration label $\nu \in \{R, L, ?\}^n$. Further, let

$$T_F := (\mathcal{S}_F, \mathcal{A}, \rightsquigarrow, \sigma_F) = \llbracket config(Prog, \theta) \rrbracket_{CCS}$$

be the LTS representing the flat semantics of $Prog$ for the concrete configuration θ . Here, the states \mathcal{S}_F are only labeled with CCS process terms. We denote them as (P) , where $P \in \mathcal{L}_{CCS}$.

Proof Outline: For any complete configuration $\theta \in \{R, L\}^n$ we define a (bisimulation) relation \mathcal{B}_θ between the states of $\Pi_\theta(T_U)$ and T_F . The relation \mathcal{B}_θ comprises all states in $\Pi_\theta(T_U)$ and T_F , and basically relates states that have the same process labels. We show for each configuration θ , that for every pair of related states in the corresponding \mathcal{B}_θ , the states can exactly perform the actions of the other state

3. PF-CCS: Product Family CCS



(a) Flat Semantics for the configuration $\theta = \langle LR \rangle$.



(b) Flat Semantics for the configuration $\theta = \langle LL \rangle$.



(c) Flat Semantics for the configuration $\theta = \langle RL \rangle$.



(d) Flat Semantics for the configuration $\theta = \langle RR \rangle$.

Figure 3.8.: The respective transition systems of the four possible configurations which represent the flat semantics of the PF-CCS program $Prog$ defined by the Equations 3.30 (on Page 132). The CCS programs $config(Prog, \theta)$ from which these LTSs are constructed are shown next to each LTS, respectively.

3.2. Semantics of a PF-CCS Program

and vice versa, resulting always in a pair of successor states which are again in \mathcal{B}_θ . Since the performable transitions for every states are characterized by the applicable SOS rules, we do this technically by performing a case discrimination over the set of applicable SOS for the respective process terms.

Detailed Proof: For every θ we define a (bisimulation) relation $\mathcal{B}_\theta \subseteq \mathcal{S}_U \times \mathcal{S}_F$ between the states of $\Pi_\theta(T_U)$ and T_F . Recall, that per construction, the projected PF-LTS $\Pi_\theta(T_U)$ contains only those states and transitions whose configuration labels ν fulfill: $\theta \sqsubseteq \nu$. The relation \mathcal{B}_θ is defined in the following way:

1. Each state $(P) \in \mathcal{S}_F$ is related by \mathcal{B}_θ to all states (P, ν) of $\Pi_\theta(T_U)$ which have an identical process expressions P , respectively, i.e. $(P, \nu)\mathcal{B}_\theta(P)$ for all ν appearing in $\Pi_\theta(T_U)$. Note that this relates every state in \mathcal{S}_F with at least one state of $\Pi_\theta(T_U)$, since (i) $config(Prog, \theta)$ has only preserved those process expressions which appear in a chosen variant, and (ii) exactly for these process expressions the corresponding states are preserved by the projection to θ in $\Pi_\theta(T_U)$.
2. All states (P, ν) of $\Pi_\theta(T_U)$ which are not related in step 1 must have a process expression P which contains a subterm $V_1 \oplus_n V_2$. Each of these states (P, ν) will be related by \mathcal{B}_θ to the state $(R) \in \mathcal{S}_F$, where R is the process term P in which every subterm $V_1 \oplus_n V_2$ is replaced
 - either by the left variant V_1 , if $\theta_n = L$,
 - or by the right variant V_2 , if $\theta_n = R$.

Thus, the variant which is chosen in R is the same as the variant which was preserved by $config$ in the program $config(Prog, \theta)$. In particular, this means that a state labeled with such a term R always exists in \mathcal{S}_F . Other states (P, ν) where $\theta \not\sqsubseteq \nu$ do not exist in the transition system $\Pi_\theta(T_U)$, as they have been discarded by the projection Π_θ .

This construction guarantees that for any complete configuration θ , the relation \mathcal{B}_θ comprises all states in $\Pi_\theta(T_U)$ as well as in \mathcal{S}_F , and relates .

We show now that for all pairs $((P, \nu), (Q)) \in \mathcal{B}_\theta$ both states afford the same transitions leading to a pair of successor states which is again in \mathcal{B}_θ . Since for each state of $\Pi_\theta(T_U)$ there exists exactly one state (Q) such that $(P, \nu)\mathcal{B}_\theta(Q)$, iterating over all states (P, ν) guarantees that all pairs in \mathcal{B}_θ are considered. To this end we perform a case discrimination over the kind of process expression P in (P, ν) , which reduces to a case discrimination over the set of SOS rules which are applicable to the process expression P .

3. PF-CCS: Product Family CCS

- Pairs $(P \oplus_n Q, \nu)\mathcal{B}_\theta(P)$ and $(P \oplus_n Q, \nu)\mathcal{B}_\theta(Q)$ (Configuration Selection Rules)
We show only the case where $\theta_i = L$ and thus consider the situation where $(P \oplus_n Q, \nu)\mathcal{B}_\theta(P)$. The dual case $(P \oplus_n Q, \nu)\mathcal{B}_\theta(Q)$ for the right variant is shown analogously.

– Transitions from $(P \oplus_n Q, \nu)$:

Since $\theta_i = L$, according to SOS rule 3.28 the only performable transition is $P \oplus_n Q, \nu \xrightarrow{\alpha, \nu'} P', \nu'$, where $\nu' \sqsubseteq \nu|_{i/L}$. This transition can only be performed if the premise $P, \nu' \xrightarrow{\alpha, \nu'} P', \nu'$ can be shown. However, if so, by SOS rule 3.11 the related state (P) can also perform a transition $(P) \xrightarrow{\alpha} P'$. Since the successor state (P') has an identical process term as the state (P', ν) , we have $(P', \nu)\mathcal{B}_\theta(P)$.

– Transitions from (P) :

The state (P) affords all transitions $P \xrightarrow{\alpha} P'$ which are derivable for the process P . Due to identical process expressions (P) , the state $(P \oplus_n Q, \nu)$ can simulate all these transitions by respective transitions $P \oplus_n Q, \nu \xrightarrow{\alpha, \nu'} P', \nu'$. This requires to select the left variant L for the variation point n . However, since the pair $(P \oplus_n Q, \nu)\mathcal{B}_\theta(P)$ is only related in configurations θ where $\theta_i = L$, this requirement is always fulfilled. Thus, both states perform the same set of transitions. In addition, by rule 3.28, for all successor states (P', ν) which can be reached from the left variant of $(P \oplus_n Q, \nu)$, the respective configuration label ν' fulfills $\nu'_i = L$ and $\nu' \sqsubseteq \nu$. This means that all successor states (P', ν) actually exist in $\Pi_\theta(T_U)$, and in particular that all resulting states are in \mathcal{B}_θ due to the equivalence of their process expressions P' .

- Pairs $(\alpha.P, \nu)\mathcal{B}_\theta(\alpha.P)$ (Action Prefixing Rule)

– Transitions from $(\alpha.P, \nu)$:

For every ν and for every $\alpha \in \mathcal{A}$, every state $(\alpha.P, \nu)$ can be perform a transition $\alpha.P, \nu \xrightarrow{\alpha, \nu} P, \nu$ by SOS rule 3.20 to a state (P, ν) . Since this is the only applicable SOS rule, no more outgoing transitions exist in $(\alpha.P, \nu)$. Per construction of \mathcal{B}_θ the state $(\alpha.P)$ is related to every one of these states $(\alpha.P, \nu)$, and affords the same α -transition by SOS rule 3.11 resulting in state (P) . Since both process expressions P are equal, per construction of \mathcal{B}_θ the pair $((P, \nu), (P))$ is again in \mathcal{B}_θ .

– Transitions from $(\alpha.P)$:

By SOS rule 3.11, the only possible transition is $\alpha.P \xrightarrow{\alpha} P$. No other transitions can be derived in this state. For every ν which exists in $\Pi_\theta(T_U)$, every state $(\alpha.P, \nu)$ can match this transition by SOS rule 3.20, i.e. $\alpha.P, \nu \xrightarrow{\alpha, \nu} P, \nu$, where all successor states (P, ν) are again in relation $(P, \nu)\mathcal{B}_\theta(P)$, respectively.

- The rules for the remaining cases, i.e. for the constant definition $P \stackrel{\text{def}}{=} Q$, and the process expressions $P+Q$, $P \parallel Q$, $P[f]$, and $P \setminus L$ are shown in an analogue way. Each proof is based on the facts that the states which are related by \mathcal{B}_θ for these SOS rules all have identical process expressions and thus equivalent SOS rules apply.

This shows that \mathcal{B}_θ is indeed a bisimulation for every θ . Since per construction of \mathcal{B} we always have $((\sigma_U, \langle ?^n \rangle), (\sigma_F)) \in \mathcal{B}_\theta$ for every configuration θ and every pair of start processes σ_U and σ_F , the systems $\Pi_\theta(\llbracket Prog \rrbracket_{UF})$ and $\llbracket config(Prog, \theta) \rrbracket_{CCS}$ are bisimilar for all PF-CCS programs $Prog$ and all configurations θ . \square

Theorem 3.1 shows what we intuitively expect from the introduced SOS rules: Regarding the resulting (transitions system of the) product there is no difference whether we (i) transform the PF-CCS program of a product line according to a given configuration and apply the original CCS SOS rules afterwards, (ii) or whether we generate a PF-LTS for the entire product line from a PF-CCS program and retrieve the transition system of an individual system by means of projection according to the same configuration. Thus, applying a configuration can be done by either removing the unselected variants from the PF-CCS term or by removing the transitions which are not selected from the corresponding PF-LTS. For all possible configurations, both ways will yield equal (bisimilar) transition systems. But most importantly, we can actually use a PF-CCS program to represent the same behavior as it can be represented by a family of corresponding CCS programs. However, in PF-CCS we have the commonalities explicitly expressed.

While the unfolded semantics is easily understood and does indeed represent the behavior of a PF-CCS program within a single transition system, a transition system constructed according to the unfolded semantics usually contains many states with identical process terms that just differ in the respective configuration label in the states, as Figures 3.6 and 3.7 demonstrate. This leads one to suspect that the unfolded semantics yields non-compact transition systems, which is a major disadvantage for model checking techniques which operate on such a PF-LTS. We will investigate this question in the following section, and in fact, we will introduce another semantics, the configured-transitions semantics, which is based on the unfolded semantics, yet yields smaller transition systems.

3.2.3. Configured-transitions Semantics

In the following, we give a further semantics for a PF-CCS program which yields a smaller transition system and, at the same time, provides the basis for model checking the entire product line as described in Chapter 4. The idea—and the

3. PF-CCS: Product Family CCS

main difference to the unfolded semantics—is to identify states that have the same PF-CCS process term but only differ in the corresponding configuration label.

The configured-transitions semantics is also given in terms of a labeled transition system which represents the entire product family. We refer to it as a PF-LTS, too. However, in contrast to the PF-LTS used for the unfolded semantics, in a PF-LTS for the configured-transitions semantics the states and transitions are labeled differently: States now correspond to process terms only (no longer to tuples of process terms/configuration label), and transitions are labeled with a pair consisting of an action and a *set* of configurations. In addition—in order to set the course for a PF-LTS as a structure suitable for model checking—we represent the transition relation no longer as a single relation but rather as a set of relations $\{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}$, where each \mathcal{R}_α represents the α -transitions in the LTS. The following definition makes the structure of a PF-LTS for the configured-transitions semantics precise.

Definition 3.18 (PF-LTS for the Configured-Transitions Semantics). *A product line labeled transition system (PF-LTS) representing the configured-transitions semantics for a PF-CCS program with n variation points is a tuple $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, \sigma)$, where*

- \mathcal{S} is a (possibly infinite) set of states,
- \mathcal{A} is a finite set of communication actions,
- $\{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}$ is a family of (transition) relations $\mathcal{R}_\alpha \subseteq \mathcal{S} \times \mathcal{P}(\{R, L, ?\}^n) \times \mathcal{S}$.
- and $\sigma \in \mathcal{S}$ is the start state.

We call a PF-LTS also a *multi-valued modal (labeled) transition system*. With $\mathcal{T}.s$ we denote the state s of the PF-LTS \mathcal{T} .

The transition relation of the configured-transitions semantics is based on the transition relation of the unfolded semantics. In order to construct the PF-LTS of the configured-transitions semantics we introduce a family of transition relations

$$\Longrightarrow_\alpha \subseteq \mathcal{L}_{PFCCS} \times \mathcal{P}(\{R, L, ?\}^n) \times \mathcal{L}_{PFCCS}$$

for each action $\alpha \in \mathcal{A}$. For convenience of notation, if $(P, V, P') \in \Longrightarrow_\alpha$ we usually write $P \xrightarrow{\alpha, V} P'$ instead. The relation \Longrightarrow_α associates all PF-CCS process expressions $P, P' \in \mathcal{L}_{PFCCS}$ for which a transition exists in the corresponding PF-LTS of the unfolded semantics in the following way

$$P \xrightarrow{\alpha, V} P' \quad \text{where} \quad V = \{\nu \mid \exists \nu' : P, \nu' \xrightarrow{\alpha, \nu'} P', \nu\}$$

where $\nu, \nu' \in \{L, R, ?\}^n$ and $\xrightarrow{\alpha, \nu}$ is the relation defined in the previous section by the PF-CCS SOS rules defined in Figure 3.4. Intuitively, the relation collects for all

3.2. Semantics of a PF-CCS Program

pairs of states labeled with process expressions P and P' which are connected in the unfolded semantics each possible configuration ν and relates the corresponding two states in the configured-transitions semantics with a transition labeled with the set V containing all collected ν . A transition $P \xrightarrow{\alpha, V} P'$ where $V = \emptyset$ means that the corresponding states are not connected by an α transition \Longrightarrow_α . In such a case we simply omit the respective transition and do not explicitly write it anymore.

Since in the configured-transitions semantics states correspond to process expressions, the relations \Longrightarrow_α allow to construct the labeled transition system for the configured-transitions semantics. We define the configured-transitions semantics of a PF-CCS program as a labeled transition system in the following way.

Definition 3.19 (Configured-transitions Semantics of a PF-CCS Program). *Let $Prog$ be a well-formed PF-CCS program $Prog = (\mathcal{E}, P_1)$ with n variants operators. The configured-transitions semantics of $Prog$, denoted by*

$$\llbracket Prog \rrbracket_{CT}$$

is defined to be the PF-LTS $(\mathcal{S}, \mathcal{A}, \{\Longrightarrow_\alpha \mid \alpha \in \mathcal{A}\}, (P_1))$ consisting of all states \mathcal{S} reachable from the main process identifier P_1 w. r. t. the relation family \Longrightarrow_α .

A PF-LTS for the configured-transitions semantics contains exactly the same process expressions which also appear in the PF-LTS which is constructed according to the unfolded semantics. However, in the configured-transitions semantics each state (P) with process expression P appears only once.

We demonstrate how a PF-LTS for the configured-transitions semantics can be constructed with a concrete example. Recall the program which we have introduced in Equation System 3.30:

$$\begin{aligned} P &\stackrel{def}{=} \alpha.P_1 \oplus_1 \beta.P_2 \\ P_1 &\stackrel{def}{=} \alpha.P \oplus_2 \beta.P_2 \\ P_2 &\stackrel{def}{=} \gamma.P \end{aligned}$$

Its unfolded semantics is shown in Figure 3.9a. As the definition of the configured-transitions semantics suggests, one can gain the PF-LTS of the configured-transitions semantics from the PF-LTS of the unfolded semantics in a systematic way. The entire Figure 3.9 illustrates the practical construction of the PF-LTS representing the configured-transitions semantics (cf. Figure 3.9b) from the PF-LTS of the unfolded semantics (cf. Figure 3.9a). At first, we identify all process terms that appear in the states of the unfolded semantics. For Figure 3.9a these process terms are P, P_1 and P_2 (where states with identical process terms are shown in the same columns). These process terms form the corresponding states (P), (P_1) and (P_2) in the configured-transitions semantics. Then, one proceeds according to Definition 3.19 by collecting for all connected pairs of states and all actions the

3. PF-CCS: Product Family CCS

respective configurations in which the unfolded semantics has a corresponding transition. In the configured-transitions semantics this set of configurations together with the respective action then builds a new transition between the considered pair of states. For example, for the pair of states (P_2, P) (in the configured-transitions semantics) we observe two transitions in the PF-LTS of the unfolded semantics, namely: $P_2, \langle R? \rangle \xrightarrow{\gamma, \langle R? \rangle} P, \langle R? \rangle$ and $P_2, \langle LR \rangle \xrightarrow{\gamma, \langle LR \rangle} P, \langle LR \rangle$. Consequently, the PF-LTS of the configured-transitions semantics shown in Figure 3.9b contains the transition $P_2 \xrightarrow{\gamma, \{\langle R? \rangle, \langle LR \rangle\}} P$.

We see another example of the configured-transitions semantics in Figure 3.10, which shows the transition system for the program $P \stackrel{\text{def}}{=} \gamma.(\alpha.P \oplus_1 \beta.P)$. A comparison with Figure 3.6, which shows the unfolded semantics for the same program, suggests that the configured-transitions semantics can indeed yield smaller transition systems. The first example given in Figure 3.9 strengthens this suggestion. By inspecting the way in which the transition relation of the configured-transitions semantics is constructed, it is easy to see that a PF-LTS comprises always fewer (or at most equally many) states than its unfolded semantics counterpart, due to the fact that each process expressions appears only once as a state in the PF-LTS of the configured-transitions semantics, while it can correspond to many states in the PF-LTS of the unfolded semantics. In particular, for PF-CCS programs with a high degree of recursive definitions, the configured-transitions semantics will in general yield a much smaller PF-LTS than the unfolded semantics.

From a PF-LTS representing the configured-transitions semantics we can derive the transitions systems of sub-families or concrete products by means of projection. The projection of a PF-LTS according to the configured-transitions semantics is similar to the projection in the unfolded semantics (cf. Definition 3.16). In fact, it simply extends Definition 3.16 to handle transitions which are labeled with a *set* of configuration labels. For every transition $\xrightarrow{\alpha, L}$ the projection according to a configuration θ discards all those configuration vectors in L which do not conform to θ . This can yield a possibly empty set L of configuration labels, which represents the situation that the respective transition does not exist at all. Due to the similarity to the projection specified in Definition 3.16 we use the same symbol Π also for the projection of PF-LTSs constructed according to the configured-transitions semantics.

Definition 3.20 (Projection of a PF-LTS (Configured-transitions Semantics)). *Let $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \{\Longrightarrow_{\alpha} \mid \alpha \in \mathcal{A}\}, \sigma)$ be a PF-LTS (as defined in Definition 3.18), and $\theta \in \{R, L, ?\}^n$ be a fitting configuration. The projection $\Pi_{\theta}(\mathcal{T})$ of \mathcal{T} according to θ is defined as the PF-LTS*

$$\Pi_{\theta}(\mathcal{T}) = (\mathcal{S}_{\theta}, \mathcal{A}, \{\Longrightarrow_{\alpha}^{\theta} \mid \alpha \in \mathcal{A}\}, \sigma)$$

where

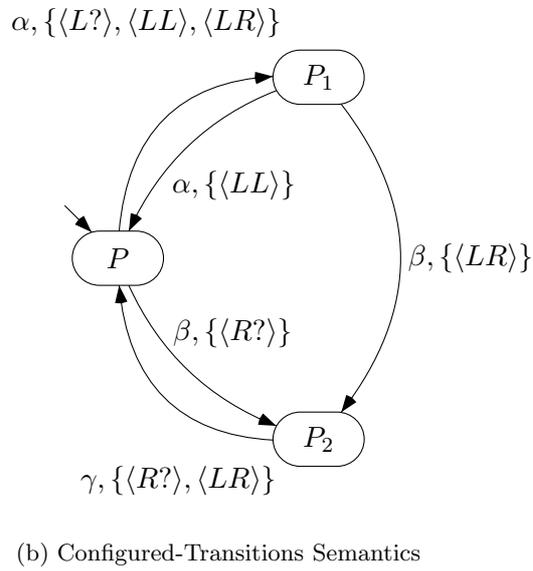
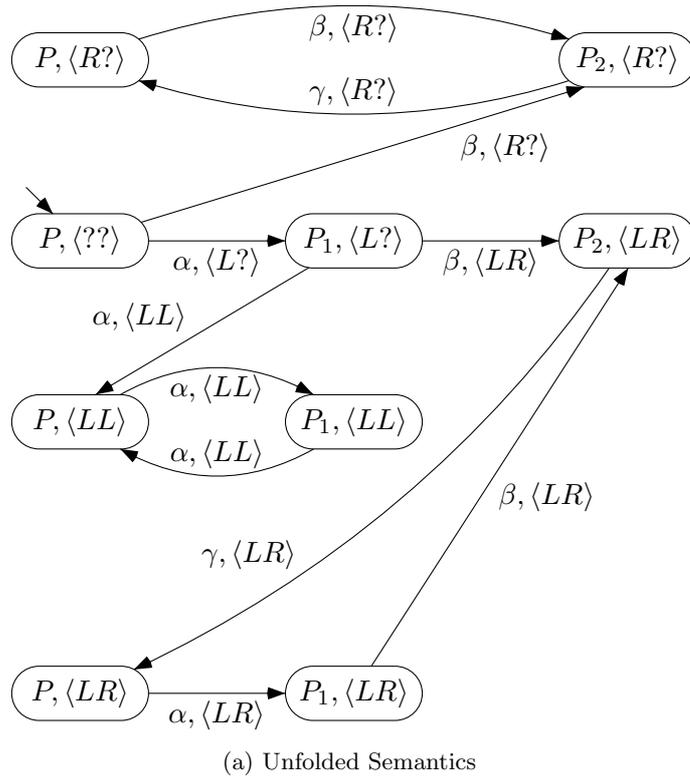


Figure 3.9.: The unfolded semantics and the configured-transitions semantics of the same PF-CCS program defined in Equations 3.30.

3. PF-CCS: Product Family CCS

- $\Longrightarrow_{\alpha}^{\theta} = \{s \xrightarrow{\alpha, L'} s' : (s \xrightarrow{\alpha, L} s' \in \Longrightarrow_{\alpha}) \wedge (L' = \{\nu \in L : \theta \sqsubseteq \nu\})\}$
- $\mathcal{S}_{\theta} \subseteq \mathcal{S}$ is the set of all states which are reachable from σ with respect to the transition relation $\Longrightarrow_{\alpha}^{\theta}$.
- \mathcal{A} is a set of communication actions,
- and $\sigma \in \mathcal{S}$ is the start state.

Note that the start state σ is always preserved by any projection, since it is labeled with the least concrete configuration label $\langle ? \rangle^n$. Further, recall that for projections according to *complete* configurations θ , the conformance relation \sqsubseteq is equal to the concretization relation \sqsubset . As a consequence of the projection some of the original states of a PF-LTS \mathcal{T} are discarded and do not exist any more in the projected system $\Pi_{\theta}(\mathcal{T})$. All preserved states correspond to their original counterparts.

The concept of projection can easily be extended to sets of configurations. As we will see in the upcoming Chapter 5.3, this is in particular useful to reason about the commonalities of alternative processes. However, as we will deal with such topics not now, we defer the definition to Chapter 5.3, where we will introduce a general form of projection in Definition 5.1 (Page 215).

The unfolded semantics and the configured-transitions semantics have a fundamental difference: For any PF-CCS program *Prog*, every path in the unfolded semantics $\llbracket Prog \rrbracket_{UF}$ corresponds to the execution of one path of a concrete product of the family, i.e. for every trace t which we can construct in the PF-LTS of the unfolded semantics (ignoring the configuration labels), there is at least one concrete product in the flat semantics $\llbracket Prog \rrbracket_{Flat}$ which also provides the trace t . This means that all transitions on any path in the unfolded semantics correspond to realistic traces which are actually realizable for at least one concrete product. This property no longer holds for the paths and traces in the PF-LTS of the configured-transitions semantics $\llbracket Prog \rrbracket_{CT}$. For example, the trace $\gamma\alpha\gamma\beta$ in the system shown in Figure 3.10 does not exist in any of the transition systems of $\llbracket P \stackrel{def}{=} c.(a.P \oplus b.P) \rrbracket_{Flat}$. However, the interesting property of the configured-transitions semantics is that for every

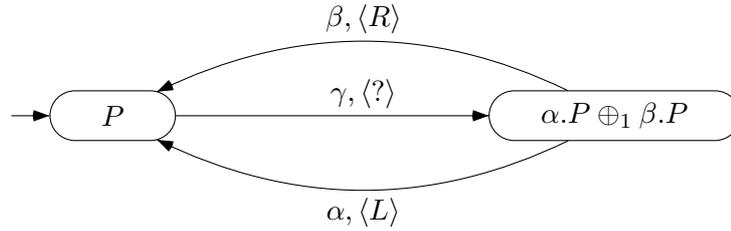


Figure 3.10.: Configured-transitions Semantics for the simple recursive PF-CCS process $P \stackrel{def}{=} \gamma.(\alpha.P \oplus_1 \beta.P)$. It's unfolded semantics is shown in Figure 3.6.

configuration vector θ , the projection of $\Pi_\theta(\llbracket Prog \rrbracket_{CT})$ yields the same transition system (modulo bisimulation) as the one obtained when configuring $Prog$ w.r.t. a fully-configured configuration θ according to the flat semantics.

Theorem 3.2 (Correctness of Configured-Transitions Semantics). *Let $Prog$ be a PF-CCS program with n variation points. For any fitting, complete configuration vector $\theta \in \{L, R\}^n$ we have*

$$\llbracket config(Prog, \theta) \rrbracket_{CCS} \approx \Pi_\theta(\llbracket Prog \rrbracket_{CT})$$

Here, the resulting transition systems have different kinds of transition labels. Similarly to Theorem 3.1, for the bisimulation relation we will only consider the actions $\alpha \in \mathcal{A}$ of the corresponding transition labels.

The proof uses the fact that the configured-transitions semantics is based on the unfolded semantics and establishes a bisimulation relation between those two PF-LTSs. Since the bisimulation is an equivalence relation, this implies also the bisimulation to the transition system representing the flat semantics.

Proof. Let

$$T_U := (\mathcal{S}_U, \mathcal{A}, \rightarrow, \sigma_U) = \llbracket Prog \rrbracket_{UF}$$

be the PF-LTS representing the unfolded semantics of $Prog$. Further, let

$$T_C := (\mathcal{S}_C, \mathcal{A}, \{\Longrightarrow_\alpha \mid \alpha \in \mathcal{A}\}, \sigma_C) = \llbracket Prog \rrbracket_{CT}$$

be the PF-LTS representing the configured-transitions semantics of $Prog$.

We show that

$$\forall \theta \in \{R, L\}^n : \Pi_\theta(T_U) \approx \Pi_\theta(T_C)$$

Due to the transitivity of the bisimulation relation and Theorem 3.1 we then can conclude that $\llbracket config(Prog, \theta) \rrbracket_{CCS} \approx \Pi_\theta(\llbracket Prog \rrbracket_{CT})$.

We proceed similarly to the proof of Theorem 3.1 and define a (bisimulation) relation

$$\mathcal{B}_\theta \subseteq \mathcal{S}_C \times \mathcal{S}_U$$

between the states of the transition systems T_C and T_U . It relates every state $(P) \in \mathcal{S}_C$ with all states $(P, v) \in \mathcal{S}_U$ which have identical process expressions P . Due to the way in which T_C is constructed from T_U , both transition systems can simulate each other for any complete configuration $\theta \in \{R, L\}^n$.

3. PF-CCS: Product Family CCS

- T_C simulates every transition of T_U :

Due to the way in which T_C is constructed, for every transition $P, \nu \xrightarrow{\alpha, \nu'} P', \nu'$ from states (P, ν) to (P', ν') in T_U , there exists a corresponding transition $P \xrightarrow{\alpha, V} P'$ in T_C which contains the label $\nu' \in V$, and where the involved states are bisimilar, i.e. $(P)\mathcal{B}_\theta(P, \nu)$ and $(P')\mathcal{B}_\theta(P', \nu')$. Further, for every complete configuration θ where the projection $\Pi_\theta(T_U)$ preserves a transition, i.e. where $\theta \sqsubseteq \nu'$, the same configuration vector ν' is also preserved by the projection $\Pi_\theta(T_C)$ and thus appears in the corresponding set of configuration labels V . Thus, for every configuration and every transition of T_U , a corresponding transition is also present in T_C .

- T_U simulates every transition of T_C :

For every configuration label $\nu' \in V$ in a transition $P \xrightarrow{\alpha, V} P'$ in T_C there exists a corresponding transition $P, \nu \xrightarrow{\alpha, \nu'} P', \nu'$ due to the construction principle of T_C . For any transition $P \xrightarrow{\alpha, V} P'$, for every $\nu' \in V$ which is not discarded by the projection to θ a corresponding transition $P, \nu \xrightarrow{\alpha, \nu'} P', \nu'$ is also preserved in T_U since in both cases the same property $\theta \sqsubseteq \nu'$ holds.

Thus, each transition system can simulate the other one. □

Theorem 3.2 guarantees that all products which we can derive from the configured-transitions semantics match the products from the standard CCS semantics, i.e. the products which we expect if we model all systems in a stand-alone fashion not as part of a product family. In fact, since the bisimulation is an equivalence relation, due to the transitivity we can combine both theorems to the following result.

Theorem 3.3 (Equivalence of Unfolded Semantics and Configured-Transitions Semantics). *For any fitting, complete configuration $\theta \in \{L, R\}^n$ we have*

$$\llbracket \text{config}(Prog, \theta) \rrbracket_{CCS} \approx \Pi_\theta(\llbracket Prog \rrbracket_{UF}) \approx \Pi_\theta(\llbracket Prog \rrbracket_{CT})$$

Proof. The proof follows directly from Theorem 3.1, Theorem 3.2, and the transitivity of the bisimulation relation \approx . □

This “equivalence” allows us to interpret a PF-CCS program with either of the two semantics. However, in the following we will use the configured-transitions semantics as the standard way of understanding the PF-CCS specification of an entire product family. In fact, we have introduced the unfolded semantics only as an intermediate step that motivates and leads to the configured-transitions semantics. But since the configured-transitions semantics was especially designed to be a suitable

structure for model checking we will use the configured-transitions semantics as the default way of constructing a PF-LTS for the remainder of this thesis.

If we interpret a PF-CCS program with the configured-transitions semantics we get a transition system where each transition is labeled with a set $V \in \mathcal{P}(\{L, R, ?\}^n)$ of configuration labels. In particular, the configuration labels $\nu \in V$ themselves must not be complete, i.e. they can still contain ?-entries. However, every configuration label which contains ?-entries represents a set of *complete* configuration vectors. Thus, for every transition $P \xrightarrow{\alpha, V} P'$ the set V of possibly incomplete configuration labels is equivalent to the following set of *complete* configuration labels $\nu' \in \{L, R\}^n$:

$$\{\nu' \in \{L, R\}^n \mid \exists \nu \in V : \nu' \sqsupseteq \nu\}$$

For example, the transition $P_2 \xrightarrow{\gamma, \{\langle R? \rangle, \langle LR \rangle\}} P$ (cf. Figure 3.9) is just a compact form of denoting the transition $P_2 \xrightarrow{\gamma, \{\langle RR \rangle, \langle RL \rangle, \langle LR \rangle\}} P$. Constructively, we obtain the latter version by replacing all ?-entries of any configuration label by the combinatorially possible configurations with L and R entries. This results in a set of complete configuration labels, only.

For the remainder of this work we use the non-compact form for any configuration labels set attached to a transition in the PF-LTS of the configured-transitions semantics. In particular, for applying model checking techniques a configuration label set consisting of complete configuration labels only, is necessary.

3.3. Design Decisions for PF-CCS

For the design of the PF-CCS framework we have made some concrete design decisions, e.g. the choice for a process algebra, the decision for synchronous communication and the associated decision for choosing bisimulation as the relevant equivalence relation, etc. In the following we discuss some of the major design decisions and in particular explain our motivation and the design drivers behind them.

Why is PF-CCS based on a process algebra?

PF-CCS allows to model the behavior of a set of systems as a software product family. As we have seen in Chapter 2, the product family concepts are independent of the concrete specification technique which we use to model the behavior or interaction with other systems or the environment. With PF-CCS, we have deliberately chosen a process algebraic approach for modeling the behavioral aspect. In particular, we have preferred a process algebra to other specification techniques, such as for example various kinds of automaton models [dAH01, LT89, Har87], Petri nets [Pet62], or specific logic based specification frameworks such as TLA [Lam94], which also allow the specification of behavior. The decision for a process algebra was motivated by the following factors:

3. PF-CCS: Product Family CCS

- The products of system families that we have in mind are systems which consist of many components that operate in parallel and interact with each other in a complex way. Typical examples are modern cars that consist of many ECUs that operate in parallel and interact by means of message passing. For capturing the interaction and the operational functionality of such systems process algebras are per se very suitable. In particular, in contrast to automaton model, a process algebra allows to denote an infinite behavior with an finite specification, where automaton models require to denote an infinite transition system, which is practically not feasible.
- Beside the mere specification of the behavior of such system families, we are also interested in the systematic restructuring of such a specification. The restructuring is based on rules which tell us how to transform our specification systematically into another representation while preserving the behavior of each derivable system. As we will see in the upcoming Chapter 5, the restructuring and the “calculation” of the greatest behavioral commonality only means to apply algebraic laws. Against this background, compared to other specification techniques for capturing the behavior, e.g. automata, a process algebra is a more natural choice since it already exhibits an algebraic nature and easily allows to define such restructuring laws.
- Beside the specification and restructuring aspect, another main driver of the PF-CCS framework is to provide a theoretical basis for the verification of behavioral properties in the context of a large variety of similar systems as we find them in a product family. Modern verification techniques, for example model checking [CGP99, BK08], for single systems usually operate on a representation of the system’s behavior as a labeled transition system. By providing an operational semantics for PF-CCS programs, any PF-CCS program can equivalently be represented by a corresponding labeled transition system. Thus, similarly to automata models which are directly usable for model checking techniques, PF-CCS specifications are also suitable for transition system based verification techniques, e.g. model checking, but at the same time fulfill the other properties which we require from a specification technique in our case.

In summary, a process algebra is a very suitable specification technique for our purpose since it combines all three aspects which are important for our purpose, i.e. for the specification, verification and restructuring of system families which we have in mind.

Is the choice for CCS as the underlying process algebra for PF-CCS compulsory?

For PF-CCS we have chosen CCS as the underlying basic process algebra, and especially not one of the other established process algebras like e.g. ACP [BK84] or CSP [Hoa85]. We have chosen CCS because among the prominent process algebras, CCS

is that one for which traditionally an operational semantics is provided, while for CSP usually denotational or axiomatic semantics prevail. Thus, it is more likely that someone who is interested in an operational specification of the system's behavior, is familiar with CCS rather than with CSP. However, CCS it is not compulsory for the realization of the software product family concepts: CCS is not better suited for the combination with product family concepts than any of the other prominent process algebras like ACP or CSP. In fact, we could have easily used CSP or ACP as the basis for PF-CCS, i.e. we could have extended CSP or ACP in the same way with product family concepts as we have actually done it with CCS. The axiomatization, i.e. the algebraic specification of the sort $\text{SPF } \alpha$ given in Chapter 2, is the formal basis that allows us to substantiate that ACP or CSP could have also been extended with software product family concepts resulting in an equivalently appropriate framework such as PF-CCS. Similarly to CCS, also the concepts of ACP and CSP are orthogonal to product family specific concepts, and thus can be combined easily, too.

We have seen that PF-CCS provides all the original operators of CCS—in a slightly adjusted kind. Every PF-CCS specification of a final product—containing no more variation points—only contains basic CCS operators. In this context, the parallel composition takes a special role since for the axiomatization it realizes the composition function and fulfills the respective axioms stated in Figure 2.11, e.g. the distributive law in connection with the variants operator, etc. Without giving an in-depth analysis here, we can easily see that the basic operators of ACP and CSP can be enriched with a variants operator in a similar way, where the various parallel composition functions of ACP (i.e. the so-called *free merge*, *communication merge*, and *left merge* operators) and CSP (i.e. the *interleaving* and *interface parallel* operators) can take the role of the composition function specified in the axiomatization in an correct way, too. Thus, product family extensions of ACP or CSP would fulfill the properties and laws of the axiomatization in the same way, as the current version of PF-CCS does. For the large variety of remaining process algebras, we can proceed with each one of these in the same way and check whether they can be extended with software product family concepts by showing the consistency of the respective extension with the axiomatization. Against this background, it becomes now also more comprehensible why we have provided an axiomatization in such a detail: The axiomatization specifies all requirements of a software product family in a way which allows check for potential models whether they fulfill the axioms and laws, or not.

Why synchronous message passing for PF-CCS?

With PF-CCS we have chosen synchronous communication, where common actions can only be performed by two parallel process if one process is willing to perform an action while at the same time the other process is actually waiting for the action to be performed, in the way of a handshake protocol. Similarly to the choice for CCS, also the choice for synchronous communication is not compulsory for PF-CCS,

3. *PF-CCS: Product Family CCS*

since the specification of the composition operation of the axiomatization does neither exclude nor require synchronous nor asynchronous communication. However, synchronous communication is closer to an implementation than asynchronous communication [Sch98]. Since with PF-CCS our focus is on describing behavior in a way which is as close to an implementation as possible, while still being platform independent, we have decided for synchronous communication. Furthermore, synchronous communication fits well with our choice for CCS, since originally CCS was also introduced with synchronous communication. It was only later when process algebras based on CCS/ACP with asynchronous communication emerged, cf. [BKT84, dKP92, Ros05].

We stress again that from the point of view of fundamental product family concepts, the kind of communication (synchronous or asynchronous) is not relevant. In particular, the axiomatization of a software product family does not require the composition operator to provide any specific kind of communication. It only requires the properties stated in the axioms, e.g. distributivity with the variants operator, etc.

3.4. **Practicability of PF-CCS**

PF-CCS is designed to be a conceptual specification framework that allows to specify the operational behavior of a family of systems, but that does not claim to be directly applicable for the practical development of families of large-scale, software-intensive systems which exist in the current industrial practice. In the following we discuss some aspects regarding the practicability of the PF-CCS framework for the practical specification of more complex, industry-relevant systems. As our focus is on establishing a conceptual framework, we have not realized these aspects in the current version of PF-CCS. Nevertheless, they are important steps to increase the practicability of PF-CCS. The most important aspect considers how and at which abstraction level PF-CCS can be integrated in the development process, and what the requirements for such an integration are.

3.4.1. **Value-Passing PF-CCS: Understanding Actions as Exchange of Messages**

PF-CCS uses pure CCS in order to model the behavior of non-variable processes and systems. In particular, in pure CCS communication simply means synchronization of actions which involves no explicit exchange of data. This abstract view onto communication does not perfectly fit the situation that we face during the development of a reactive system, where—at more concrete levels which are closer

to an implementation—communication is usually modeled as the exchange of data elements via a kind of communication channels. In [Mil95], Milner himself has extended pure CCS with a mechanism called *value passing CCS*, which integrates the idea of passing data elements along communication lines in order to realize communication. Some subsequent work [Bru91] introduces a concrete language for value passing CCS, which makes it even more applicable in a realistic context and lifts it to an entire programming language like setting, whose specifications can be directly imported in verification tools, e.g. like the concurrency workbench [Cle93]. However, in the same work [Mil95], Milner also argues that the value-passing extension is convenient, although theoretically unnecessary. PF-CCS—as it is introduced in this chapter—uses the mechanism based on pure CCS, i.e. it does not explicitly allow to model communication of data elements over channels.

In an analogous manner to the extension of CCS, also PF-CCS can easily be extended to a value-passing version, where actions correspond to tuples of channel names and data messages, and performing an action means to either send or receive a data element via a (directed) channel. However, we have omitted to introduce such an extension for PF-CCS in the scope of this thesis since (i) value-passing PF-CCS offers just a more convenient specification technique without giving more theoretical expressiveness [Mil95], and (ii) since a concept of directed channels that reflects communication between architectural entities requires also a corresponding underlying model of a component architecture that also facilitates the notion of variability. Since we do not provide such an architectural model in the scope of this thesis either, also a value-passing extension of PF-CCS makes only little sense.

3.4.2. **Placing PF-CCS in the Development Process**

In this thesis we have not integrated the PF-CCS framework into a corresponding development process, as our focus is on the theoretical level dealing with the fundamental question of how to combine product family concepts with an (process algebraic) approach to specify the operational behavior of systems. At such a formal, theoretical level, describing a reasonable integration would require the existence of compatible, formal models at other abstraction levels, e.g. the component architecture or user requirements level, which also support the notion of variability. The development of such models is beyond the scope of this thesis, as it opens a whole new field of research. However, the indispensable basis for a successful application of the PF-CCS framework in practice—and the exploitation of its advantages—is the integration of PF-CCS into a seamless model-based development process. Therefore, we sketch in the following our vision of how PF-CCS can be reasonably applied in an overall development process which allows the development of product families, as we encounter them for example in the automotive domain. In particular, we describe the abstraction level on which PF-CCS can be applied in a software product family development process.

3. PF-CCS: Product Family CCS

For single system development, [GHH07] and [BFG⁺08] introduce a development process which is suitable for the application in the automotive domain. However, this development process does not cover the notion of variability (in the sense of a software product family) and hence does not allow the development of a set of systems as a product family. Due to the lacking support of variability (and for several other reasons which we will discuss in the following) PF-CCS cannot reasonably be integrated into this specific development process. However, the structure of the development process, i.e. the structure of the consecutive abstraction layers and the aspects which are addressed by each layer, serves as a template for an equivalent development process for product families, in which PF-CCS can be used to represent the operational functionality of the product family and its products. We briefly introduce the development process of [GHH07, BFG⁺08] in the following in more detail, before we sketch an equivalent development process which contains PF-CCS.

The model-based development process of [GHH07, BFG⁺08] is based on several consecutive abstraction layers. The abstraction layers realize a separation of concerns in the sense that each layer addresses a special concern/aspect of the system design and provides the appropriate models. Although each layer represents a specific concern, the arrangement of the layers guarantees a seamless, overall development process. This means that the properties and aspects which are modeled at one layer are preserved in the successive layer, i.e. successive layers respect the properties of preceding layers and only contribute additional information. Thus, the models of consecutive abstraction layers provide a more and more concrete view of the system, leading towards a concrete implementation (Deployable code together with the deployment architecture). In this context we speak of an *abstraction layer framework*.

Figure 3.11 shows such a layer framework. The functional part of the (informally represented) requirements (topmost layer) is formalized in the second layer, the so-called *functional specification layer*. This layer gives an abstraction of the system which represents an implementation independent view of the system's functionality. The system's functionality is hierarchically structured into smaller functional entities, so-called *services* (cf. [BKM07]), where the behavior of any service is precisely defined by its direct sub-services. Depending on whether the focus is on a more interface-centric, black-box view, or on a more operational view of functionality, services can be specified using various specification techniques, such as stream-based functions [Bro05, BS01b], variants of I/O-automata [LT89], Message Sequence Charts [IT96], or a logic-based assumption/guarantee style [BS01b]. The subsequent layer, the *conceptual component architecture*, implements the functionality specified in the functional specification layer by a network of interacting, hierarchical, conceptual components which communicate by passing messages via directed channels. The channels and messages are realized as abstract data types, and the behavior of atomic components is given in an operational, "executable" but still implementation-platform independent way, using some appropriate representation techniques known from layer two, for example I/O-automata. Compared to the

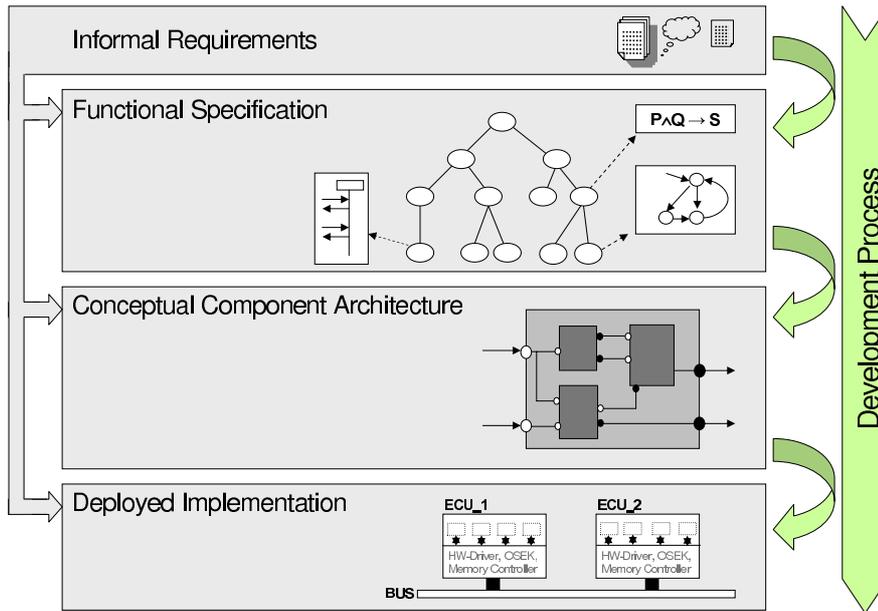


Figure 3.11.: A layer framework [GHH07, BFG⁺08] for a seamless model-based development process, whose separation into abstraction layers serves as a template for a corresponding development process for PF-CCS.

functional layer, the third layer introduces the new aspect of communication structure. Finally, the bottom layer, the so-called *deployed implementation*, corresponds to an implementation of the conceptual component architecture on a specific software/hardware platform. The conceptual component architecture of layer three is now implemented as a network of concrete software components which are allocated to ECUs (electronic control unit). Communication channels are implemented using concrete buses, protocols and technologies, and the behavior of components is given using concrete programming languages like C or Assembler. This abstraction layer represents an implementation of the final system. Throughout the abstraction layers, any non-functional requirements which are not explicitly covered in one of these layers are integrated into the relevant layers, respectively.

For the development of software product families we assume a layer framework with an equivalent structure, i.e. with equivalent abstraction layers and an equivalent separation of concerns. However, the models of every abstraction layer have to explicitly support the notion of variability. Assuming such a “product family layer framework”, PF-CCS allows to model the operational functionality and would integrate in such a product family layer framework at the lower part of the functional specification layer. In order to benefit from the behavioral variability which is expressible in the PF-CCS framework, successive abstraction layers have to take over the variability of PF-CCS programs. In particular the conceptual component architecture has to provide appropriate models that allow to take over the idea of

3. PF-CCS: Product Family CCS

variability to the architectural dimension. Regarding (structural) variability within the conceptual component architecture, some work in this direction has already been done by the Fraunhofer ISST and BMW as part of the project MOSES [ISS06].

Beside providing adequate variability concepts, all other models which are involved in such a layer framework have to match the process algebraic nature of PF-CCS. This comprises for example to have a “compatible” concept of how units of composition are structured, or the ability to deal with synchronous communication as it is used in PF-CCS. In particular the latter property of using synchronous communication is one reason why PF-CCS programs do not fit into the development process introduced in [GHH07, BFG⁺08], in which the models at the *functional specification* and *conceptual component architecture* are based on the more general form of asynchronous communication. Here, it seems more feasible to extend a process algebraic system engineering approach by variability concepts. However, process algebras are usually used by the theoretic community for fundamental reasoning rather than for the specification of complex, industry-scale systems. In particular, as far as we know there is no comprehensive system engineering method or process based on process algebras which is successfully applied for the development of realistic systems.

A process algebraic development method which comes the closest to a comprehensive system engineering method is LOTOS [BB87]. LOTOS is a ISO-standardized formal description technique based on CCS, CSP and ACT-ONE [EM85] for the design of distributed systems, which was used recently for example for the design of middleware behavior [RC04]. LOTOS supports various levels of abstraction, provides several specification styles, and comes with tools supporting specification, verification and code generation. It permits modeling of both synchronous and asynchronous communication and thus allows a smooth integration with PF-CCS, in principle. Even though in this thesis we have not integrated PF-CCS with LOTOS, the fact that LOTOS is also based on CCS is very conducive to a smooth integration with PF-CCS. Beside LOTOS, with PSF (Process Specification Formalism) [Die08] there exists a process algebra based specification language which is accompanied by *ToolBus* [BK96], a process algebra based coordination architecture developed at the CWI (Amsterdam) and the University of Amsterdam. *ToolBus* serves as an implementation model for PSF. Together, these approaches constitute a software engineering environment which aims at the development of software applications and allows specification of the behavior, the data types, and the structure of client/server systems, e.g. tool chains consisting of several interacting tools and programs that offer certain services. Due to their process algebra basis, both of these approaches seem to be generally suitable for the definition of a development process for software product families in which PF-CCS can be integrated. Still, a more in-depth consideration is necessary, since small pitfalls have to be overcome, for example that *ToolBus* cannot directly handle recursive processes as they are allowed in PF-CCS.

Regarding *architectural description languages* (ADLs), there exists a large pool of different approaches (Medvidovic et al. [MT00] give a nice summary and compari-

son). Some of these are based on a process algebra, for example (i) *Wright* [All97], an ADL developed at the Carnegie Mellon University which uses CSP to capture the dynamic behavior of its architectural entities, (ii) *Darwin* [MDEK95], an ADL developed at Imperial College London, whose operation semantics is given in terms of Milner's π -calculus [Mil99], and (iii) *PADL* [BCD02], by Bernardo et al., which even aims at describing architectures in the presence of variable components with the aim to detect and rule out mismatches which arise due to the combination of components. These process algebra based ADLs are also possible candidates for a combination with PF-CCS and its software product family specific concepts.

In summary, although the integration of PF-CCS into the development process is not done in this thesis, it is necessary to bring a framework like PF-CCS into practice. As we have just seen, some possibilities for an integration with existing techniques and frameworks (for the development and design of single systems) exist. However, the development of the corresponding models and description techniques which provide an appropriate variability concept at the various abstraction levels requires more in-depth research and represents a reasonable continuation of the research carried out in this thesis.

3.5. Related Work

The first process algebras have been developed in the 1970s, driven by different groups and persons simultaneously, and have been a fruitful area of research since then. Consequently, up to today, a large variety of different kinds of process algebras for various purposes has been developed. A nice overview of the evolution of process algebras, which also covers various extensions of process algebras, is given by Baeten in [Bae05]. However, to our knowledge, until today there is no single process algebraic approach that explicitly integrates modeling techniques for the specification of variants in the sense of a software product family. From this point of view, our PF-CCS approach is novel.

The approaches which bear the greatest resemblance to our approach use nondeterminism as one way of representing a set of possible versions or concrete models of a system. Although, this is still far from being a product family, the idea of explicitly understanding a nondeterministic specification as a set of concrete (deterministic) specifications, which represent the concrete systems, is very close to the intention of a software product family. Thus, we briefly introduce some of these approaches in the following.

Close to our approach is an approach by Vegliani and De Nicola [VN98]. Even if they do not introduce a product family extension, they define how a nondeterministic process can be understood as a representation of a set of deterministic processes, its

3. PF-CCS: Product Family CCS

so-called *possible worlds*. For this purpose they introduce several semantics which relate a nondeterministic process to a set of its possible worlds. The designated goal of their approach is to provide a specification framework where refinement can be proven efficiently, not to provide a specification technique for product families. Their setting is a basic process algebra consisting of a *Nil* constant, a *prefix* and *choice* operator $+$, but without recursion. On top of this basic algebra they introduce a so-called *underspecification operator* \oplus . Using the notation of Viglioni and De Nicola, the central axiom for their \oplus operator is

$$aP + aQ = aP \oplus aQ$$

where a is an action and P, Q are process identifiers. This means that they understand and realize their underspecification operator \oplus simply as a regular nondeterministic choice. In particular, this implies that we cannot use such an \oplus operator in order to model alternative variants in our sense as it directly contradicts the fundamental axioms of our \oplus operator as defined in the product family axiomatization. Moreover, such a law contradicts our view of a variants operator, which cannot be “simulated” or implemented by a nondeterministic choice operator, since the alternative selection of variants is an orthogonal concept compared to the nondeterministic choice between two processes. For example—recall the discussion in Chapter 2.2.2 and 3.1—the ordinary nondeterministic recursive process

$$P \stackrel{\text{def}}{=} \alpha.P + \beta.Q$$

does not guarantee that in a recursive re-entrance, the same side of the nondeterministic $+$ operator, which has been chosen in the preceding recursion, is selected again in the upcoming re-entrance. For example, if during a first step, the left-hand side process $\alpha.P$ is chosen, there is no guarantee that in the following recursive re-entrance the right-hand side process $\beta.Q$ is not executed instead, according to the fact of being a nondeterminism choice. However, for a product line this is not feasible since whenever we have chosen one concrete variant of a variation point (\oplus operator), the same choice has to be made in every upcoming recursive re-entrance. This fundamental difference makes the approach of Veglioni and De Nicola not usable for our purpose of specifying a software product line.

In [MC01], Majster-Cederbaum takes on Veglioni and De Nicolas’ approach and extends it to a setting with recursion. On this basis she discusses some problems which arise in the recursive setting and gives some possible extensions which move her framework closer to our PF-CCS framework, than the one of Veglioni and De Nicola actually is. However, the general scenario of Majster-Cederbaum’s work is still the same as the one of Veglioni and De Nicola: underspecification. In particular, this approach is not tailored to product lines, either, and cannot be considered in our sense as a software product family framework.

Another specification technique which we want to mention in this context are modal transition systems [LT88] as proposed by Larsen and Thomson in 1988. Even though

modal transition systems are in a first instance not a process algebraic technique, they are still very suitable to serve as a semantic domain when giving a SOS semantics for a process algebra. For example for PF-CCS, the PF-LTS transition systems which we have introduced to define our SOS semantics can be seen as a special kind of modal transition systems.

A modal transition system has two flavors of transitions: so-called *must* and *may* transitions. If a modal transition system has a *must* transition, all refining systems must have this transition, too, while a *may* transition can exist in a refining system, but does not necessarily have to. This definition already indicates the area of application for modal transition systems: it offers a more systematic development process by relating specifications to its implementations within a single specification technique. In this context we also speak of *mixed specifications*.

The concept of *must* and *may* transitions bears a basic resemblance to what we define as mandatory and optional parts within a product family: If we consider a product family, some derivable products may contain a certain behavior while others may not; thus, *may* transitions correspond to optionality. A *must* transition can be used to model mandatory behavior which always exists in every derivable product. However, for the modeling of a full software product family in our sense *must* and *may* transitions are not yet sufficient since they do not allow to express a more fine grain relation between *may* transitions and the corresponding configuration of the product family.

In [LNW07], Larsen et al. introduce a special kind of modal transition systems, so-called *modal I/O automata*, which basically extend interface automata [dAH01] with modality. In the same paper the authors show how modal I/O automata can be applied for product line development, by sketching the basic ideas. This is done by introducing a behavioral formalism for specifying the variability of systems.

The PF-LTS transition system, which we introduced for the definition of the semantics of PF-CCS, lifts the idea of modality to a higher dimension: multi-modality. Instead of only considering *may* transitions, in a multi-modal transition system we now have multiple instances of *may*-values, where each one is associated with a unique property. Thus, we can not only express that a transition may be present, but that a transition may be present in a specific situation under specific circumstances. In our setting, these circumstances reflect the different configurations of a product family, and thus allow to model that a transition is only present in a certain set of products. The formalism of Larsen et al. [LNW07] is not capable of expressing such a detailed information.

Fantechi et al. [AtBGF09] use deontic logics to model variabilities in product family descriptions. Connecting deontic logics with modal transitions systems, they show how modal transition systems can be characterized with deontic logics, and thus

3. *PF-CCS: Product Family CCS*

provide a way of how a logical-based formalism can be used to specify variable systems. Using deontic logics makes it possible to express concepts like obligation and permission, and to show some properties of variable systems. However, since they base their formalism on modal transition system which do not support the notion of different configurations, also this approach is not directly useful for us to model the operational behavior of software product families as we are interested in.

In [CHS⁺10], Classen et al. introduce an approach for model checking entire product families. Product families are modeled as feature-labeled transitions systems which are very similar to our notion of a PF-LTS. For a more detailed discussion of their approach we refer to Section 4.4 (Page 188).

Verifying Properties of PF-CCS Software Product Families

In this section we introduce the logic $mv\text{-}\mathcal{L}_\mu$, which is a *multi-valued* extension of the modal μ -calculus for the specification of behavioral properties of PF-CCS product families. Formulae of $mv\text{-}\mathcal{L}_\mu$ are interpreted over multi-valued transition systems, which we introduced in the preceding chapter in order to define the semantics of PF-CCS product families. The interpretation of a $mv\text{-}\mathcal{L}_\mu$ formula φ over a PF-CCS product families is no longer a “simple” yes/no result, but rather yields a *set of configurations*. We show that the contained configurations represent exactly those products of the corresponding product family which fulfill φ . With respect to the evaluation of $mv\text{-}\mathcal{L}_\mu$ formulae, we have designed $mv\text{-}\mathcal{L}_\mu$ in a way that existing model checking algorithms can be applied to compute the value of a $mv\text{-}\mathcal{L}_\mu$ formula with respect to a given multi-valued transition system.

Contents

4.1. The Multi-Valued Modal μ -calculus	162
4.2. Model Checking	177
4.3. Example: Verifying a Family of Windscreen Wipers	180
4.4. Related Work	185

4. Verifying Properties of PF-CCS Software Product Families

In the preceding Chapter 3 we have introduced PF-CCS, an algebraic specification framework which allows to specify the behavior of a set of similar systems as a software product family. As we have seen, a PF-CCS model represents an entire product family with all its derivable products. In particular, it not only comprises the behavior of each individual product, but it also contains the information how the behaviors of different products are related with each other. As we have already seen in the preceding chapter—and as we will discuss in even more detail in Chapter 5—this is the basis to talk about common and optional behavior of similar systems.

However, being able to specify a PF-CCS product family is just half of the story. Especially for reactive systems, as important as the specification itself is the analysis of the system’s behavior, i.e. to check if a system meets certain behavioral properties. In the case of safety-critical systems, we even want to verify such properties formally. In the context of single systems—when a system is seen in its own rights and not in relation to other similar systems—there is a multitude of modal logics like linear temporal logic LTL [Pnu77], computation tree logic CTL [CE81a], CTL* [EL86] (cf. [CGP99]), or the modal μ -calculus [Koz83] which allow to specify such properties. Usually, these logics are accompanied by a variety of verification techniques, e.g. model checking [CGP99], and corresponding tools (e.g. [Cle93, Hol03, CCG⁺02, BLL⁺95]) that allow to check the properties on the given model in a more or less automated and efficient way. The result of such a check is usually a yes/no answer indicating whether a property holds for the system or not.

Certainly, we can apply these logics and tools to investigate and verify properties of *single* products of a product family in the same way as we apply them for verifying properties of any other standalone system. However, in particular in the context of a software product family we are often interested in properties which span several products. In the automotive domain for example, where a car model, e.g. the *7-series BMW*, can be ordered in a huge variety of different configurations [Sch08], the developers are often faced with the question in which variants of a series a certain property holds. For example, does every *7-series BMW* model which is equipped with Break-by-Wire [Jur09] technology always break when the break pedal is operated, independently of the specific variant of the ACC (Adaptive Cruise Control) being installed in the respective model? The naive way to check whether a property holds for all products of a product family is to derive the specification of every product and to check the property in the default manner for every product individually. Certainly, this is the most expensive way in terms of time and effort and it becomes more and more intricate with an increasing number of variants. In fact, in situations where not all model variants can explicitly be constructed, such an approach is not feasible at all. Apart from that, if the products have a high degree of behavioral commonalities, a common behavior between variants is not taken into account for the verification using the naive method of checking each configuration variant separately.

In contrast to single system development, if systems are developed as members of a software product family, behavioral commonalities should also be considered for the verification of behavioral properties. Instead of performing the verification of properties on the specific models of the concrete products, we perform the verification directly on the model of the product family, itself. Especially for properties which apply to several configurations this is the adequate way, since it overcomes the previously stated drawbacks. However, current temporal logics are not explicitly tailored for the application on models which provide the explicit notion of variants, such as for example a software product family. The meaning of formulae of traditional two-valued logics like the μ -calculus, CTL, or LTL, for the model of a product family is per se not clear. For product families, traditional logics do not suffice, as they do not support the concept of variants or variability in the logic itself.

Therefore, in this chapter we introduce an adequate modal logic which allows to explicitly deal with configurations as they are relevant in a software product family. We call the logic the *multi-valued modal μ -calculus*. The multi-valued modal μ -calculus is tailored to be a property specification language suitable for the specification of properties of PF-CCS programs. Even though the logic is interesting mostly from a theoretic point of view and not interesting for a practitioner, the multi-valued modal μ -calculus is very suitable as a specification language in combination with model checking.

In this context we show how existing model checking techniques can be used to check multi-valued modal μ -calculus properties on a PF-CCS specification of the system (family). In particular, we can reduce typical questions from software product line engineering to model checking problems. In our setting, the result of model checking a property of a PF-CCS program is the *set* of configurations satisfying the property at hand and not only the answer if the property holds or not.

We have chosen to develop our verification approach for specifications as a variant of the μ -calculus, as it is a very expressive logic which subsumes linear temporal logic (LTL) as well as computation tree logic (CTL) as first shown in [EL86, Wol] and nicely summarized in [Dam94]. This makes our multi-valued modal μ -calculus a fundamental logic which is capable of handling product family specific properties in general, and which is not limited to a certain flavor of logic like linear-time or branching-time. On the other hand, the decision for the μ -calculus also means that we have chosen a (fixpoint) logic which is notorious for being incomprehensible, and which is far away from being applicable in industrial practice. If used at all, temporal logics like CTL or LTL are more common in industrial practice, depending on the domain and area of application. However, there exists a large pool of translations of common CTL and LTL properties into equivalent μ -calculus formulae [Sti01]. On an even more abstract level, there are also property specification languages like SALT [BLS06] which aim at a more comfortable specification of temporal properties in the industrial practice. SALT can be seen as a specification front end which translates

4. Verifying Properties of PF-CCS Software Product Families

a (program like) property specification into a corresponding temporal logic formula. Techniques like these relativize our choice for the μ -calculus for the transfer to the industrial practice.

4.1. The Multi-Valued Modal μ -calculus

In the following we introduce a multi-valued version of the modal μ -calculus as a property specification language tailored specifically for PF-CCS software product families. For the scope of this thesis we refer to this multi-valued version as the *multi-valued modal μ -calculus*. The *multi-valued modal μ -calculus* is a modal logic which bases on the modal μ -calculus [Koz83], and a general, multi-valued version of the μ -calculus as described by Bruns and Godfroid [BG04], and Shoham and Grumberg [SG05]. With the multi-valued modal μ -calculus we lift the multi-valued μ -calculus as defined in [SG05] to the application area of software product families in general, but in particular to the area of PF-CCS product families. More precisely, we (i) enrich [SG05], which only supports unlabeled diamond and box operators, by providing also action-labeled versions of these operators, which is essential to formulate properties of PF-CCS programs, and (ii) adjust the version of [SG05] to the specific case of PF-CCS product families. Thus, our terminology “multi-valued modal” is somehow misleading, since strictly speaking we define a multi-valued (i.e. the result of a formulae is no longer restricted to one of the two values *true* or *false*) and multi-modal (i.e. the modal operators now correspond to action-labeled families of operators) version of the μ -calculus. However, we stick to the shorter name for simplicity.

The essential difference between the original modal μ -calculus and our multi-valued version lies in the semantical interpretation of formulae. Formulae of our multi-valued modal μ -calculus are not interpreted over a regular Kripke structure, but over a Kripke structure defined over a powerset lattice of configurations. This means that the transitions of the Kripke structure which we consider for our interpretation are labeled with actions (similar to a regular Kripke structure), and additionally with a set of configurations. This set is an element of the powerset lattice over configurations, and contains exactly those configurations in which this particular transition exists. From this point of view the Kripke structures over which we interpret formulae of our multi-valued μ -calculus correspond exactly to the PF-LTS transition systems which we use in Chapter 3 to represent the semantics of PF-CCS product families. We define the syntax and the semantics of the multi-valued modal μ -calculus in the remainder of this section. Note that for this chapter we require some fundamental knowledge of basic lattice theory, as well as knowledge of the (standard) modal μ -calculus, and in particular its semantics. For the reader who is not familiar with these topics we have summarized the relevant prerequisites in the Appendices C (Page 267), and D (Page 269).

4.1.1. Syntax of the Multi-Valued Modal μ -Calculus

Syntactically, our multi-valued modal μ -calculus is equivalent to the standard μ -calculus [Koz83]. Formulae of the multi-valued modal μ -calculus are built from propositions, boolean connectives, modal operators, variables, and fixpoint operators in the same way as in the standard μ -calculus. Let

- \mathcal{V} be an (infinite) set of *propositional (fixpoint) variables*,
- \mathcal{P} be a set of *propositional constants*, and
- \mathcal{A} be a set of *action names*.

Usually, we use the capital letters X, Y, Z to range over variables in \mathcal{V} , the lowercase letters p, q to range over propositions in \mathcal{P} , and α, β to range over actions in \mathcal{A} . Formulae of the multi-valued modal μ -calculus are given by the EBNF grammar shown in Figure 4.1. The symbols \wedge and \vee denote the logical AND and OR operations in the usual way. With $[\alpha]$ and $\langle \alpha \rangle$ we denote the modal box (necessity) and diamond (possibility) operator, which are both labeled with an action $\alpha \in \mathcal{A}$. Intuitively, the meaning of $\langle \alpha \rangle \varphi$ is that there exists a state that can be reached by an α -transition in which the property φ holds. Similarly, $[\alpha] \varphi$ means that φ holds in all states which can be reached by an α -transition. The symbols μ and ν denote the fixpoint operators yielding the smallest and largest fixpoint, respectively.

$$\begin{array}{l}
 \varphi = \quad true \quad | \quad false \\
 \quad | \quad q \\
 \quad | \quad \neg q \\
 \quad | \quad \varphi \vee \varphi \quad | \quad \varphi \wedge \varphi \\
 \quad | \quad \langle \alpha \rangle \varphi \quad | \quad [\alpha] \varphi \\
 \quad | \quad Z \\
 \quad | \quad \mu Z. \varphi \quad | \quad \nu Z. \varphi
 \end{array}$$

Figure 4.1.: Syntax of the multi-valued modal μ -calculus $mv\text{-}\mathcal{L}_\mu$, where $q \in \mathcal{P}$ denotes an atomic proposition, $\alpha \in \mathcal{A}$ an action, and $Z \in \mathcal{V}$ a (fixpoint) variable.

With $mv\text{-}\mathcal{L}_\mu$ we denote the set of *closed* multi-valued modal μ -calculus formulae generated by the grammar shown in Figure 4.1, where the fixpoint quantifiers μ and ν are variable binders. In the following we write η for either μ or ν in order to ease the notation. In order to emphasize that a variable X occurs in a formula φ

4. Verifying Properties of PF-CCS Software Product Families

we write $\varphi(X)$. We assume that formulae are well-named, i.e. no variable is bound more than once in any formula.

Following the definition of the standard μ -calculus, the formulae of $mv\text{-}\mathfrak{L}_\mu$ as introduced in Figure 4.1 are in so-called *positive normal form*. This means (i) that the negation operation \neg is only applied directly to atomic propositions in $q \in \mathcal{P}$, and (ii) that all bound variables are distinct. The positive requirement is a syntactic means to ensure that a formula $\varphi(Z)$ is actually monotonic in Z , which is the basis for having unique maximal and minimal fixpoints. Note that in other sources, e.g. [BS01a], the μ -calculus syntax is introduced in *parsimonious form*. Here, negation can be applied to arbitrary formulae, and derived operators are defined by de Morgan duality in the usual manner, e.g. $\varphi_1 \vee \varphi_2$ is defined as $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, the diamond operator $\langle \alpha \rangle \varphi$ is defined in terms of the box operator as $\neg[\alpha]\neg\varphi$, etc. However, for such formulae it has to be ensured that every free occurrence of a variable Z only occurs within the scope of an even number of negations. Certainly, this implies an administrative overhead compared to the positive normal form. However, according to [BS01a], both forms are equivalent, as any formula can be turned into positive normal form by use of de Morgan laws and variable renaming.

4.1.2. Semantics of the Multi-Valued Modal μ -Calculus

While syntactically the multi-valued μ -calculus is equivalent to the standard μ -calculus, semantically the calculi are not comparable, since (i) they are interpreted over different kinds of Kripke structures, and (ii) the interpretations yield different kinds of result values. While a formula of the standard modal μ -calculus is interpreted over a regular Kripke structure (cf. [CGP99]) yielding a *true/false*-information whether the formula holds in a given state, formulae of our multi-valued μ -calculus are interpreted over so-called multi-valued modal Kripke structures yielding an element of a lattice (for every atomic proposition in \mathcal{P}). As we will explain in the following, such a multi-valued modal Kripke structure is just a slightly adjusted way of representing a multi-valued labeled transition system (PF-LTS), which we have used in the preceding chapter in order to represent the semantics of a PF-CCS product family. Accordingly, the lattice element (which we obtain for a fixed proposition) which is the result of interpreting a multi-valued μ -calculus formula over a multi-valued Kripke structure represents the set of configurations in which the desired property holds for the given product family (and for the given proposition).

For the definition of the semantics of $mv\text{-}\mathfrak{L}_\mu$ -formulae—which we will provide in the remainder of this subsection—we proceed as follows: Initially we give a formal definition of a multi-valued modal Kripke structure. Then we explain the connection between the lattice elements which that the states and the transitions in a multi-valued modal Kripke structure are labeled and the configurations of a product

family. Then we proceed to define the semantics of a $mv\text{-}\mathfrak{L}_\mu$ -formula. The semantics is defined over multi-valued modal Kripke structures, since Kripke structures are the common kind of structure to represent system (families) in the context of temporal logics. After we have defined the semantics we show how an MMKS can be understood as a PF-LTS and thus bridge the formal gap between the logic $mv\text{-}\mathfrak{L}_\mu$ and the semantic domain which we use for PF-CCS product families. Ultimately, this allows to use formulae of $mv\text{-}\mathfrak{L}_\mu$ in order to reason about PF-CCS product families.

Multi-Valued Modal Kripke Structures

As we have defined in the beginning of this chapter, let \mathcal{P} be a set of propositional constants, and \mathcal{A} be a set of action names. We define the semantics of $mv\text{-}\mathfrak{L}_\mu$ formulae in terms of multi-valued modal Kripke structures.

Definition 4.1 (Multi-Valued Modal Kripke Structure (MMKS)). *Let $(\mathcal{L}, \sqcap, \sqcup, \neg)$ be a (complete) de Morgan lattice¹. A multi-valued modal Kripke structure (MMKS) is a tuple*

$$\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L)$$

where

- \mathcal{S} is a set of states,
- for each action $\alpha \in \mathcal{A}$, the relation $\mathcal{R}_\alpha : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{L}$ is a valuation function mapping each pair of states to a lattice element, and
- $L : \mathcal{S} \rightarrow \mathcal{L}^{\mathcal{P}}$ is a (labeling) function yielding a value in \mathcal{L} for each state $s \in \mathcal{S}$ and proposition $p \in \mathcal{P}$.

We write $(\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L, \sigma)$ for a MMKS with start state σ .

Unlike to standard Kripke structures, where each states holds the *yes/no*-information whether an atomic proposition holds in this state, in a MMKS each state holds the information to which *degree* every proposition in this state is fulfilled. The degree is represented by an element of the lattice \mathcal{L} over which the Kripke structure is defined. As we will explain in the following in more detail, for our purpose we let \mathcal{L} always be the powerset lattice over a given set of configurations, which means that individual elements of the lattice represent *sets of configurations*. Note that we can equivalently denote the labeling function L in a more functional style as a function $L : \mathcal{S} \rightarrow (\mathcal{P} \rightarrow \mathcal{L})$, emphasizing that L actually returns a lattice element in \mathcal{L} . The transitions in a MMKS are labeled with an action $\alpha \in \mathcal{A}$ and a lattice element

¹In a de Morgan lattice every element x has a unique complement $\neg x$. For more details on lattices see the Appendix C, Page 267ff.

4. Verifying Properties of PF-CCS Software Product Families

$l \in \mathcal{L}$. Regarding the definition of the transition relation, we denote the entire relation as a family of individual transition relations \mathcal{R}_α . For example, for all actions in $\mathcal{A} = \{\beta_1, \beta_2, \dots\}$, the relation \mathcal{R}_{β_1} represents the β_1 -transitions only, \mathcal{R}_{β_2} only the β_2 -transitions, etc. For each state s we require that it has a reasonable successor, i.e. at least one successor state s' which can be reached under an α -transition with a lattice value “greater” than \perp , i.e. we require $\exists \alpha \in \mathcal{A}, \exists s' \in \mathcal{S} : \mathcal{R}_\alpha(s, s') \neq \perp$. We write $\mathcal{T}.s$ to denote the state s of the MMKS \mathcal{T} .

A MMKS is like a regular Kripke structure [CGP99], except that propositions and transitions are now interpreted over a lattice \mathcal{L} . Usually, a regular Kripke structure $(S, \xrightarrow{\alpha}, L)$ consists of a finite set S of states, an action-labeled transition relation $\xrightarrow{\alpha} \subseteq S \times \mathcal{A} \times S$, which is assumed to be total, and a labeling function $L : S \rightarrow \{\text{true}, \text{false}\}^{\mathcal{P}}$ which associates either the truth value *true* or *false* to every atomic proposition $p \in \mathcal{P}$ in a given state, indicating whether the proposition holds or not in this state. In contrast thereto, in a MMKS the atomic propositions (at given states) as well as the transitions are now interpreted as elements of a lattice, which represent the degree to which a certain proposition (in a given state) or transition holds or exists.

Note that a Kripke structure in the usual sense can be regarded as a MMKS with values over the two element lattice $(\{\perp, \top\}, \sqsubseteq)$ consisting of a bottom element \perp and a top element \top , ordered in the expected manner $\perp \sqsubseteq \top$. Value \top then means that the property holds (it represents *true*) in the considered state while \perp means that it does not hold. Similarly, $\mathcal{R}_\alpha(s, s') = \top$ reads as there is a corresponding α transition while $\mathcal{R}_\alpha(s, s') = \perp$ means there is no α -transition from s to s' in the respective MMKS.

Connection between the Lattice and Configurations of Software Product Families

The lattice in a MMKS can be any arbitrary de Morgan lattice (cf. Appendix C for the de Morgan property). However—in particular for software product families—a very important example of a de Morgan lattice is the powerset lattice $(\mathcal{P}(S), \subseteq)$ defined over a set S , where the ordering is given by subset inclusion. The elements of the lattice are the elements of the powerset of S , i.e. individual *subsets* of the set S . The lattice operation \sqcap (*meet*) is given by set intersection \cap , the operation \sqcup (*join*) is given by set union \cup , and the lattice complement operation \neg is given by set complement $M^c := S \setminus M$, $M \subseteq S$, respectively. In the lattice $(\mathcal{P}(S), \subseteq)$, the set S takes the role of \top , while the empty set \emptyset takes the role of \perp .

In order to model a PF-CCS software product family, we let the set S be the set of all possible configurations of the software product family. Recall, that we denote the set of all (combinatorially) possible configurations of a PF-CCS product

family $Prog$ by $CONFIGS_{Prog}$ (cf. Section 2.2.3.3), where we omit the index $Prog$ if it is clear which system we mean. Thus, an individual element $s \in CONFIGS$ represents a single (complete) configuration of the product family, which associates a concrete variant to each variation point. In a PF-CCS software product family with $N \in \mathbb{N}$ variation points (offering 2 alternative variants each) the set $CONFIGS$ has $|CONFIGS| = 2^N$ elements which represent all N possible configurations of the software product family. Then, any element $s \in \mathcal{P}(CONFIGS)$ of the powerset lattice over $CONFIGS$ corresponds to a set of configurations.

A MMKS over such a powerset lattice $(\mathcal{P}(CONFIGS), \subseteq)$ is appropriate to represent a software product family. The elements of the lattice $(\mathcal{P}(CONFIGS), \subseteq)$ which are attached to transitions represent the set of configurations in which this particular transition is present, i.e. $\mathcal{R}_\alpha(s, s')$ denotes the set of configurations in which there is an α -transition from state s to s' . Likewise, the lattice elements which are used to interpret the atomic propositions in individual states represent the set of configurations in which this proposition holds. The two extreme lattice elements $CONFIGS$ and \emptyset represent the situation, when a transition (or a proposition) exists in every configuration (represented by $CONFIGS$), or when it does not exist at all (represented by \emptyset).

However, for the following definition of the semantics we will use a general de Morgan lattice instead of its special case, the power lattice of configurations. This represents the more general case, but more importantly allows to apply directly certain model checking algorithms and techniques, which are defined on a general lattice structure. In any case, if we interpret the general de Morgan lattice as a powerset lattice of configurations as described above, then the following semantics definition is directly applicable for PF-CCS software product families.

Semantics of $mv\text{-}\mathcal{L}_\mu$ Formulae

The semantics of a $mv\text{-}\mathcal{L}_\mu$ formula is an element of $\mathcal{L}^{\mathcal{S}}$, i.e. a *satisfaction function*

$$f : \mathcal{S} \rightarrow \mathcal{L}$$

yielding for the formula at hand and a given state the *satisfaction value* (given in terms of the lattice \mathcal{L}). In our product family setting, this satisfaction value is the set of configurations for which the formula holds in the given state. With $FUNCTIONALS$ we denote the set of all satisfaction functions. Let the *environment* which explains the meaning of free variables in $mv\text{-}\mathcal{L}_\mu$ formulae be represented by the function

$$\rho : \mathcal{V} \rightarrow \mathcal{L}^{\mathcal{S}}$$

Note that this is equivalent to $\rho : \mathcal{V} \rightarrow (\mathcal{S} \rightarrow \mathcal{L})$. With

$$\rho[Z \mapsto f]$$

4. Verifying Properties of PF-CCS Software Product Families

$$\begin{aligned}
\llbracket true \rrbracket_\rho &:= \lambda s. \top \\
\llbracket false \rrbracket_\rho &:= \lambda s. \perp \\
\llbracket q \rrbracket_\rho &:= \lambda s. L(s)(q) \\
\llbracket \neg q \rrbracket_\rho &:= \lambda s. \neg L(s)(q) \\
\llbracket \varphi \vee \psi \rrbracket_\rho &:= \llbracket \varphi \rrbracket_\rho \sqcup \llbracket \psi \rrbracket_\rho \\
\llbracket \varphi \wedge \psi \rrbracket_\rho &:= \llbracket \varphi \rrbracket_\rho \sqcap \llbracket \psi \rrbracket_\rho \\
\llbracket \langle \alpha \rangle \varphi \rrbracket_\rho &:= \lambda s. \bigsqcup_{s' \in \mathcal{S}} \{ \mathcal{R}_\alpha(s, s') \sqcap \llbracket \varphi \rrbracket_\rho(s') \} \\
\llbracket [\alpha] \varphi \rrbracket_\rho &:= \lambda s. \bigsqcap_{s' \in \mathcal{S}} \{ \neg \mathcal{R}_\alpha(s, s') \sqcup \llbracket \varphi \rrbracket_\rho(s') \} \\
\llbracket Z \rrbracket_\rho &:= \rho(Z) \\
\llbracket \mu Z. \varphi \rrbracket_\rho &:= \bigsqcap \{ f \mid \llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} \sqsubseteq f \} \\
\llbracket \nu Z. \varphi \rrbracket_\rho &:= \bigsqcup \{ f \mid f \sqsubseteq \llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} \}
\end{aligned}$$

Figure 4.2.: Semantics of $mv\text{-}\mathfrak{L}_\mu$ formulae.

we denote the environment that maps Z to f and agrees with ρ on all other arguments. The semantics $\llbracket \varphi \rrbracket_\rho^\mathcal{T}$ of a $mv\text{-}\mathfrak{L}_\mu$ formula φ with respect to a MMKS $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L)$ and an environment ρ , is defined inductively on the structure of the formula as shown in Figure 4.2. We assume \mathcal{T} to be fixed and do not mention it explicitly anymore. In the case of closed formulae we will also drop the environment ρ from the semantic brackets.

The semantics given in Figure 4.2 is defined in a standard manner. The constant *true* is fulfilled in all states to the maximal degree. Thus, the corresponding function $\lambda s. \top$ always returns the maximal lattice value \top independently of the specific state. For the interpretation of the lattice values as sets of configurations this means that the formula *true* is fulfilled in any state by all configurations, represented by the set S of all possible configurations. The opposite holds for the constant *false*, which can not be fulfilled at all in any state, represented by the lattice value \perp which corresponds to the empty set of configurations. The semantics of atomic propositions is directly given by evaluating the labeling function for the specific state and proposition. Note that the symbol \neg in Line 4 in Figure 4.2 is overloaded: on the left-hand side of defining equation it represents the logical negation operation of $mv\text{-}\mathfrak{L}_\mu$, while on the right-hand side it represents the complement operation of the lattice. Realizing the $mv\text{-}\mathfrak{L}_\mu$ negation directly by the lattice complement operation is possible since in a de Morgan lattice every element has a unique complement. The semantics of the boolean operations \wedge and \vee is straight forward.

The only operators deserving a more thorough discussion are the diamond and box-operator (cf. Lines 7 and 8 in Figure 4.2). Intuitively, $\langle\alpha\rangle\varphi$ is classically supposed to hold in states that have an α -successor satisfying φ . In a multi-valued version, we first consider the value of an α -transition and reduce it (meet it) with the value of φ in the successor state (inner meet). As there might be different α -transitions to different successor states, we take the best value (outer join). For PF-CCS programs, this meets exactly our intuition: a configuration in state s satisfies a formula $\langle\alpha\rangle\varphi$ if it has an α -successor satisfying φ , where we chose the best α -successor (the one which exists in the most configurations) in the case when multiple α -successors exist. The semantics (Line 8 in Figure 4.2) of the box operator is based on the fact that box is the dual of the diamond operator, i.e.

$$\langle\alpha\rangle\varphi \equiv \neg[\alpha]\neg\varphi$$

which is equivalent to

$$\neg\langle\alpha\rangle\neg\varphi \equiv [\alpha]\varphi$$

Thus, based on the semantics of our diamond operator we derive the semantics of the box operator as:

$$\begin{aligned} \llbracket[\alpha]\varphi\rrbracket_\rho &:= \llbracket\neg\langle\alpha\rangle\neg\varphi\rrbracket_\rho \\ &= \neg\lambda s. \bigsqcup_{s' \in \mathcal{S}} \{ \mathcal{R}_\alpha(s, s') \sqcap \llbracket\neg\varphi\rrbracket_\rho(s') \} \\ &= \lambda s. \bigsqcap_{s' \in \mathcal{S}} \{ \neg \left(\mathcal{R}_\alpha(s, s') \sqcap \llbracket\neg\varphi\rrbracket_\rho(s') \right) \} \\ &= \lambda s. \bigsqcap_{s' \in \mathcal{S}} \{ \neg\mathcal{R}_\alpha(s, s') \sqcup \neg\llbracket\neg\varphi\rrbracket_\rho(s') \} \\ &= \lambda s. \bigsqcap_{s' \in \mathcal{S}} \{ \neg\mathcal{R}_\alpha(s, s') \sqcup \llbracket\varphi\rrbracket_\rho(s') \} \end{aligned}$$

Intuitively, the semantics of the box operator $[\alpha]\varphi$ seems to be somehow less obvious to understand. We illustrate it with the concrete example of the powerset lattice of configurations. Since we are interested in the set of configurations in that every α -successor state s' can fulfill φ we take the intersection over all possible α -transitions, which motivates the outer meet. For the inner part, $\neg\mathcal{R}_\alpha(s, s') \sqcup \llbracket\varphi\rrbracket_\rho(s')$, suppose we consider a transition labeled with the set of configurations $\mathcal{R}_\alpha(s, s')$. We are interested in all configurations which fulfill the property: “*If there is a way from s to s' in this particular configuration, then the successor state s' must also afford this configuration*”. For all those configurations which are *not* in $\mathcal{R}_\alpha(s, s')$ this property certainly holds, and we include them in the semantics (represented by the term $\neg\mathcal{R}_\alpha(s, s')$). In addition, those configuration which appear both in $\mathcal{R}_\alpha(s, s')$ and

4. Verifying Properties of PF-CCS Software Product Families

in $\llbracket \varphi \rrbracket_\rho(s')$, the property holds, too. These configurations are given by $\mathcal{R}_\alpha(s, s') \sqcap \llbracket \varphi \rrbracket_\rho(s')$. Combining those two sets which fulfill the property yields

$$\neg \mathcal{R}_\alpha(s, s') \sqcup \left(\mathcal{R}_\alpha(s, s') \sqcap \llbracket \varphi \rrbracket_\rho(s') \right) = \neg \mathcal{R}_\alpha(s, s') \sqcup \llbracket \varphi \rrbracket_\rho(s')$$

which is the desired result for the inner part of the semantics of the box operator.

The semantics of atomic variables is determined exclusively by the environment $\rho : \mathcal{V} \rightarrow \mathcal{L}^{\mathcal{S}}$ which returns the degree (satisfaction value) to which each variable is fulfilled in a certain state. The fixpoint operators μ and ν —yielding the least and greatest fixpoint, respectively—are also defined in the usual manner. For these fixpoint operators we are interested in the functionals

$$\lambda f. \llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} : \mathcal{L}^{\mathcal{S}} \rightarrow \mathcal{L}^{\mathcal{S}}$$

where for every concrete variable the corresponding functional maps satisfaction functions to satisfaction functions. For any Z, φ and \mathcal{S} , these functionals are monotonic (for pointwise ordering) w. r. t. \sqsubseteq ². Thus, according to Tarski's fixpoint theorem [Tar55], least and greatest fixpoints of these functionals exist. As usual, in the semantics we have defined the operators μ and ν in terms of prefixed and postfixed points, since the least prefixed and the greatest postfixed point coincide with the least and greatest fixpoint of a monotonic functional [Sti01]. Regarding fixpoint iteration, *approximants* of $mv\text{-}\mathfrak{L}_\mu$ formulae are defined in the usual way: if $fp(Z) = \mu Z. \varphi$ then $Z^0 := \lambda s. \perp$, $Z^{\alpha+1} := \llbracket \varphi \rrbracket_{\rho[Z \mapsto Z^\alpha]}$ for any ordinal α and any environment ρ , and $Z^\lambda := \prod_{\alpha < \lambda} Z^\alpha$ for a limit ordinal λ . Dually, if $fp(Z) = \nu Z. \varphi$ then $Z^0 := \lambda s. \top$, $Z^{\alpha+1} := \llbracket \varphi \rrbracket_{\rho[Z \mapsto Z^\alpha]}$, and $Z^\lambda := \bigsqcup_{\alpha < \lambda} Z^\alpha$. The following theorem due to Tarski states that least and greatest fixpoints are indeed approximants.

Theorem 4.1 (Computation of Fixpoints, [Tar55]). *For all MMKS \mathcal{T} with state set \mathcal{S} there is an $\alpha \in \text{Ord}$ s.t. for all $s \in \mathcal{S}$ we have: if $\llbracket \eta Z. \varphi \rrbracket_\rho(s) = x$ then $Z^\alpha(s) = x$.*

In summary, our multi-valued modal μ -calculus exhibits all the general properties known from the standard μ -calculus. It is a modal logic where action labeled paths can be considered. Its semantics exhibits the expected dualities between the operators and fulfills our expectations of a useful fixpoint logic: least and greatest fixpoints always exist and coincide with the least and upper bounds of the respective approximants.

MMKS vs. PF-LTS: Connection between the Two Kinds of Structures

An MMKS over a powerset lattice $(\mathcal{P}(\mathcal{S}), \sqsubseteq)$ (as we have introduced it in this chapter) is the appropriate structure to reason about software product family properties using $mv\text{-}\mathfrak{L}_\mu$. In contrast to an MMKS, in Chapter 3.2.3 we have seen that

²Let the auxiliary relation $\preceq \subseteq (\mathcal{S} \times \mathcal{L}) \times (\mathcal{S} \times \mathcal{L})$ extend the lattice ordering \sqsubseteq to functions $g, h : \mathcal{S} \rightarrow \mathcal{L}$ by pointwise application, i.e. $\forall s \in \mathcal{S} : g \preceq h$ iff $g(s) \sqsubseteq h(s)$. Then, a functional $f \in \text{FUNCTIONALS}$ is monotonic w. r. t. \sqsubseteq iff $g \preceq h \Rightarrow f(g) \preceq f(h)$.

the configured-transitions semantics of a PF-CCS program is given as a PF-LTS—which is formally not quite a MMKS. Obviously, despite their slight differences, both structures can equivalently be used to represent the semantics of a PF-CCS product family. In the following we show how one structure can be transformed into the other one, and thus establish the connection between both. Ultimately, this allows to use formulae of $mv\text{-}\mathcal{L}_\mu$ in order to reason about the properties of a PF-CCS software product family.

Essentially, the only difference between both kinds of structures is that a MMKS additionally holds the information to what degree atomic propositions hold in individual states. Consequently, it is a simple matter to translate (on-the-fly) a PF-LTS transition system (cf. Definition 3.18, Page 140) which represents the configured-transitions semantics of a PF-CCS product family into a corresponding MMKS. For a PF-LTS $(\mathcal{S}, \mathcal{A}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, \sigma)$ the corresponding MMKS is given as $(\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L)$. The set of states \mathcal{S} , the transition relations \mathcal{R}_α , and the corresponding actions in \mathcal{A} are identical for the PF-LTS and the MMKS structure. The state labeling function L can not be derived from the respective PF-LTS, since so far, PF-CCS programs do not support propositional constants. Thus, when deriving a MMKS from a PF-LTS system, the validity of atomic propositions in individual states, i.e. the function L itself, has to be defined explicitly. However, we can also specify reasonable $mv\text{-}\mathcal{L}_\mu$ -formulae without making use of the values of atomic propositions in the respective formulae. Such properties consider only the action traces of the PF-LTS system and involve the basic constants *true* and *false*. We will see some examples of such properties in Section 4.3.

For the other direction, when a MMKS \mathcal{T} with an explicit state labeling function L is given, we can easily derive a consistent state labeling of the corresponding PF-LTS \mathcal{Q} by labeling the states according to

$$\theta \in L(s)(q) \iff q \in L_{\mathcal{Q}}(s)$$

where we use the function $L_{\mathcal{Q}} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{P})$ to label the states of the PF-LTS, as done for regular Kripke structures (cf. Appendix D). Thus, also the way in which states are labeled in both structures can be easily kept consistent.

For the remainder of this chapter, whenever we evaluate a $mv\text{-}\mathcal{L}_\mu$ -formula on a PF-LTS, we mean the evaluation of the formula on the MMKS constructed from the respective PF-LTS according to the construction described before. Concepts, such as for example projection (cf. Definition 3.20 on Page 142), which we have defined for a PF-LTS, are defined in an equivalent way also for a MMKS. In particular, from now on we do not distinguish between a PF-LTS and the corresponding MMKS for the verification of $mv\text{-}\mathcal{L}_\mu$ -formulae, anymore.

4.1.3. Correctness of the Provided Semantics

The semantics which we have defined for $mv\text{-}\mathfrak{L}_\mu$ yields a lattice element when evaluating it in a certain state. In the concrete case of the lattice $(\mathcal{P}(\text{CONFIGS}), \subseteq)$ this lattice element represents the set of all configurations which fulfill the specified property. In the following we elaborate on the correctness of our logic, i.e. we show that the semantics of a $mv\text{-}\mathfrak{L}_\mu$ -formula interpreted over a powerset lattice of configurations actually represents the result which we intuitively expect. As we have motivated in the last chapter, the intuitive correctness of our approach is based on relating a PF-CCS software product family—respectively its products—to a corresponding set of standalone CCS programs. Recall, that this intuitive meaning of a PF-CCS program is represented by the flat semantics (Definition 3.11, Chapter 3.2.1). Given this idea of correctness, we now show that the result of checking a $mv\text{-}\mathfrak{L}_\mu$ -formula on a PF-CCS product family actually conforms to the results which we expect when checking an equivalent μ -calculus formula on the respective CCS systems directly. More precisely, if a specific configuration θ is contained in the set of configurations for which a $mv\text{-}\mathfrak{L}_\mu$ -formula φ holds for a specific PF-CCS product family *Prog*, then the same property φ (now expressed as a formula in the standard μ -calculus) also holds for that CCS program which represents the product for configuration θ . This means that the semantics which we have defined for our multi-valued modal μ -calculus is indeed suitable for checking the different configurations of a PF-CCS program, since it coincides with the two-valued yes/no results we would obtain when checking the same property for every (standalone) system independently. In particular, this is the foundation for the correctness of any model checking algorithm for our multi-valued μ -calculus.

This fundamental connection between the result of a $mv\text{-}\mathfrak{L}_\mu$ and a \mathfrak{L}_μ formula interpreted over a product family and a standalone CCS system is formalized in Theorem 4.2. However, the formulation of this theorem requires some preparation, which we will provide in the following. A substantial part thereof is summarized in Lemma 4.1, which relates $mv\text{-}\mathfrak{L}_\mu$ -properties of a MMKS \mathcal{T} (representing a product family) to \mathfrak{L}_μ -properties of complete projections of \mathcal{T} (representing concrete products).

Lemma 4.1 formally involves the two logics $mv\text{-}\mathfrak{L}_\mu$ and \mathfrak{L}_μ , where \mathfrak{L}_μ denotes the standard μ -calculus. As we have already discussed in the preceding part of this chapter, both logics have the same syntax, but have different semantics. In order to distinguish clearly between both logics we decorate the symbol \models_μ with the index μ to denote the satisfaction relation of modal μ -calculus formulae in \mathfrak{L}_μ . Since formulae of \mathfrak{L}_μ and $mv\text{-}\mathfrak{L}_\mu$ have the same syntax we can express a property with the same formulae φ in both logics.

Syntactically, both logics make use of the same set of variables \mathcal{V} . Semantically, the interpretation of free variables is formally fixed by two different variable environments ρ and V , where ρ denotes the variable environment for the interpretation

of $mv\text{-}\mathfrak{L}_\mu$ formulae over a MMKS, while V denotes an environment used for the interpretation of \mathfrak{L}_μ formulae over a corresponding PF-LTS (cf. Appendix D for details on V). Certainly, only if both environments define the interpretations of identical variables in a consistent way, we can expect that the same formula yields a consistent result when being interpreted according to both semantics. The following definition makes the idea of consistent variable environments more precise. Since the consistency of environments is an auxiliary concept which we need for Lemma 4.1, where we evaluate a formula on both a MMKS and a complete projection of the corresponding PF-LTS, we introduce the following definition for the special case where both the MMKS and the corresponding PF-LTS share the same set \mathcal{S} of states.

Definition 4.2 (Consistency of Variable Environments). *Let $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L)$ be a MMKS over which $mv\text{-}\mathfrak{L}_\mu$ formulae are interpreted with respect to a variable environment $\rho : \mathcal{V} \rightarrow (\mathcal{S} \rightarrow \mathcal{L})$, and $\mathcal{Q} = (\mathcal{S}, \mathcal{A}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\})$ be the PF-LTS constructed according to \mathcal{T} , over which \mathfrak{L}_μ formulae are interpreted with respect to a variable environment $V : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{S})$. Regarding the interpretation of the same formula φ over both structures \mathcal{T} and \mathcal{Q} in the respective logic we call both environments consistent iff*

$$\forall Z \in \mathcal{V}, s \in \mathcal{S} : s \in V(Z) \iff \theta \in \rho(Z)(s)$$

The idea of consistent variable environments states that if in a variable environment ρ , a proposition p holds in a state s of the structure \mathcal{T} for a configuration θ , then this state s also has to be labeled with p in the structure \mathcal{Q} by the environment V . Definition 4.2 directly implies a construction rule for an environment V given an environment ρ , if V is constructed according to

$$\forall Z \in \mathcal{V} : V(Z) = \{s \in \mathcal{S} \mid \theta \in \rho(Z)(s)\}$$

Note that—unless explicitly respecified—the variable environment V specified for a PF-LTS \mathcal{Q} also holds for all projections $\Pi_\theta(\mathcal{Q})$. Thus, if for a system \mathcal{Q} and a variable Z we have $s \in V(Z)$, then $s \in V(Z)$ holds also in any projected system $\Pi_\theta(\mathcal{Q})$ in which the state s is preserved according to the configuration θ .

Note that we specify Theorem 4.2 and Lemma 4.1 for the specific case where the underlying lattice is the powerset lattice $(\mathcal{P}(\{R, L, ?\}^n), \subseteq)$ over the set of configurations $\{R, L, ?\}^n$. Further, as a notational easement, we assume that the transitions in \mathcal{T} are labeled with sets of *complete* configuration vectors, only. Thus, no configuration vector contains an ?-entry anymore, and any lattice element represents a set of complete configurations. Recall, that any set of incomplete configuration labels can uniquely be denoted as an equivalent set of complete configuration vectors (cf. Page 147). In particular, this is just a technical issue and does not change the meaning of a configuration label.

4. Verifying Properties of PF-CCS Software Product Families

With these premises we now introduce Lemma 4.1. It relates properties of a product family represented as a MMKS/PF-LTS \mathcal{T} to properties of its products (given as complete projections of \mathcal{T}). Essentially, it states that whenever \mathcal{T} fulfills a $mv\text{-}\mathfrak{L}_\mu$ -property φ for a certain configuration θ in a state s , then the product which can be derived from \mathcal{T} by projection to θ also fulfills the property φ (now interpreted as a formula in \mathfrak{L}_μ) in its corresponding state s .

Lemma 4.1 (Property Equivalence of a Projected Family and a Product). *Let $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L)$ be a MMKS with labeling function L constructed according to the PF-LTS $\mathcal{Q} = (\mathcal{S}, \mathcal{A}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, \sigma)$. Further, let ρ and V_θ be consistent variable environments for \mathcal{T} and each $\Pi_\theta(\mathcal{Q})$, respectively. Then, for all fitting and complete configurations $\theta \in \text{CONFIGS}$, states s of $\Pi_\theta(\mathcal{Q})$, and formulae φ we observe*

$$\Pi_\theta(\mathcal{Q}).s \models_\mu^{V_\theta} \varphi \quad \text{iff} \quad \theta \in \llbracket \varphi \rrbracket_\rho^\mathcal{T}.s$$

Proof. By structural induction on the formula.

- Base cases: $\varphi = \text{true}, \text{false}, q, Z$
 According to the semantics of $mv\text{-}\mathfrak{L}_\mu$, for all states s we have $\llbracket \text{true} \rrbracket_\rho^\mathcal{T}.s = \text{CONFIGS}$, and thus all configurations θ fulfill the formula *true* in every state. Regarding the projection, since per definition $\tilde{s} \models_\mu^{V_\theta} \text{true}$ for any state $\tilde{s} \in \mathcal{S}_\mathcal{Q}$, also all states of every projected PF-LTS $\Pi_\theta(\mathcal{Q})$ fulfill the formula *true* for any θ . For the formula *false*, an analogue argument shows that for any configuration θ , no state ever fulfills *false* in either of the two structures. Regarding atomic propositions, according to the semantics a proposition $q \in \mathcal{P}$ holds in a state s in all configurations $\theta \in L(s)(q)$. According to the consistent construction of the labeling function for both structures, this means that q is in the set of propositions that hold in state s of \mathcal{Q} (in every configuration θ). Thus, according the definition of $\models_\mu^{V_\theta}$, every state s also fulfills $s \models_\mu^{V_\theta} \varphi$ in the corresponding projection to θ . Likewise, in all other configurations $\theta \notin L(s)(q)$, the proposition q does not hold in a state s , and thus $s \not\models_\mu^{V_\theta} q$. Regarding any free variable Z , our semantics yields $\theta \in \rho(Z)(s)$. According to the construction of consistent variable environments this directly implies that $s \in V_\theta(Z)$ for all θ . This is exactly the definition of $\models_\mu^{V_\theta}$ and directly yields $\Pi_\theta(\mathcal{Q}).s \models_\mu^{V_\theta} \varphi$.

- Inductive step:

– Formulae $\neg q$:

$$\text{To show: } \Pi_\theta(\mathcal{Q}).s \models_\mu \neg \varphi \quad \text{iff} \quad \theta \in \llbracket \neg \varphi \rrbracket_\rho^\mathcal{T}.s$$

$$\text{Ind. hypothesis: } \Pi_\theta(\mathcal{Q}).s \models_\mu \varphi \quad \text{iff} \quad \theta \in \llbracket \varphi \rrbracket_\rho^\mathcal{T}.s$$

Let $M := \llbracket q \rrbracket_\rho^\mathcal{T}.s$. Applying the semantics yields $\llbracket \neg q \rrbracket_\rho^\mathcal{T}.s = M^c$. Thus $\neg q$ holds for all configurations $\eta \notin M$. Since per induction hypothesis we can assume that $\Pi_\theta(\llbracket \text{Prog} \rrbracket_{CT}).s \models_\mu q$ for all configurations $\theta \in M$, we conclude that $\Pi_\eta(\llbracket \text{Prog} \rrbracket_{CT}).s \not\models_\mu q$ for all configurations $\eta \notin M$. Since $s \not\models_\mu \varphi$ is equal to $s \models_\mu \neg \varphi$, we have $\Pi_\eta(\llbracket \text{Prog} \rrbracket_{CT}).s \models_\mu \neg \varphi$ iff $\eta \in M^c$.

- Formulae $\varphi \wedge \psi$:

To show: $\Pi_\theta(\mathcal{Q}).s \models_\mu \varphi \wedge \psi$ iff $\theta \in \llbracket \varphi \wedge \psi \rrbracket_\rho^T.s$

Ind. hypothesis: $\Pi_\theta(\mathcal{Q}).s \models_\mu \varphi$ iff $\theta \in \llbracket \varphi \rrbracket_\rho^T.s$

and $\Pi_\theta(\mathcal{Q}).s \models_\mu \psi$ iff $\theta \in \llbracket \psi \rrbracket_\rho^T.s$

According to the semantics of $mv\text{-}\mathfrak{L}_\mu$, $\theta \in \llbracket \varphi \wedge \psi \rrbracket_\rho^T.s$ means that $\theta \in (\llbracket \varphi \rrbracket_\rho^T.s \cap \llbracket \psi \rrbracket_\rho^T.s)$. In order to be in the intersection, θ must be part of all subsets, i.e. $\theta \in \llbracket \varphi \rrbracket_\rho^T.s$ and $\theta \in \llbracket \psi \rrbracket_\rho^T.s$. Thus, according to the induction hypothesis, $\Pi_\theta(\mathcal{Q}).s \models_\mu \varphi$ and $\Pi_\theta(\mathcal{Q}).s \models_\mu \psi$. According to the semantics of \wedge in the μ -calculus we can conclude that $\Pi_\theta(\mathcal{Q}).s \models_\mu \varphi \wedge \psi$.

- The case for formulae $\varphi \vee \psi$ is shown similarly.

- Formulae $\langle \alpha \rangle \varphi$:

To show: $\Pi_\theta(\mathcal{Q}).s \models_\mu \langle \alpha \rangle \varphi$ iff $\theta \in \llbracket \langle \alpha \rangle \varphi \rrbracket_\rho^T.s$

Ind. hypothesis: $\Pi_\theta(\mathcal{Q}).s' \models_\mu \varphi$ iff $\theta \in \llbracket \varphi \rrbracket_\rho^T.s'$, where the state s' is an α -successor of s .

Applying the semantics yields $\llbracket \langle \alpha \rangle \varphi \rrbracket_\rho^T = \lambda s. \bigcup_{s' \in S} \{\mathcal{R}_\alpha(s, s') \cap \llbracket \varphi \rrbracket_\rho^T(s')\}$. If we assume that $\theta \in \llbracket \langle \alpha \rangle \varphi \rrbracket_\rho^T.s$ (the premise of what we want to show), θ must be at least in one of the intersections $\mathcal{R}_\alpha(s, s') \cap \llbracket \varphi \rrbracket_\rho^T.s'$. According to the induction hypothesis $\theta \in \llbracket \varphi \rrbracket_\rho^T.s'$. Since θ is in the intersection and in the right subset $\llbracket \varphi \rrbracket_\rho^T.s'$, we can conclude that $\theta \in \mathcal{R}_\alpha(s, s')$ for at least one transition. Together with the induction hypothesis $\Pi_\theta(\mathcal{Q}).s' \models_\mu \varphi$ we can conclude that there exists an α -transition to a state s . This matches exactly the requirements of the semantics of diamond operator in the μ -calculus, which implies that $\Pi_\theta(\mathcal{Q}).s \models_\mu \langle \alpha \rangle \varphi$. For the situation where $\theta \in \llbracket \langle \alpha \rangle \varphi \rrbracket_\rho^T.s$ does not hold, following the same argumentation we see that θ is not in the intersection and thus $\theta \notin \mathcal{R}_\alpha(s, s')$. Thus, in this case $\Pi_\theta(\mathcal{Q}).s \not\models_\mu \langle \alpha \rangle \varphi$.

- The case for formulae $[\alpha] \varphi$ is shown similarly.

- Formulae $\mu Z.\varphi$:

To show: $\Pi_\theta(\mathcal{Q}).s \models_\mu \mu Z.\varphi$ iff $\theta \in \llbracket \mu Z.\varphi \rrbracket_\rho^T.s$

Ind. hypothesis: $s \in S$ where ${}_\mu \llbracket \varphi \rrbracket_{V[Z \rightarrow S]}^{\Pi_\theta(\mathcal{Q})} \subseteq S$

iff $\theta \in f.s$ where $\llbracket \varphi \rrbracket_{\rho[Z \rightarrow f]}^T.s \subseteq f.s$

Recall, that ${}_\mu \llbracket \varphi \rrbracket_V$ denotes the semantics of φ in the μ -calculus for environment V . According to our semantics, $\theta \in \bigcap \{f \mid \llbracket \varphi \rrbracket_{\rho[Z \rightarrow f]} \subseteq f\}.s$ ³. Since θ is in the intersection, it is part of any such f , which means that for all f with $\llbracket \varphi \rrbracket_{\rho[Z \rightarrow f]}^T.s \subseteq f.s$ we have that $\theta \in f.s$. Thus, according to the induction hypothesis for every such s we can conclude

³The expression $\{f \mid \llbracket \varphi \rrbracket_{\rho[Z \rightarrow f]} \subseteq f\}.s$ is an abbreviation for the set $\{f.s \mid \llbracket \varphi \rrbracket_{\rho[Z \rightarrow f]}^T.s \subseteq f.s\}$.

4. Verifying Properties of PF-CCS Software Product Families

that $s \in S$ where $\mu \llbracket \varphi \rrbracket_{V[Z \mapsto S]} \subseteq S$ in the structure $\Pi_\theta(\mathcal{Q})$. Since state s is in every such set S , it is also part of the intersection of all S , i.e. $s \in \bigcap \{S \mid \mu \llbracket \varphi \rrbracket_{V[Z \mapsto S]} \subseteq S\}$. According to the definition of the fixpoint operator μ in the standard μ -calculus this is equivalent to the desired result $\Pi_\theta(\mathcal{Q}).s \models_\mu \mu Z.\varphi$.

– The case $\nu Z.\varphi$ for the greatest fixpoint is shown similarly.

□

Now, we can formulate the central result of this section in the following theorem. Essentially, Theorem 4.2 states that the result of evaluating a property φ according to the multi-valued μ -calculus semantics on a PF-CCS model of a product family agrees with the separate evaluation according to the standard μ -calculus on the corresponding CCS systems, respectively. Ultimately, this justifies the semantics of our multi-valued modal μ -calculus. Recall, that the expression $\mathcal{Q} \models_\mu \varphi$ means that the transition system \mathcal{Q} fulfills property φ in its start state.

Theorem 4.2 (Correctness of the Multi-Valued μ -Calculus Semantics). *Let $Prog = (\mathcal{E}, P_1)$ be a PF-CCS program, and $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L, p_1)$ be a MMKS constructed according to the PF-LTS $\llbracket Prog \rrbracket_{CT}$, where the start state p_1 corresponds to the start process P_1 of $Prog$. Further, let ρ and V_θ be consistent variable environments as defined in Lemma 4.1. For all such PF-CCS programs $Prog$, complete configurations θ , and formulae φ*

$$\llbracket config(Prog, \theta) \rrbracket_{CCS} \models_\mu^{V_\theta} \varphi \quad \text{iff} \quad \theta \in \llbracket \varphi \rrbracket_\rho^{\mathcal{T}}.p_1$$

Proof. According to Lemma 4.1, for any state $s \in \Pi_\theta(\llbracket Prog \rrbracket_{CT})$ the results of evaluating a formula φ on a product family $\llbracket Prog \rrbracket_{CT}$ and of evaluating the same formulae φ on a single product $\Pi_\theta(\llbracket Prog \rrbracket_{CT})$ which is derived according to the complete configuration θ are consistent, i.e.

$$\Pi_\theta(\llbracket Prog \rrbracket_{CT}).s \models_\mu^{V_\theta} \varphi \quad \text{iff} \quad \theta \in \llbracket \varphi \rrbracket_\rho^{\mathcal{T}}.s$$

In particular, this holds also for the start state p_1 . Theorem 3.2 (Page 145) shows that every derived product $\Pi_\theta(\llbracket Prog \rrbracket_{CT})$ is (strongly) bisimilar to the corresponding CCS program $\llbracket config(Prog, \theta) \rrbracket_{CCS}$ for the same configuration θ . According to [BS01a], if two states (respectively processes) are (strongly) bisimilar, then they satisfy exactly the same \mathfrak{L}_μ -formulae. Thus, $\llbracket config(Prog, \theta) \rrbracket_{CCS} \models_\mu^{V_\theta} \varphi$ iff $\theta \in \llbracket \varphi \rrbracket_\rho^{\mathcal{T}}.p_1$. □

In essence, Theorem 4.2 establishes the fundamental result that the semantics of our logic $mv\text{-}\mathcal{L}_\mu$ indeed corresponds to our intuitive expectations which we have for the result of the evaluation of a formula on a PF-CCS product family. In particular, the semantics of $mv\text{-}\mathcal{L}_\mu$ is correct in the sense that we can likewise check properties on the model of the product family or on the individual products directly. In any case, the evaluations yield consistent results. This is the indispensable basis for the application of model checking techniques. From this point of view, Theorem 4.2 establishes also a kind of fundamental correctness of model checking.

4.2. Model Checking

So far, we have introduced an appropriate logic, $mv\text{-}\mathcal{L}_\mu$, for the specification of properties of a software product family which is given as a multi-valued labeled transition system (PF-LTS). We have defined the meaning of $mv\text{-}\mathcal{L}_\mu$ -formulae and shown the (intuitive) correctness of our semantics. As this is sufficient for a pure theoretician, for the practitioner the question that remains is how to compute the satisfaction value of a $mv\text{-}\mathcal{L}_\mu$ formula over a PF-LTS in a practically feasible way. Certainly, while Theorem 4.1 (Page 170) implies a way for computing the satisfaction value of an $mv\text{-}\mathcal{L}_\mu$ -formula and a given MMKS based on applying the semantics definition, the associated fixpoint computation is typically expensive as it is based on the construction of subsequent approximants in a naive brute force fashion. In particular for complex systems this fixpoint iteration is not practicable. For evaluating $mv\text{-}\mathcal{L}_\mu$ formulae, *model checking* offers a more suitable solution. In fact, PF-CCS and $mv\text{-}\mathcal{L}_\mu$ are actually both designed for the application with model checking.

Model checking [CGP99, BK08] is an automatic verification technique which was pioneered by Clarke and Emerson [EC81, CE81b, CES86], and Queille and Sifakis [QS82] in the early 1980s. Until this day, model checking has evolved to become a serious and promising verification technique for verifying complex software systems in an automatic fashion. In 2007, Clarke, Emerson and Sifakis even received the Turing Award for their work on model checking.

Model checking is an algorithmic approach to solve the question whether a system S satisfies a property φ w. r. t. one of its states s . Usually, the system S is given as a Kripke structure and the property φ is specified using a temporal logic such as LTL, CTL, or the standard modal μ -calculus. Typically, the result of model checking is a *true/false*-answer, indicating whether φ holds in state s or not. We refer to this kind of model checking as *standard* or *two-valued* model checking.

However, as the definition of the semantics of $mv\text{-}\mathcal{L}_\mu$ shows, standard model checking algorithms are not applicable for multi-valued structures like a PF-LTS. In the

4. Verifying Properties of PF-CCS Software Product Families

context of multi-valued systems, another branch of model checking is of particular interest: *multi-valued model checking*. Multi-valued model checking is tailored to multi-valued structures which represent partial information, such as for example our PF-LTS structures. In particular, evaluating a $mv\text{-}\mathcal{L}_\mu$ formula φ on a PF-LTS—and thus finding the set of configurations of a PF-CCS product family that fulfill a certain multi-valued μ -calculus property φ —can be done using existing multi-valued model checking techniques and algorithms.

In our concrete situation we have chosen the multi-valued, game-based model checking approach introduced by Shoham and Grumberg [SG05] for the evaluation of $mv\text{-}\mathcal{L}_\mu$ formulae on a given MMKS structure. First and foremost, because this approach is designed for multi-valued Kripke structures defined over a lattice, which are almost like our PF-LTS. In fact, a slight adaption of [SG05] yields a game-based approach for our multi-valued modal μ -calculus. Secondly, in particular the realization as a game-based approach—originating from the work by [EJS93] and [Sti95]—represents an efficient model checking technique which additionally allows model checking in an *on-the-fly* or *local* fashion. In contrast to other so-called *reduction* approaches, e.g. [BG04], in which a multi-valued problem is solved based on the evaluation of several two-valued problems, the approach of [SG05] checks the property on the multi-valued structure directly, and thus can provide auxiliary information (in terms of the multi-valued structure itself) that explains its result. In particular for our scenario of PF-CCS product families these auxiliary information can lead to the variation point whose configuration selection has caused the property to fail. As the game-based model checking approach of [SG05] is not a contribution of this thesis, we do not reproduce [SG05] in its full detail, here. Instead, we just give an overview of the approach and describe it to a level where the necessary adaptations which have to be made to fit to our approach can be understood.

Adjustments of the Game-Based Approach of Shoham/Grumberg [SG05] to fit the Setting of PF-CCS Product Families

In the context of the multi-valued μ -calculus, the game-based setting becomes technically more involved compared to its two-valued counterpart. Nevertheless, the essence of the game-based approach of computing a satisfaction value based on the so-called *game graph* is similar in the multi-valued and the two-valued situation. Like a two-valued model checking game, a multi-valued model checking game of [SG05] is also played by two players, the *verifier* and the *refuter*. A *play* of the game $\Gamma_{\mathcal{M}}(s_0, \varphi_0)$ is a maximal sequence of *play configurations*, which consist of a state s and a sub-formula φ denoted by $s \vdash \varphi$. A game is played according to rules which determine the moves that each player can make in a certain play configuration. The rules are applied depending on the current game configuration, i.e. the next move in a play yielding from a game configuration $s_i \vdash \varphi_i$ to a successor configuration $s_{i+1} \vdash \varphi_{i+1}$ is determined by the main connective of φ_i . A play ends

when no rule can be applied anymore. While in a two-valued model checking play the final play configuration (or the play configuration structure in the case of an infinite play) determines the winner of the game, in a multi-valued model checking play, the idea of winning a single play is not really applicable anymore, since the outcome of a single play is an element of the lattice and no longer a \top/\perp -result. Thus, the aim of the verifier in the multi-valued situation is no longer to win the play (as in the two-valued setting), but to “maximize” (in terms of the lattice) the value of the play instead. Similarly, the refuter tries to minimize the result of the play. In this light, the result of a multi-valued play can be seen as a measure for how close the verifier is to winning.

By adapting the multi-valued model checking game of [SG05] to action labeled transitions, we can directly use it as a model checking algorithm to compute the satisfaction value of a $mv\text{-}\mathfrak{L}_\mu$ formula in a state s for a given MMKS \mathcal{T} . For our purpose the game rules for the box and diamond operator are updated by the following two rules, which use action labeled versions of both operators:

$$\frac{s \vdash \langle \alpha \rangle \varphi}{s' \vdash \varphi} \text{ (verifier) : } \mathcal{R}_\alpha(s, s') \neq \perp \quad (4.1)$$

$$\frac{s \vdash [\alpha] \varphi}{s' \vdash \varphi} \text{ (refuter) : } \mathcal{R}_\alpha(s, s') \neq \perp \quad (4.2)$$

The first parts of the side conditions, i.e. the words *verifier* and *refuter*, indicate which player is allowed to make the move according to this rule. For example, Rule 4.1 describes a move of the verifier: if the current play configuration is $s \vdash [\alpha] \varphi$ the verifier makes the move according to this rule to a subsequent play configuration $s' \vdash \varphi$, if there is an α -transition from s to s' for which the transition value is greater than \perp (indicating that the respective transition actually exists). The rest of the multi-valued model checking game rules are exactly as the original ones described in [SG05].

In the two-valued situation, we say that a player has a *winning strategy* for a game if he can win any play when playing according to this strategy. In this context a strategy is simply a mapping which tells the player how to move for every play configuration in which it is his turn to move. Thus, a strategy specifies to which play configuration a player has to go next. In contrast to a two-valued game, the winning criteria in a multi-valued model checking game have to be adjusted, as the multi-valued game now yields an element of the lattice and no longer a true/false answer. The best game strategy for the verifier is that one which maximizes the results over all plays. In particular, the verifier is interested in a strategy which guarantees that the result of any play he plays will be equal or higher to the value

4. Verifying Properties of PF-CCS Software Product Families

of the strategy, i.e. he is interested in maximizing the greatest lower bound of all play results. Reflecting this idea, the value of the strategy for the verifier is defined as the greatest lower bound over all his plays. For the refuter, the situation is dual: he aims to play according to a strategy which keeps the result of all plays as low as possible in a guaranteed way. Thus, in the multi-valued setting the players are no longer seeking “winning” strategies, but rather strategies for gaining a (maximal or minimal) value.

The idea of strategies for gaining a value allows us to make the connection to model checking, and thus the satisfaction value of a $mv\text{-}\mathcal{L}_\mu$ formula. More precisely, the value of the game $\Gamma_{\mathcal{M}}(s_0, \varphi_0)$ determines the truth value of the $mv\text{-}\mathcal{L}_\mu$ formula φ_0 evaluated in state s_0 over the multi-valued structure \mathcal{M} , where the definition of the value of a game is directly adapted from [SG05]: The value of the multi-valued model checking game $\Gamma_{\mathcal{T}}(s, \varphi)$ is defined as the least upper bound over all strategy values for the verifier. The corresponding proof found in [SG05] for the correctness of a multi-valued model checking game immediately implies the correctness for multi-valued multi-valued games using the adjusted rules 4.1 and 4.2. Moreover, [SG05] also provides a way of solving a multi-valued model checking game based on the game graph which we can also use in our setting by simply adjusting the transition relation to use action labeled transitions. In summary, the game-based model checking approach of [SG05] can be directly used—with the minor adjustments described above—to compute the satisfaction value of a $mv\text{-}\mathcal{L}_\mu$ formula over a MMKS. This allows us to exploit all the advantages of a game-based model checking approach [Sti95] also for our multi-valued modal μ -calculus, and in particular provides a much more practicable way of solving the model checking problem for $mv\text{-}\mathcal{L}_\mu$ than the naive fixpoint iteration does.

4.3. Example: Verifying a Family of Windscreen Wipers

Let us now demonstrate our approach on a product line whose configurations realize different versions of a windscreen wiper system. Note that this is just a small example to make the reader familiar with PF-CCS specifications and the multi-valued modal μ -calculus.

4.3.1. Specification of the Product Family of Windscreen Wipers

At first, we specify the family of systems, using the PF-CCS formalism introduced in Chapter 3. As a convention, action names start with lowercase letters, while process identifiers start with uppercase letters. The windscreen wiper systems that we specify in our family *WipFam* are each built of two subcomponents: a rain sensor,

4.3. Example: Verifying a Family of Windscreen Wipers

Sensor, and a windscreen wiper, *Wiper*. Both subcomponents can be realized by two variants, a high and a low one, respectively:

$$WipFam \stackrel{def}{=} Sensor \parallel Wiper \quad (4.3)$$

$$Sensor \stackrel{def}{=} SensL \oplus_1 SensH \quad (4.4)$$

$$Wiper \stackrel{def}{=} WipL \oplus_2 WipH \quad (4.5)$$

The low variant *SensL* of the sensor is specified as follows:

$$SensL \stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{noRain}.SensL \quad (4.6)$$

$$Raining \stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{rain}.Raining \quad (4.7)$$

The low variant *SensL* only detects two different environmental conditions—dry and raining—even though the environment can stimulate the sensor with three different conditions: *hvy* for heavy rain, *ltl* for little rain and *non* for no rain. However, this sensor cannot differ between heavy and little rain, i.e. for this sensor, *hvy* and *ltl* have the same effect, as the sensor reaches a process *Raining* and provides an action \overline{rain} , indicating solely the fact that it is raining (without precisely characterizing the intensity). As long as no rain has been detected, the sensor provides the action \overline{noRain} , respectively.

The high version of the sensor can distinguish between different degrees of rain intensity, i.e. *SensH* additionally differentiates heavy rain from little rain. Its PF-CCS specification is given in the following:

$$SensH \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{noRain}.SensH \quad (4.8)$$

$$Medium \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{rain}.Medium \quad (4.9)$$

$$Heavy \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{hvyRain}.Heavy \quad (4.10)$$

In this product family, the sensors can be arbitrarily combined with two variants of windscreen wipers, *WipL* and *WipH*. In particular, for this example we have no additional non-functional dependencies between the possible variants which would restrict the set of combinatorially possible configurations.

The low version *WipL* offers two operation modes: (i) a manual mode with perpetual wiper arm movement (action *permWip*), which has to be activated explicitly by the

4. Verifying Properties of PF-CCS Software Product Families

driver, (ii) and a semi-automatic interval mode in which the wiper arm moves at a lower frequency triggered by the rain sensor (via the action *rain*).

$$WipL \stackrel{def}{=} off.WipL + permOn.Perm + intvOn.Interval \quad (4.11)$$

$$Interval \stackrel{def}{=} noRain.Interval + intvOff.WipL + intvOn.Interval \\ + rain.Wiping + hvyRain.Wiping \quad (4.12)$$

$$Wiping \stackrel{def}{=} \overline{slowWip}.Interval + intvOn.Interval \quad (4.13)$$

$$Perm \stackrel{def}{=} \overline{permWip}.Perm + off.WipL + intvOn.Interval \quad (4.14)$$

The high variant *WipH* can operate at two speeds: slow (action $\overline{slowWip}$) and fast (action $\overline{fastWip}$). Here, the wiper arm movement is fully controlled by the rain sensor and adjusts its frequency automatically to the current rain intensity.

$$WipH \stackrel{def}{=} off.WipH + intvOn.AutoIntv \quad (4.15)$$

$$AutoIntv \stackrel{def}{=} noRain.AutoIntv + intvOn.AutoIntv + rain.Slow \\ + intvOff.WipH + hvyRain.Fast \quad (4.16)$$

$$Slow \stackrel{def}{=} \overline{slowWip}.AutoIntv + intvOn.AutoIntv \quad (4.17)$$

$$Fast \stackrel{def}{=} \overline{fastWip}.AutoIntv + intvOn.AutoIntv \quad (4.18)$$

The PF-CCS program specifying the entire product line *WipFam* is given by the Equations 4.3–4.18. The whole program *WipFam* is well-formed, which allows a unique numbering of all (two) variation points as shown by Equations 4.4 and 4.5.

4.3.2. Verification

From our example system family *WipFam*, we can derive four products as we can combine the subsystem variants arbitrarily. Having specified the family in PF-CCS, we can now apply the verification approach described in this chapter in order to verify functional properties for the individual configurations in the product family. In the following we use $\langle \cdot \rangle$ and $[\cdot]$ to denote a successor state which can be reached using a transition labeled with an arbitrary action.

Thinking of a relevant property, for instance, one could possibly be interested in verifying for a windscreen wiping system whether or not a driver is always able to switch to automatic windscreen wiping mode.

$$\mu Z. \left(\langle intvOn \rangle true \vee \langle \cdot \rangle Z \right) \quad (4.19)$$

4.3. Example: Verifying a Family of Windscreen Wipers

Note that this property corresponds to the CTL formula scheme EFP which corresponds to the fixpoint characterization $EFP = \mu Z.P \vee EXZ$ in the modal μ -calculus.

Another property could demand the windscreen wiper to wipe fast, once it is raining heavily.

$$\nu Z. \left((\neg \langle \text{intvOff} \rangle \text{true} \vee [\text{hvy}] \langle \overline{\text{fastWip}} \rangle \text{true} \right) \wedge [.]Z \quad (4.20)$$

This property is of the kind $\nu Z.P \wedge AXZ$, and thus represents the fixpoint characterization of the CTL formula AGP . Note that both properties do not use atomic propositions, but still specify reasonable properties of the wiper family.

In order to evaluate whether the properties 4.19 and 4.20 hold for our exemplary product family we can use one of the two possibilities that we have introduced in this thesis, i.e. we can either

1. evaluate the formula directly by applying the semantics rules for $mv\text{-}\mathcal{L}_\mu$ -formulae (cf. Figure 4.2), or
2. apply the game-based model checking algorithm of Shoham and Grumberg [SG05] with the adjustments introduced in the previous Section 4.2.

Both possibilities are interpreted over the PF-LTS (and the corresponding MMKS) that represents the semantics of the family of wiper systems which is given by the PF-CCS program $(\mathcal{E}, \text{WipFam})$, where the set \mathcal{E} of process definitions is specified by the Equations 4.3–4.18. Recall that $[[\dots]]_{CT}$ represents the configured-transitions semantics of PF-CCS programs (cf. Chapter 3.2.3, Definition 3.19). Let

$$(\mathcal{S}, \mathcal{A}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, \sigma) := [[(\mathcal{E}, \text{WipFam})]]_{CT}$$

be that PF-LTS, and

$$(\mathcal{S}, \{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}, L) =: \mathcal{W}$$

be the corresponding MMKS (which we abbreviate as \mathcal{W}), where

- \mathcal{S} denotes the set of states,
- $\mathcal{A} = \{ \text{non}, \text{ltl}, \text{hvy}, \text{noRain}, \text{rain}, \text{hvyRain}, \text{off}, \text{permOn}, \text{intvOff}, \text{intvOn}, \text{slowWip}, \text{fastWip}, \text{permWip}, \overline{\text{non}}, \overline{\text{ltl}}, \dots \}$ represents the set of actions which are used in the PF-CCS program,
- $\{\mathcal{R}_\alpha \mid \alpha \in \mathcal{A}\}$ is the family of transition relations,
- σ is the start state, and
- L is the labeling function that associates for every state in \mathcal{S} the respective lattice values to each atomic proposition. Since no atomic propositions exist in the Formulae 4.19 and 4.20, the labeling function can be an arbitrary function as it is not relevant for the following interpretation.

4. Verifying Properties of PF-CCS Software Product Families

For the four possible configurations $\langle LL \rangle$, $\langle LR \rangle$, $\langle RL \rangle$, and $\langle RR \rangle$, the lattice \mathcal{L} is given as the powerset lattice $\left(\mathcal{P}(\{\langle LL \rangle, \langle LR \rangle, \langle RL \rangle, \langle RR \rangle\}), \subseteq \right)$ over the set of these configurations.

Evaluation of the Properties

The first way of evaluating the property is to perform the semantics rules which we have defined for the multi-valued modal μ -calculus in Figure 4.2 (Page 168) in the start state σ of the corresponding MMKS \mathcal{W} . For the evaluation of property 4.19 this means to compute

$$\llbracket \mu Z. (\langle \text{intvOn} \rangle \text{true} \vee \langle \cdot \rangle Z) \rrbracket_{\rho}^{\mathcal{W}} . \sigma \quad (4.21)$$

where the variable environment ρ is empty for this property since the formula contains no free variables. Computing the semantics of this property means to apply the corresponding semantics rules for the $mv\text{-}\mathcal{L}_{\mu}$ which are given in Figure 4.2. Since the outermost operator is the least fixpoint operator μ the computation directly requires to perform a corresponding fixpoint iteration according to the second last rule of Figure 4.2. The fixpoint iteration is performed as usual, i.e. starting with the least defined function (that one mapping all states, especially the start state σ , to the least lattice element \emptyset) for the state σ , computing the value, reapplying the definition again with the resulting value, and so on, until the the resulting lattice element (i.e. the set of configurations) does not change anymore between consecutive iterations. Since due to Theorem 4.1 the fixpoint is guaranteed to be reached, we omit to present the entire computation here. Finally, the result of computing the semantics of property 4.21 is the lattice element $\{\langle LL \rangle, \langle LR \rangle, \langle RL \rangle, \langle RR \rangle\}$. The configurations which are elements of this lattice element represent those products of the wiper family in that the desired property 4.19 holds. Thus, in the particular case of property 4.19, all derivable wiper variants fulfill the property. This result matches what we have intuitively expected by inspecting the PF-CCS product family: in every state of the corresponding MMKS \mathcal{W} the action *intvOn* is performable.

The Property 4.20 can be checked in an analogous way as Property 4.19, i.e. by computing the semantics

$$\llbracket \nu Z. (\neg \langle \text{intvOff} \rangle \text{true} \vee [hvy] \overline{\langle \text{fastWip} \rangle} \text{true}) \wedge [\cdot] Z \rrbracket_{\rho}^{\mathcal{W}} . \sigma$$

The result of the corresponding fixpoint iteration yields the lattice element $\{\langle RR \rangle\}$. This means that property 4.20 is only satisfied in the configuration in which the high variants of both subsystems are used. Intuitively, it is easy to see why: as the low version of the windscreen wiper does not provide a fast wiping mode, it never provides the output action $\overline{\text{fastWip}}$. Thus, the wind screen wiper can never wipe fast

if the low version is used, which means that the configurations $\{\langle LL \rangle\}$ and $\{\langle RL \rangle\}$ do not fulfill the property. However, if the high version of the windscreen wiper is used, but combined with the low version of the rain sensor (this corresponds to the configuration $\langle LR \rangle$), the property is not satisfied, either, since the sensor (low version) is not able to provide the output action $\overline{hvyRain}$, which would trigger the wiper to wipe fast.

As an alternative to the evaluation of the properties by means of applying the semantics rules of $mv\text{-}\mathcal{L}_\mu$, we can use the algorithm defined by Shoham and Grumberg [SG05] with the adjustments that we have described in Section 4.2. For example for evaluating Property 4.19 this requires to play the corresponding game

$$\Gamma_{\mathcal{S}}(\sigma, \mu Z.(\langle intvOn \rangle true \vee \langle \cdot \rangle Z))$$

where σ and \mathcal{S} denote the MMKS and the start state of it, as defined above. The final *value* of the game, which is reached in that configuration of the game in which the current player can apply no more play rule and thus make no more move, is the same set of configurations that we have obtained by evaluating the property using the $mv\text{-}\mathcal{L}_\mu$ semantics. As far as we know there is no implementation of the game-based algorithm for an existing model checker, which means that so far the *value* of the game has to be computed by playing the game according to the defined rules (cf. [SG05] and Section 4.2) manually. We omit to present this play here, as the original algorithm is not our contribution.

4.4. Related Work

Clearly, the most important related approach to the verification formalism which we have presented in this chapter is the game-based, multi-valued model checking approach by Shoham and Grumberg [SG05], which we have already introduced in detail in this chapter. In summary, in [SG05], Shoham and Grumberg extend the game-based framework of μ -calculus model checking to the multi-valued setting. They define a new model checking game that is equivalent to the problem of determining the value of a multi-valued μ -calculus formula over a multi-valued Kripke structure directly. In addition, they derive a direct model checking algorithm to solve this kind of games. The algorithm handles the underlying multi-valued structure directly without any reduction. Moreover, Shoham and Grumberg explore the connection between their multi-valued game-based approach and the established automata-based approach. As we explain the relevant details of their approach in detail in the scope of this chapter, we do not provide a more detailed summary here.

In fact, our version of the multi-valued μ -calculus was entirely influenced by the work of Shoham and Grumberg in the sense that we have defined the logic as well

4. Verifying Properties of PF-CCS Software Product Families

as the semantics in a more general way than necessary, in order to fit the (general) concepts and setting of Shoham and Grumberg’s approach directly. As we have seen, this allows to apply the game-based model checking algorithm of Shoham and Grumberg (with only minimal adjustments) in order to compute the values of $mv\text{-}\mathcal{L}_\mu$ -formulae. In this light our approach is a specialization and adaption of Shoham and Grumberg’s approach for the special case of PF-CCS product families.

The approach of Shoham and Grumberg bases on the work of Bruns and Godefroid [BG04], in which they explore model checking for multi-valued logics in general. Bruns and Godefroid show how to reduce multi-valued model checking to standard two-valued model checking, and present an automata-theoretic algorithm for multi-valued model checking with logics as expressive as the modal μ -calculus. As the game-based algorithm of Shoham and Grumberg, also the algorithm of Bruns and Godefroid can likewise be applied to evaluate multi-valued μ -calculus formulae defined over PF-CCS product families.

De Alfaro et al. introduce in [dAFH⁺05] a model checking approach to a so-called *discounted logic* DCTL. The logic DCTL is a generalization of CTL in which the predicates and the operators have a quantitative interpretation, i.e. where in every state the value of a formula is a real number in the interval $[0, 1]$. DCTL provides the default operators of CTL, where the interpretation of the operators is adjusted to the quantitative setting. Additionally, all operators can be discounted by a parameter which allows to change the weight (value) of the interpretation in states that are closer to the beginning of a path. While the fundamental idea, and the implementation of the operators is close to our approach, the discounted approach aims at the analysis of stochastic systems, and not at the application for product families. Beside an interpretation over transition systems, DCTL can additionally be interpreted over Markov chains, and Markov decision processes, where both a *path semantics*, and a *fixpoint semantics*—similar to the semantics of $mv\text{-}\mathcal{L}_\mu$ —are introduced. For both semantics, model checking algorithms are introduced, respectively.

Thrane, Fahrenberg and Larsen [LFT09] extend the usual notion of Kripke structures with a weighted transition relation. Based on this extended notion they provide a general framework for the analysis of quantitative and qualitative properties of reactive systems. They apply the framework to check the equivalence between an implementation of a system and its specification, while they express the equivalence not as *yes/no*-result in the classical sense, but as a *real-valued distance* which measures to which degree the implementation is equivalent/different to the specification. They provide different kinds of distances and discuss their computations. Their quantitative framework is applied to *implementation verification* for *weighted timed automata*. Similar to the work of de Alfaro et al. also Thrane et al. do not use the weights on transitions as a means to differentiate between the various configurations of a product family of transition systems, which classifies their approach into a different area of application.

In the shortpaper [PS08], Post and Sinz sketch a general idea for the verification of software product lines, which the authors call *lifting*. Essentially, lifting means to convert all variants of a software product line into a meta-program, which—similarly to our notion of a PF-CCS product family—represents a configuration-aware, integrated model of all products. The basic idea is to perform verification techniques on the meta-program rather than deriving and verifying each of the possible programs separately. The authors demonstrate the lifting technique by checking “configuration dependent hazards for the Linux kernel”, where the SAT-based model checker CBMC is used as verification backend. The authors assume a scenario where valid software variants are specified by a feature model, and where configuration is realized by preprocessor directives on the level of source code. The configuration process itself is performed by a corresponding tool chain, which the authors call a *Software Variant Generation System (SVGS)*, e.g. realized by makefiles, and a C preprocessor.

Post and Sinz apply the idea of lifting to the verification of a Linux kernel, since the configurable Linux editions constitute a software product line. This is done in several subsequent steps, which essentially correspond to (i) the encoding of the feature model (decoded as *Kbuild*) into C code, (ii) the lifting of the semantic information contained in the makefiles to C code, and (iii) the encoding of arbitrary preprocessor statements into preprocessor-free C code. In summary, the resulting C code represents the so-called meta-program to which verification tools like C software model checkers can be applied. Following this concept the authors address questions such as “May a feature be enabled, although its dependencies are not fulfilled”, or “May a function be used in configurations in which it is not defined”. In summary, this paper reduces to the general idea of checking properties of a software product line on an integrated model of the family rather than on the flat set of individual products, and does not provide any technical or methodological information in more detail. The idea of Sinz et al. is the same as ours, since we also reason and check properties of individual products on basis of the integrated product family model.

In [LPT09], Lauenroth et al. present a model checking approach which allows to verify properties of the domain artifacts of a product line in the presence of variability. The paper refers to one of our papers [GLS08b] from 2008 (in which we define the basic concepts of PF-CCS and the relation to our logic $mv\text{-}\mathfrak{L}_\mu$) and addresses also the challenge of verifying entire product families. In [LPT09], individual products are represented as adjusted I/O-automata, and the properties are specified in CTL. According to Lauenroth et al. model checking of domain artifacts means to verify that every possible product that can be assembled from domain artifacts fulfills the specified properties. In contrast to our logic $mv\text{-}\mathfrak{L}_\mu$ and the approach presented in this chapter, the approach of Lauenroth et al. gives a yes/no-answer whether the property holds, but does not return the set of products in which a property holds, as we do in our approach. In particular, for determining the set of all products for which a particular property holds, the approach of Lauenroth et al. is no improvement compared to the trivial way of checking the property for each product separately.

4. Verifying Properties of PF-CCS Software Product Families

The variability of a product line is specified using the *orthogonal variability modeling language* developed by Pohl et al. . An *orthogonal variability model* provides variation points, variants, variability dependencies and constraint dependencies. Variability dependencies define the allowed selections of a variation point, and constraint dependencies define constraints for the configurations of variations points. Formally, variation points and variables are formalized as Boolean variables. Variability models are interpreted as Boolean expressions over these variables. For a given configuration vector $v (\in \mathbb{Bool}^n)$ Lauenroth et al. introduce the function $OVM(v)$ which returns *true* only if the vector v is a model of the variability model.

I/O-automata are used for the specification of the system behavior. More precisely, the idea of Larsen’s modal I/O-automata is adapted and *may*-transitions are associated with a so-called *variability relation* $VRel_{IO}$ that represents the information of the variability model. This results in so-called *variable I/O-automata* which are used to specify domain artifacts. A transition t in a variable I/O-automaton is variable if t is related to a variant by the variability relation $VRel_{IO}$, otherwise t is said to be common. System properties are specified using an adaption of CTL. A second variability relation $VRel_{CTL}$ relates variants to CTL properties. If a CTL property p is related to a variant v by $VRel_{CTL}$ the property p has to be fulfilled only if the related variant v is selected. As a strong restriction the authors assume that a property cannot be related to more than one variant.

In order to model check variable I/O-automata specifications the I/O-automata that represent the domain artifacts are merged into a single product automaton which provides the notion of variable, common and implicit transitions. Model checking of variable I/O-automata is based on the fundamental model checking approach of Clarke, but adapted to handle the variability information: During the exploration of the state space the algorithm of Lauenroth et al. considers the variability model to ensure that the current path is valid with respect to the variability model. In summary, the approach addresses the challenge that every I/O-automaton that can be derived from a variable I/O-automaton fulfills its CTL properties, i.e. the properties specified for the individual I/O-automaton. However, since the authors do not use a multi-valued logic, they address a different application scenario compared to our multi-valued logic $mv\text{-}\mathcal{L}_\mu$. While in our logic $mv\text{-}\mathcal{L}_\mu$ the result of an evaluation is the set of configurations/products that fulfill the given property, the approach of Lauenroth et al. only allows to check whether a property holds for all products.

In [CHS⁺10], Classen, Heymans, Schobbens, Legay, and Raskin introduce an approach that addresses the model checking of entire product families similarly to ours. After a research visit of one of the authors—Axel Legay—in our group, our work about PF-CCS and $mv\text{-}\mathcal{L}_\mu$ [GLS08b] was well known to him at the time of writing [CHS⁺10]. Consequently, our concepts of checking a product family as a whole, as well as the fundamental concepts of $mv\text{-}\mathcal{L}_\mu$ agree quite well with the approach presented in [CHS⁺10].

Similarly to a PF-LTS (cf. Definition 3.18, Page 140) Classen et al. also define a special kind of labeled transition systems called *Feature Transitions Systems (FTS)*. Like a PF-LTS, an FTS represents an entire product family. An FTS is a transition system in which the transitions are not only labeled with an action, but additionally with the information in which products this transition exists. However, unlike to a PF-LTS where this information is captured by a vector containing configuration decisions of individual variation points, in an FTS this information is modeled by attaching a single feature to each transition. The meaning is that the respective transition is present in all products that contain that feature. In an FTS priorities can be associated to transitions. Priorities allow to model the situation in which one feature overrides the behavior of another. The meaning of a transition priority is that for transitions with a common start state but with different features, only the feature attached to the transition with the higher priority does exist in the resulting product if both features are selected simultaneously. In our approach—where a system is modeled in PF-CCS and not directly as a transition system—transition priorities are not necessary since PF-CCS guarantees that the configuration information attached to the transitions is always constructed such that only consistent products can be derived.

Similarly to a PF-LTS, the individual products can be derived from an FTS by projection to a set of features. Projection is the basis for the semantics of an FTS. The semantics of an FTS is defined as the set of all behaviors of the projections on all valid products. This kind of semantics is conceptually equivalent to our flat semantics (cf. Definition 3.11, Page 124) which is defined as the set of all configurable products. The authors define a reachability relation on the set of products which states in which products (feature combinations) a particular state is reachable. Together with another relation that characterizes the successors of a given state the reachability relation is the basis for the search performed by the model checking algorithm.

The model checking approach of Classen et al. uses an automata-based algorithm which follows the well established ideas of Vardi and Wolper [VW94]. Reachability and the (temporal) properties are expressed by automata, respectively, and model checking reduces to checking whether the product of these automata is empty or not. Conceptually, this kind of automaton-based model checking approach is different to the game-based approach which we introduce in this chapter. The model checking algorithm of Classen et al. is constructed in a way that if the property is satisfied by the FTS then it is also satisfied by every product of the FTS. Otherwise, a counterexample is produced, and the set of all products which violate the property is returned. To this extend, the algorithm of Classen et al. is similar to ours since both allow to determine the set of products that fulfill a given property. The model checking algorithm is implemented in Haskell and comes as a command line tool. According to the benchmarks which Classen et al. performed, the algorithm achieves in average a 20% improvement over the classical algorithm.

4. Verifying Properties of PF-CCS Software Product Families

In general, compared to our approach, by modeling variability in terms of features (and not variation points), Classen et al. focus more on the aspect of structuring the behavior of a software product family and its products into modular entities, in the sense of atomic building entities. While we have discussed the concept of composition and atomic assets in detail in Section 2.2.2.1 (Page 33), for PF-CCS processes are the fundamental structural entities. While in this thesis we have not made a statement on the efficiency gain that is obtained by checking properties on the model of a (PF-CCS) product family directly the approach of Classen et al. comes with a running implementation that allows to gain experimental results regarding the efficiency gain. Although Classen et al. consider “only” LTL properties, these properties represent already a representative subset of the properties that are expressible in our multi-valued variant of the modal μ -calculus, and the observed efficiency gain suggests that a similar efficiency gain can also be expected in our setting based which is also based on a similar representation of the product family as an integrated transition system.

Restructuring PF-CCS Software Product Families

In this chapter we take advantage of the algebraic nature of PF-CCS in order to restructure the term representation of a PF-CCS program while preserving its meaning. This means that we deal with (semantically) one and the same PF-CCS product family, but look at it using different program-representations. Each representation is useful for a different purpose. The formal basis for the restructuring, i.e. the transition from one representation to another, are algebraic laws which we introduce in this chapter. In particular with respect to the commonalities between the products of a PF-CCS product family, the restructuring laws allow to represent the products with a higher or lower degree of commonalities, and represent a formal framework to calculate behavioral commonalities between products.

Contents

5.1. Algebraic Laws	194
5.2. Calculating Commonalities: A Detailed Example	206
5.3. Common Parts	214

5. Restructuring PF-CCS Software Product Families

In order to formally reason about software product families, a formal framework for modeling product families is necessary. With PF-CCS (cf. Chapter 3) we have introduced such a framework. PF-CCS allows the formal specification and modeling of the behavior of a set of similar products in an algebraic manner. Thereby, the entire behavior of all products is represented in an integrated way by the PF-CCS program of the product family.

In this chapter, we consider how to restructure the representation of a PF-CCS product family, resulting in other representations, i.e. other PF-CCS programs, of the same product family. These restructured representations specify still the same product family (with respect to its semantical meaning), but are more suitable to reason about certain aspects of the product family. In particular, the aspect in which we are interested is the common part of a set of products, which for PF-CCS products corresponds to the common operational functionality which is offered by all products.

Thereby, the motivation for restructuring, as well as the technique of restructuring is already familiar to every one of us—even though not from the specific area of software product line engineering, but from a much more common application context: arithmetic. Suppose the task is to multiply the two numbers 49 and 35. There are many possibilities to calculate the result. A very common way is the so-called *long multiplication method*, which is taught already in elementary school as the natural way of multiplication. Another multiplication method, which is in particular useful if a table of pre-calculated square numbers is at hand, bases on the transformation of the product $49 * 35$ into a form in which the relation to square numbers is directly given, according to the following calculation.

$$\begin{aligned} 49 * 35 &= (42 + 7) * (42 - 7) \\ &= 42^2 - 7^2 \\ &= 1764 - 49 \\ &= 1715 \end{aligned}$$

As the square numbers 42^2 and 7^2 can directly be retrieved from the table, the multiplication reduces to performing some other—possibly simpler—operations such as subtraction. Thereby, the original representation ($49 * 35$) is changed into an other, equivalent form ($42^2 - 7^2$), following arithmetic restructuring laws, e.g. the well-known binomial theorem $(x + y) * (x - y) = x^2 - y^2$ (line 1 to 2). Laws like this describe universally valid restructuring principles for the treatment of arithmetic equations, as their correctness can be shown within the laws of arithmetic. While both representations, $49 * 35$ and $42^2 - 7^2$, are equivalent—they denote the same natural number 1715—the last form is much more useful in the presence of a table of pre-calculated square numbers.

Similarly to arithmetic, which specifies a mathematical, formal framework for the calculation with numbers and the restructuring of arithmetic equations, with PF-CCS we have created a formalism for reasoning about and calculating with the operational behavior incorporated in a family of products.

The advantage of using the PF-CCS representation for the restructuring is that compared to the labeled transition system, which defines the semantics of a PF-CCS program, a PF-CCS program itself is always a finite representation, while the corresponding transition system might have infinitely many states. Thus, by restructuring a PF-CCS program directly we can deal with infinite behavior which we could not adequately handle with corresponding restructuring rules that operate on the labeled transition system.

The motivation for restructuring a PF-CCS product family is similar to the one of restructuring an arithmetic equation: while a product family is initially specified in a certain way, this single representation is usually not suitable to equally address all questions in which we are interested during the life of a software product family. Thus—similarly to the laws in arithmetic—a restructuring mechanism based on formally defined laws allows to transform one representation of a product family into another “more useful/interesting” representation, without altering its initial meaning.

In the context of software product families, a very interesting representation form is that one from which we can easily determine the commonalities of products. Undoubtedly, the ability to represent and retrieve the commonalities of a set of products is the main advantage of software product line engineering compared to independent product development, and is a key property of any product family (cf. Chapter 2.2). However, since a software product family evolves over time, i.e. new features and variants are added, the common parts between products change, too, and thus can not be determined statically. In addition, depending on the application scenario, we are interested in a representation of the product family that shows the maximal common part of products, while during the specification of a PF-CCS product family it is methodological very convenient to simply specify the points in which the products differ without being concerned about commonalities.

Thus, in order to work effectively with a software product family, not only is it necessary to be able to *represent* the common parts in a formal way, but even more importantly to be able to work with changing common parts effectively. First and foremost, this requires the ability to *determine* the commonalities of two or more products, similarly to solving an arithmetic equation. In this context—and in analogue to arithmetic—we also speak of the calculation of common parts.

Such a powerful mechanism for the calculation of common parts allows to maximize the reuse of existing software components. For example in a top-down development

5. Restructuring PF-CCS Software Product Families

process, the restructuring mechanism of PF-CCS allows the developer to adjust the behavioral model, i.e. the PF-CCS product family, to match existing (implementation) assets or COTS components in a way that the maximal part of the product family can be “covered” and realized with existing assets. For the opposite situation, where no or very little existing components have to be reused, the software product family can be restructured to a form which shows a maximal degree of common parts. Since common parts have to be implemented only once with a high degree of reuse within the product family, this allows to maximize the efficiency of the development.

In the following we introduce a corresponding restructuring formalism that allows to restructure a PF-CCS program, yielding a syntactically different program that is semantically equivalent, and that represents the commonalities of the derivable products more directly. The restructuring formalism is based on algebraic laws. We introduce these laws in the following and show their correctness, i.e. we show that they do not change the semantics of a PF-CCS program.

5.1. Algebraic Laws

In the following we introduce the relevant algebraic laws of PF-CCS that are the foundation of the restructuring of PF-CCS specifications (PF-CCS programs). In this section we will present the laws in a rather theoretical way without giving a detailed explanation or a methodological embedding. For an illustrative application, which demonstrates how these laws can be used to actually calculate the common parts of products, we refer to the detailed example given in the following Section 5.2.

We speak of “restructuring” since these laws change the term structure, i.e. the *representation*, of a PF-CCS program. Thus—similarly to the laws in arithmetic—a restructuring mechanism based on formally defined laws allows to transform one representation of a software product line into another representation, while preserving its initial meaning. However, in contrast to arithmetic, where the meaning of an arithmetic expression was defined as an integer number, the meaning of a software product line is defined as the set of derivable products, i.e. their number and individual behaviors (cf. Definition 2.8, Page 72). In particular, this means that the application of algebraic restructuring laws does not change the number nor the behavior of the products of a PF-CCS product family. Regarding the correctness of the laws, since the equivalence relation for PF-CCS systems is given by strong bisimulation, we show that the application of any law results in a set of bisimilar products.

The restructuring laws—especially those which are relevant for the purpose of calculating commonalities of products—are mainly distributive laws. In particular, they

implement the essential property of a product family, which is specified by the distributive law given in Axiom 3 of the algebraic specification (Figure 2.11, Page 90). While in the axiomatization we used only one general composition operation and thus required only one distributive law, in PF-CCS we can build up processes using several composition operators, i.e. essentially the operators '.', '+' and '||' inherited from CCS. Thus, we are also interested in the distributive laws of any of these operators over \oplus , as these laws establish the connection between these operators and the variants operator \oplus , respectively. Thereby—following Milner [Mil95]—we differentiate between the PF-CCS operator ||, which describes the static structure of processes, and the operators . and + which describe the more dynamic aspect of how a process evolves by performing actions.

5.1.1. Distributivity of Action Prefixing over \oplus

At first, we consider the distributivity of the variants operator \oplus over the action prefixing operator '.', i.e. we are interested in processes of the kind

$$\alpha.P \oplus_i \alpha.Q$$

where both variants of a variation point \oplus_i can initially perform the same action. For such a situation we observe that the configuration selection of the variation point \oplus_i does not influence the initial action which the entire process $\alpha.P \oplus_i \alpha.Q$ can perform. This means, that for a process $\alpha.P \oplus_i \alpha.Q$, instead of performing the configuration selection between the variants $\alpha.P$ and $\alpha.Q$ right away, we can postpone the configuration decision, perform an action α first, and then select between the processes P or Q , instead. Formally, this is expressed precisely in the following distributive law.

Theorem 5.1 (Distributivity of Action Prefixing). *Action prefixing distributes over the alternative selection of processes, i.e. for any variation point \oplus_i , and any configuration $\theta \in \{L, R, ?\}^n$ where $\theta_i = L$ or $\theta_i = R$, we observe the law*

$$\alpha.P \oplus_i \alpha.Q = \alpha.(P \oplus_i Q)$$

Proof. We show that applying the law does not change the number of derivable products, nor the behavior of any of the products. Obviously, since the law does not alter the number of \oplus_i operators the number of derivable products is not changed. Regarding the behavior, we show that

$$\Pi_\theta([\alpha.P \oplus_i \alpha.Q]_{UF}) \approx \Pi_\theta([\alpha.(P \oplus_i Q)]_{UF})$$

5. Restructuring PF-CCS Software Product Families

for every configuration θ where either $\theta_i = L$ or $\theta_i = R$. For any configuration $\theta \in \{R, L, ?\}^n$ let \mathcal{S}_{left} denote the set of states of the PF-LTS $\Pi_\theta(\llbracket \alpha.P \oplus_i \alpha.Q \rrbracket_{UF})$ (left-hand side of the law), and \mathcal{S}_{right} denote the set of states of the PF-LTS $\Pi_\theta(\llbracket \alpha.(P \oplus_i Q) \rrbracket_{UF})$ (right-hand side). For any projection Π_θ we define a (bisimulation) relation $\mathcal{B}_\theta \subseteq \mathcal{S}_{left} \times \mathcal{S}_{right}$ in the following way:

- $\left((\alpha.P \oplus_i \alpha.Q, \nu) , (\alpha.(P \oplus_i Q), \nu) \right) \in \mathcal{B}_\theta$
- $\left(X , (\alpha.(P \oplus_i Q), \nu) \right) \in \mathcal{B}_\theta$, where $X = \begin{cases} (P, \nu|_{i/L}) & , \text{ if } \theta_i = L \\ (Q, \nu|_{i/R}) & , \text{ if } \theta_i = R \end{cases}$
- $\left((P, \nu) , (P, \nu) \right) \in \mathcal{B}_\theta$ for all remaining states.

This yields two classes of bisimulation relations (cf. Definition 3.17, Page 134) where for one class $\theta_i = L$ and for the other one $\theta_i = R$. Figures 5.1a and 5.1b illustrate the respective PF-LTSs for the unfolded semantics (for the case of an initial ν with $\nu_i = ?$), while the remaining figures show how the relation \mathcal{B}_θ relates the states of the projected systems according to θ for the two cases of $\theta_i = L$ (Figure 5.1c) and $\theta_i = R$ (Figure 5.1d), respectively. Figures 5.1c and Figure 5.1d contain all possible transitions which can be constructed for the two sides $\alpha.P \oplus_i \alpha.Q$ and $\alpha.(P \oplus_i Q)$ according to the SOS rules (cf. Figure 3.4, Page 128). We check easily that for all pairs of states $(s, t) \in \mathcal{B}_\theta$ and every outgoing transition in $\Pi_\theta(\llbracket \alpha.P \oplus_i \alpha.Q \rrbracket_{UF})$, the pair of successor states (s', t') is again in \mathcal{B}_θ for the corresponding transition in $\Pi_\theta(\llbracket \alpha.(P \oplus_i Q) \rrbracket_{UF})$, and vice versa. In all configurations where $\theta_i = L$, the pair $((P, \nu|_{i/L}), (P \oplus_i Q, \nu))$ affords transitions to pairs of identical states $((P', \nu|_{i/L}), (P', \nu|_{i/L})) \in \mathcal{B}_\theta$. The situation for configurations where $\theta_i = R$ is similar. In general, the states in the last rows of Figures 5.1c and 5.1d are to be understood symbolic, since they represent any possible successor of $(P, \nu|_{i/L})$ and $(Q, \nu|_{i/R})$, respectively. However, since each such pair consists of successor states with identical process terms and configuration labels, both states afford the same transitions (according to the SOS rules), yielding again a pair of identical states in \mathcal{B}_θ , and thus are bisimilar, too. Note that for the initial vector ν we assume that $\nu_i = ?$. If this does not hold, then the SOS semantics guarantees that the PF-LTS in Figures 5.1a and 5.1b contain only the transitions for the right or left variant, respectively. \square

Note that since the unfolded semantics and configured-transitions semantics yield bisimilar systems (cf. Theorem 3.3, Page 146) for the same complete configurations, due to the transitivity of the bisimulation relation, the above proof holds likewise for the configured-transitions semantics, too. Consequently, the distributive law holds for all of our semantics. With the same argumentation, the following distributive laws for the other composition operations hold in all of our semantics, too.

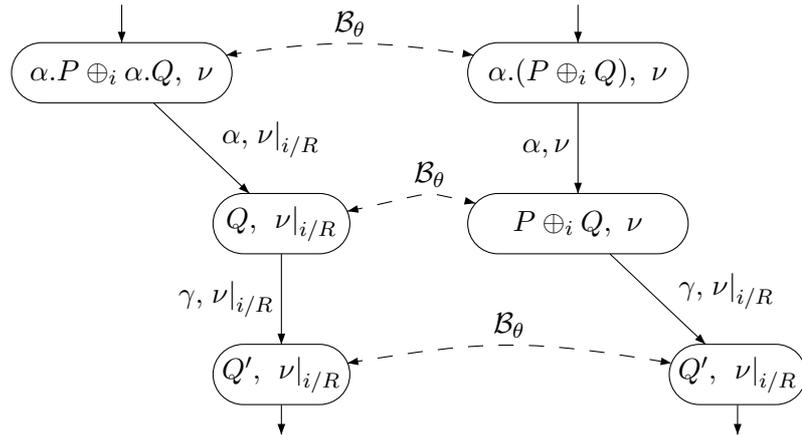
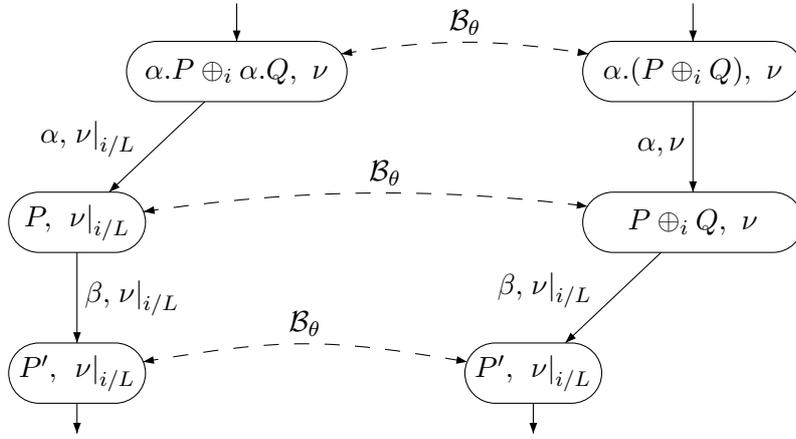
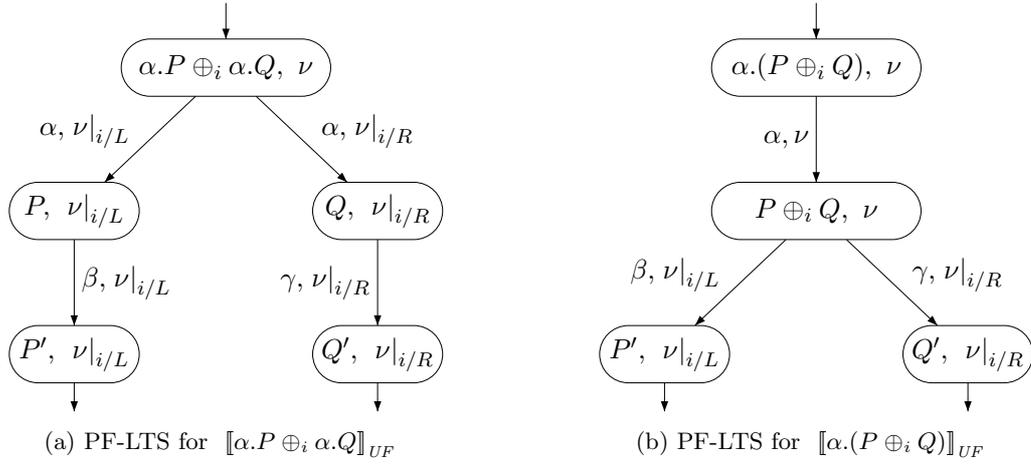


Figure 5.1.: PF-LTS illustrating the distributivity of action prefixing. For all figures we assume that $P \stackrel{def}{=} \beta.P'$ and $Q \stackrel{def}{=} \gamma.Q'$.

5. Restructuring PF-CCS Software Product Families

Theorem 5.1 directly extends to entire sequences of common actions, i.e. it allows us to factor out an entire sequence of identical actions which the variants of the same variation point have initially in common. For example, by repeated application of the Theorem 5.1 we can factor out the initial sequence $\alpha\alpha\alpha$ of the process

$$\begin{aligned}\alpha.\alpha.\alpha.\beta.P \oplus_i \alpha.\alpha.\alpha.\gamma.Q &= \alpha.(\alpha.\alpha.\beta.P \oplus_i \alpha.\alpha.\gamma.Q) \\ &= \alpha.\alpha.(\alpha.\beta.P \oplus_i \alpha.\gamma.Q) \\ &= \alpha.\alpha.\alpha.(\beta.P \oplus_i \gamma.Q)\end{aligned}$$

Such initial sequences are the first example of common parts of variants, and of commonalities of the corresponding products.

However, with common parts we mean common *initial* behavior, not common behavior which is nested in the variants of the same variation point without having an initial common action. Consider for example the process

$$\alpha.P \oplus_i \beta.P$$

Here, the distributive law does not allow to extract P as a common part of both variants, since P is preceded by different actions in the respective variants. Note that this limitation is due to the conceptual idea embodied by the prefixing operator, and not due to the variants operator.

Comparing the variants operator \oplus with the standard non-deterministic choice $+$, we observe that a similar distributive law for the standard non-deterministic choice operator $+$ does *not* hold, i.e. for CCS and PF-CCS we have to reject a law

$$\alpha.P + \alpha.Q = \alpha.(P + Q)$$

since both sides yield non-bisimilar systems, as Figure 5.2 demonstrates. Thus, for the non-deterministic choice operator $+$ we can not factor out an identical initial action of both processes. Against the background of common behavior, this means that although two processes might start with an identical initial sequence of actions, we can not consider this sequence as a common part of both. This shows the conceptual difference between a non-deterministic choice and our configuration-controlled choice operator \oplus , and demonstrates why the non-deterministic choice operator $+$ of CCS can not simulate the variants operator \oplus of PF-CCS.

5.1.2. Distributivity of Non-Deterministic Choice over \oplus

Beside the action prefixing operator, the $+$ operator of PF-CCS is the second *dynamic* composition operation. Like action prefixing, also $+$ has a dynamic character since the $+$ operator is present in the premise of the corresponding SOS rule, but absent in the result (conclusion) of such an SOS rule. This means that the $+$ operation

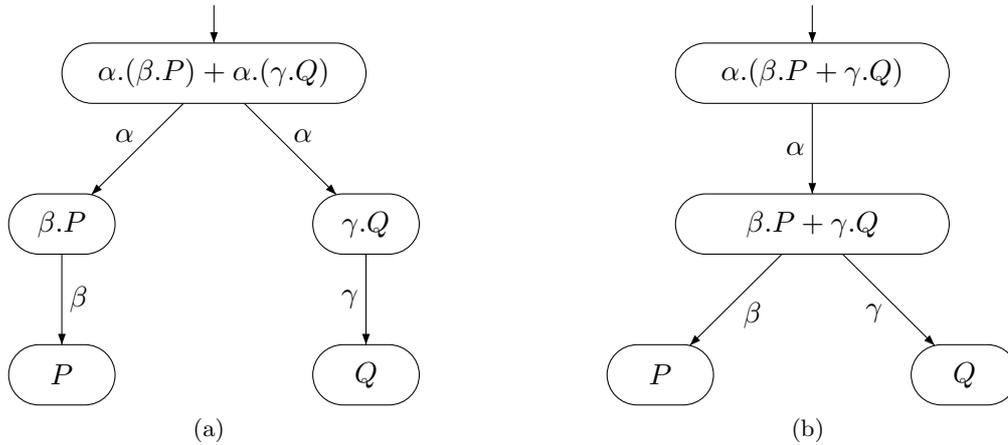


Figure 5.2.: Action prefixing does not distribute over the non-deterministic choice operator of CCS, since the CCS semantics (cf. SOS rules in Figure 3.3, Page 124) for the CCS processes $\alpha.(\beta.P) + \alpha.(\gamma.Q)$ and $\alpha.(\beta.P + \gamma.Q)$ yields transition systems which are obviously not bisimilar.

is “disturbed” by action and thus has no static character. Also for the $+$ operator, we observe a similar distributive law as with the action prefixing operation. For a variation point

$$(P + Q) \oplus_i (P + S)$$

whose two variants offer both the same process P as part of a non-deterministic choice, we can understand the entire variation point as a non-deterministic choice whose two possibilities are the common process P and the alternative choice $Q \oplus_i S$ between the remaining parts of the variants, i.e. between Q and S . The following Theorem 5.2 makes this distributive law more precise.

Theorem 5.2 (Distributivity of Non-Deterministic Choice). *Let \oplus_i be an arbitrary variation point. Then, for any configuration $\theta \in \{L, R, ?\}^n$ where $\theta_i = L$ or $\theta_i = R$, the non-deterministic composition $+$ distributes over the alternative selection \oplus of processes:*

$$P + (Q \oplus_i S) = (P + Q) \oplus_i (P + S)$$

This defines a left-distributivity. Since $+$ is commutative, this also implies right-distributivity and hence full distributivity of $+$ over \oplus .

Proof. We consider the semantics of both sides of the distributive law, where we make a case discrimination according to the initial configuration label ν .

- Assuming $\nu_i = ?$, the application of the corresponding SOS rules (cf. Figure 3.4, Page 128) to the left and right side of the equation yields the two PFLTSs shown in Figure 5.3a and 5.3b. For any configuration $\theta \in \{R, L, ?\}^n$, let

5. Restructuring PF-CCS Software Product Families

\mathcal{S}_{left} denote the set of states of the PF-LTS $\Pi_\theta(\llbracket P + (Q \oplus_i S) \rrbracket_{UF})$, and \mathcal{S}_{right} denote the set of states of the PF-LTS $\Pi_\theta(\llbracket (P + Q) \oplus_i (P + S) \rrbracket_{UF})$. For any projection Π_θ according to a configuration θ we define a (bisimulation) relation $\mathcal{B}_\theta \subseteq \mathcal{S}_{left} \times \mathcal{S}_{right}$ in the following way:

- $\left((P + (Q \oplus_i S), \nu) , ((P + Q) \oplus_i (P + S), \nu) \right) \in \mathcal{B}_\theta$
- $\left((P', \nu) , X \right) \in \mathcal{B}_\theta$, where $X = \begin{cases} (P', \nu|_{i/L}) & , \text{ if } \theta_i = L \\ (P', \nu|_{i/R}) & , \text{ if } \theta_i = R \end{cases}$
- $\left((P, \nu) , (P, \nu) \right) \in \mathcal{B}_\theta$

This yields two classes of bisimulation relations (cf. Definition 3.17, Page 134), where for one class $\theta_i = L$ and for the other one $\theta_i = R$. The relevant extracts of both classes are illustrated in Figures 5.3c and 5.3d. It is easy to see that all states which are related by \mathcal{B}_θ afford the same transitions, and for all such transitions the successor states are again related by \mathcal{B}_θ . Only the pair $((P', \nu), (P', \nu|_{i/L}))$ in configurations θ where $\theta_i = L$ deserves a closer discussion (Similarly the pair $((P', \nu), (P', \nu|_{i/R}))$ in a configuration θ with $\theta_i = R$): Although the SOS rules allow the state (P', ν) to have some outgoing transitions labeled with $\nu|_{i/R}$, the projection according to θ with $\theta_i = L$ discards all such transitions for sure, and only those transitions labeled with vectors $\nu|_{i/L}$ remain. Thus, both states (P', ν) and $(P', \nu|_{i/L})$ actually afford the same transitions to bisimilar states, even though their configuration labels ν and $\nu|_{i/L}$ are not identical.

- For the situation where $\nu_i = L$ (for the states in the topmost row of Figure 5.3c), the SOS semantics only allows to derive outgoing transitions labeled with vectors $\nu|_{i/L}$. Then, any projection according to a configuration θ with $\theta_i = L$ obviously yields bisimilar states, while for the case of $\theta_i = R$ no states exist (not derivable due to the SOS rules), and thus the projections are trivially bisimilar for $\theta_i = R$, too.
- The cases for the situation with an configuration label $\nu_i = R$ are similar.

□

Theorem 5.2 allows to deal with commonalities of processes that are forking non-deterministically. In combination with Theorem 5.1 this distributive law allows us to factor out common action sequences of alternative variants even if each variant itself exhibits a non-deterministic behavior structure.

Note that a similar law for the distributivity of $+$ over $+$, i.e. a law such as

$$P + (Q + S) = (P + Q) + (P + S)$$

does also hold. We can show this directly with the CCS law $P = P + P$ and the associativity of $+$. This, with respect to this law, the variants operator can be “emulated” by the non-deterministic choice operator $+$ of CCS.

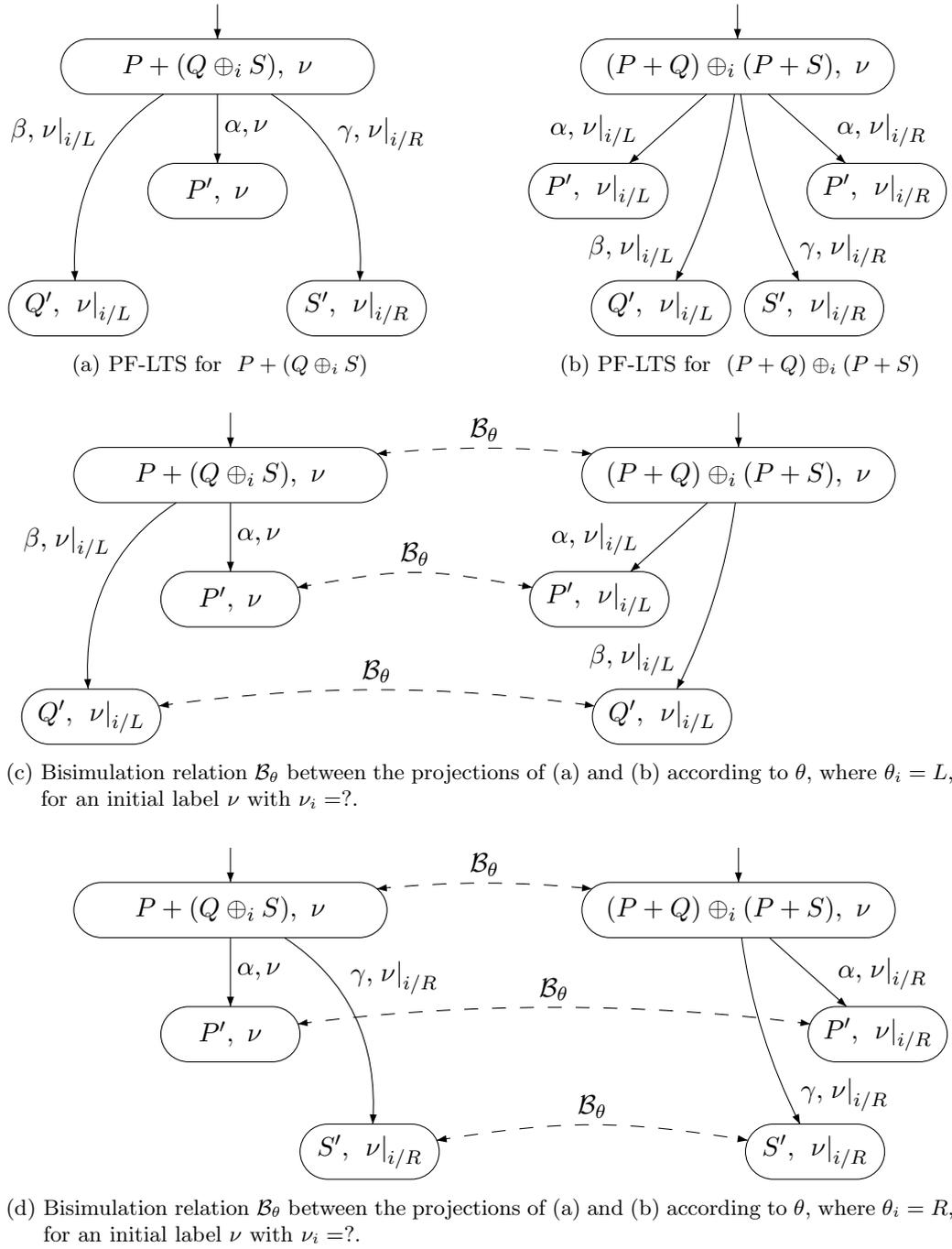


Figure 5.3.: PF-LTS showing the unfolded semantics and some relevant projections for the left-hand and right-hand side of the distributive law for $+$ over \oplus_i (cf. Theorem 5.2). W.l.o.g. we assume that $P \stackrel{\text{def}}{=} \alpha.P'$, $Q \stackrel{\text{def}}{=} \beta.Q'$, and $S \stackrel{\text{def}}{=} \gamma.S'$.

5.1.3. Distributivity of Parallel Composition over \oplus

In contrast to the dynamic operators $+$ and action prefixing $'.'$, the parallel operator \parallel is a so-called *static* operator (cf. Milner [Mil95]), which allows to model the static structure of a process. An inspection of the SOS rules for \parallel (cf. the rules for the parallel composition shown in Figure 3.4, Page 128) shows that the corresponding \parallel operator is present before as well as after the action, while the only things that have changed are the processes which contributed to the action of the compound process. Thus, the process structure which is modeled by the \parallel operators remains and is not “disturbed” by performing actions. Against this background, the \parallel operation allows to model the structure of sub-processes which make up a compound process.

Although the dynamic operators $+$ and $'.'$ are different to the static operator \parallel , for the \parallel operator we observe a distributive law similarly to the ones of the dynamic operators $+$ and $'.'$, too. In the light of the static character of the parallel operator, the distributive law (read from right to left) for the parallel operator formalizes the fact that common parts (in the sense of a contains-relationship of sub-components) can be factored out from the static structure of two alternative processes.

Theorem 5.3 (Distributivity of Parallel Composition). *Let \oplus_i be an arbitrary variation point. Then, for any configuration $\theta \in \{L, R, ?\}^n$ where $\theta_i = L$ or $\theta_i = R$, parallel composition distributes over the alternative selection of processes, i.e.*

$$P \parallel (Q \oplus_i S) = (P \parallel Q) \oplus_i (P \parallel S)$$

This defines a left-distributivity. Since the \parallel operator is commutative, this implies also right-distributivity and hence full distributivity of \parallel over \oplus .

Proof. This proof is analogue to the one of Theorem 5.2. Therefore we omit to visualize the corresponding transition systems. Applying the SOS rules to each side of the distributive law with an initial configuration label ν , where $\nu_i = ?$, yields the two PF-LTSs $\llbracket P \parallel (Q \oplus_i S) \rrbracket_{UF}$ and $\llbracket (P \parallel Q) \oplus_i (P \parallel S) \rrbracket_{UF}$. For any configuration $\theta \in \{R, L, ?\}^n$, let \mathcal{S}_{left} denote the set of states of $\Pi_\theta(\llbracket P \parallel (Q \oplus_i S) \rrbracket_{UF})$, and \mathcal{S}_{right} denote the set of states of $\Pi_\theta(\llbracket (P \parallel Q) \oplus_i (P \parallel S) \rrbracket_{UF})$. For any projection Π_θ according to a configuration θ we define a relation $\mathcal{B}_\theta \subseteq \mathcal{S}_{left} \times \mathcal{S}_{right}$ in the following way:

- $\left((P \parallel (Q \oplus_i S), \nu) , ((P \parallel Q) \oplus_i (P \parallel S), \nu) \right) \in \mathcal{B}_\theta$
- $\left((P' \parallel (Q \oplus_i S), \nu) , X \right) \in \mathcal{B}_\theta$, where $X = \begin{cases} (P' \parallel Q, \nu|_{i/L}) & , \text{ if } \theta_i = L \\ (P' \parallel S, \nu|_{i/R}) & , \text{ if } \theta_i = R \end{cases}$
- $\left((P, \nu) , (P, \nu) \right) \in \mathcal{B}_\theta$

For any projection θ , this relation comprises all states of \mathcal{S}_{left} and \mathcal{S}_{right} , respectively. The inspection of the relation shows, that each pair of states in \mathcal{B}_θ provides the same outgoing transitions resulting in a pair of successor states which are in \mathcal{B}_θ , again. The pair $\left((P' \parallel (Q \oplus_i S), \nu), (P' \parallel Q, \nu|_{i/L}) \right)$ in configurations θ where $\theta_i = L$ deserves a closer discussion (Similarly the pair $\left((P' \parallel (Q \oplus_i S), \nu), (P' \parallel S, \nu|_{i/R}) \right)$ in a configuration θ with $\theta_i = R$): Although the SOS rules allow the state $(P' \parallel (Q \oplus_i S), \nu)$ to have some outgoing transitions labeled with $\nu|_{i/R}$, the projection according to θ with $\theta_i = L$ discards all such transitions, preserving only those transitions labeled with vectors $\nu|_{i/L}$. Thus, both states $(P' \parallel (Q \oplus_i S), \nu)$ and $(P' \parallel Q, \nu|_{i/L})$ actually afford the same transitions to bisimilar states, even though their configuration labels ν and $\nu|_{i/L}$ are not identical.

For the situation where $\nu_i = L$, the SOS semantics only allows to derive outgoing transitions labeled with vectors $\nu|_{i/L}$. Then, any projection according to a configuration θ with $\theta_i = L$ obviously yields bisimilar states, while for the case of $\theta_i = R$ no states exist (not derivable due to the SOS rules), and thus the projections are trivially bisimilar for $\theta_i = R$, too. The cases for the situation with an configuration label $\nu_i = R$ are similar. \square

Comparing the variants operator \oplus with the non-deterministic choice operator $+$, a similar law for the distributivity of \parallel over the non-deterministic choice $+$ does *not* hold. To see this, consider for example the processes

$$\alpha.P \parallel (\beta.Q + \gamma.S) \quad \text{and} \quad (\alpha.P + \beta.Q) \parallel (\alpha.P + \gamma.S)$$

While the left process can always perform a β and a γ action after an initial α action, the right process can perform either a β or a γ action, once an initial α action has been performed. This example shows once more, that the configuration selection is a conceptually different operation than the non-deterministic choice, and that the non-deterministic choice can not simulate the configuration selection \oplus .

Remarks on the Distributive Laws

From an algebraic perspective, the introduced distributive laws make a rather unspectacular statement about the connection of different operators. However, from a product family perspective, the distributive laws describe an essential concept how to restructure the assets of a product family. In particular, they represent the connection between common (mandatory) and variable parts. More precisely, if we read the Theorems 5.2 and 5.3 from left to right (Theorem 5.1 from right to left), they express the formal relationship how common parts can be distributed over variation points. Applied in the other direction from right to left (Theorem 5.1 from left to right), the theorems describe how *identical* (in terms of bisimulation) parts P can

5. Restructuring PF-CCS Software Product Families

be extracted from two alternative variants of the same variation point. This equals the extraction of the common part of two variants. When applying the distributive laws in this direction we speak of *factoring out* a common part. Thinking of the representation of a product family as a term hierarchy containing several variation points at various hierarchical levels, “restructuring” by applying the distributive laws corresponds to moving the variation points throughout the term hierarchy towards the leaves or towards the root. In this context, factoring out a common part equals to push the corresponding variants operator deeper into the term hierarchy towards the leaves.

Note that the application of the distributive laws does not change the number of variation points, nor does it change the assignment between a variant and its representation label R or L in the configuration. Recall for example Theorem 5.2.

$$P + (Q \oplus_i S) = (P + Q) \oplus_i (P + S)$$

If—without loss of generality—the right variant S is selected, i.e. $\theta_i = R$, the resulting system will always be $P + S$, no matter which side of the distributive law we use to perform the configuration (derive the system). In particular, choosing the right variant will always result in choosing process S , independent of the application of the distributive law.

However, applying the distributive law changes the variants which a variation point \oplus_i offers. Consider again the example of the distributive law for $+$ over \oplus from above. While for the left side of the law the variation point \oplus_i offers the variants Q and S , the same variation point \oplus_i offers different variants, the variants $(P + Q)$ and $(P + S)$, when we consider the right side of the law. However, this has no effect on the semantics of a PF-CCS program, as the proof for the theorems has shown that the same configuration yields the same final product, independently of the representation (before or after applying the law) that is used for performing the configuration.

5.1.4. Miscellaneous Laws

While the distributive laws introduced above are very important for any PF-CCS product family, they are not the only laws which we use to restructure a PF-CCS program. In fact, many laws which we use are not even specific for PF-CCS, but are inherited from CCS.

In Chapter 3 we have seen how PF-CCS extends CCS with the concept of alternative variants (implemented by the variants operator) and adjusts the original CCS semantics to be able to deal with such variants. Regarding the properties and laws that hold in CCS, PF-CCS can be seen as a *conservative* extension of CCS in the

following sense: PF-CCS inherits all algebraic laws which already hold in CCS. In particular, there is no law which holds in CCS, but which does not hold anymore in PF-CCS. For example, the simple law concerning the restriction operation

$$P \setminus L_1 \setminus L_2 = P \setminus \{L_1 \cup L_2\}$$

holds likewise in CCS and PF-CCS. Certainly, in this context the term *conservative* is not to be understood in its strict (mathematical) logical meaning, nevertheless, the idea is the same: With PF-CCS we only add new concepts while preserving the validity of the original ones. In particular, a PF-CCS program containing no variation points corresponds directly to the equivalent CCS program, i.e. it fulfills the same properties and the same laws are applicable.

Against this background, for restructuring a PF-CCS program we can use the existing algebraic laws and theorems that already hold in CCS. All important algebraic laws of CCS—and thus also all inherited laws which we take over to PF-CCS—are introduced and proven in one of the sources [Mil95, Fok00, BK84]. Since these laws (i) belong to the common knowledge of anyone familiar with process algebras, and (ii) are not a contribution of this thesis we do not list them here explicitly again, and simply use them in the following calculations, referring to them as *standard laws* or *CCS laws*. Beside these standard laws, there are some more PF-CCS specific algebraic laws and restructuring principles which are necessary to effectively perform a restructuring of a PF-CCS program. We will introduce them in the following.

Substitution of Variants Identifiers

The following Theorem 5.4 deals with the role of a variant in the scope of its variation point. More precisely, it defines a substitution rule for the variants P and Q , if they are used as direct variants of a variation point $P \oplus_i Q$.

Theorem 5.4 (Substitution of Variant Identifiers). *Let (\mathcal{E}_P, P) and (\mathcal{E}_Q, Q) be two PF-CCS programs where the sets of process identifiers appearing in the equations \mathcal{E}_P and \mathcal{E}_Q are disjoint. Further, let the alternative combination of two systems (\mathcal{E}_P, P) and (\mathcal{E}_Q, Q) be defined as (\mathcal{E}_S, S) , where*

- $\mathcal{E}_S = \mathcal{E}_P \uplus \mathcal{E}_Q$,
- $S \stackrel{\text{def}}{=} P \oplus_i Q$, and
- \oplus_i is a fresh variants operator.

Then, we can substitute every occurrence of P and Q in the equations of \mathcal{E}_S by the term $P \oplus_i Q$.

5. Restructuring PF-CCS Software Product Families

Proof. Both systems P and Q appear as variants in a variation point $P \oplus_i Q$. Since P and Q might be recursive process specifications, both might contain their main process identifiers again within their equations \mathcal{E}_P and \mathcal{E}_Q , respectively. The process identifier P is only substituted on the left-hand side of the variation point, while Q is only substituted on the right-hand side (due to the disjointness of the process identifiers). Additionally, every substitution $P \oplus_i Q$, which is made for P or Q , refers to the same variation point \oplus_i . Consequently, depending on the configuration for \oplus_i , either (i) for a configuration $\theta_i = L$, all substitution terms $P \oplus_i Q$ are set to P and the left-hand side of the equation for the variation point is selected, or (ii) for $\theta_i = R$, all substitution terms $P \oplus_i Q$ are set to Q , while now the right-hand side of the variation point is selected. In both cases, the remaining terms contain the correct process identifiers P or Q at the appropriate sides of the variation point. Note that the unselected side of the equations always contains the wrong process identifiers. But since the terms of this side are not selected anyway, the remaining system is still correct. \square

The correctness of the previous theorem depends crucially on the disjointness of the process identifiers in both programs. This ensures that when the programs are combined as variants of the variation point $P \oplus_i Q$, the process identifier P only appears on the left-hand side, and Q only on the right-hand side of \oplus_i . Thus, the identifiers P and Q can everywhere be substituted by $P \oplus_i Q$, and any configuration the entire program will always result in the desired program. The example in the next section will motivate why and in which situations this theorem is beneficial for the computation of common parts.

5.2. Calculating Commonalities: A Detailed Example

In this section we give a detailed, guided example of how the algebraic laws of the previous section can be applied in order to compute common parts of similar PF-CCS systems. Thereby—anticipating the definition of a common part as given in Section 5.3—we use the concept of *common parts* already in this example, without having defined it precisely beforehand. However, in this case we can accept this style since the example is understandable in its own rights and in particular greatly eases the understanding of the upcoming definition of common parts.

We use again the example of a family of windscreen wipers—yet in an adjusted version compared to the example introduced in Section 4.3. More precisely, in our example we consider two version of similar windscreen wiper systems and compute their “maximal” common part. As a prerequisite for the calculation both wiper systems are combined as variants of the same variation point. Then the distributive laws are applied in order to factor out common actions and (bisimilar) action

5.2. Calculating Commonalities: A Detailed Example

structures. Regarding the distributive laws, the goal is to restructure the product family into a kind of normal form, in the sense of the normal form which we have introduced in Chapter 2.2.3.2 for software product families in general. Recall, that for constructing the normal form in Chapter 2.2.3.2 common parts which exist in the variants of a variation point are factored out by applying the distributive law in the appropriate direction, until all variants have no more common parts, and the common parts form compound elements with the variation points from which they were factored out. Thus, with respect to common parts, the goal of the calculation that we perform in the following example is to bring the respective windscreen wiper family in its normal form (regarding the application of the distributive laws).

We consider a simple version of a windscreen wiper *FWS* (Front Wiper Standard) for the front window of a car. It offers three different operation modes, where the behavior in each mode is represented by the three processes *FWS*, *Intv*, and *Perm*. They describe the situations where the windscreen wiper

1. is not operating but waiting in its initial mode (*FWS*),
2. operates with intermittent periodic stops (*Intv*),
3. operates continuously without intermittent breaks (*Perm*).

The corresponding PF-CCS specification of *FWS* is shown in Figure 5.4. Initially, the operation modes are triggered by the actions *off*, *intv*, and *permOn*, respectively. In *Perm*, the system executes constantly single wiper arm movements (action *wipe*) unless ceased by an *off* action. The interval mode (*Intv*) behaves similarly but adds a rest period between successive wiper arm movements, which is realized by shared communication of the action *wait* between the restricted, parallel processes *Intv* and *Nil*. In addition, the wiper offers a second interval mode *Intv2* with a shorter interval period, which can be activated only consecutively from the basic interval mode *Intv*. This kind of interval control corresponds to the one typically found in cars with a turn-switch for the wiper functionality. In all three modes, an *off* action sets the wiper back to its starting behavior specified by the initial process *FWS*.

Another version of a windscreen wiper for the front window offers similar functionality as *FWS*, but includes some more comfort features, such as automatically

$$\begin{aligned}
 FWS &\stackrel{def}{=} off.FWS + intv.Intv + permOn.Perm \\
 Intv &\stackrel{def}{=} \overline{wipe}.WaitL + intv2.Intv2 + off.FWS \\
 WaitL &\stackrel{def}{=} (\overline{wait}.wait.Intv \parallel wait.wait.Nil) \setminus \{wait\} \\
 Intv2 &\stackrel{def}{=} \overline{wipe}.Wait + intv.Intv \\
 Wait &\stackrel{def}{=} (\overline{wait}.Intv2 \parallel wait.Nil) \setminus \{wait\} \\
 Perm &\stackrel{def}{=} \overline{wipe}.Perm + off.FWS
 \end{aligned}$$

Figure 5.4.: Specification of a front screen wiper *FWS*, standard version.

5. Restructuring PF-CCS Software Product Families

$$\begin{aligned}
FWC &\stackrel{\text{def}}{=} \text{off}.FWC + \text{intv}.Intv + \text{permOn}.Perm \\
Intv &\stackrel{\text{def}}{=} \overline{\text{wipe}}.WaitL + \text{hvyRn}.Fast + \text{off}.FWC \\
WaitL &\stackrel{\text{def}}{=} (\overline{\text{wait}}.\overline{\text{wait}}.Intv \parallel \text{wait}.\text{wait}.Nil) \setminus \{ \text{wait} \} \\
Fast &\stackrel{\text{def}}{=} \overline{\text{wipe}}.Wait + \text{off}.FWC + \text{ttlRn}.Intv + \text{permOn}.Perm \\
Wait &\stackrel{\text{def}}{=} (\overline{\text{wait}}.Fast \parallel \text{wait}.Nil) \setminus \{ \text{wait} \} \\
Perm &\stackrel{\text{def}}{=} \overline{\text{wipe}}.Perm + \text{off}.FWC
\end{aligned}$$

Figure 5.5.: Specification of a front screen wiper FWC with comfort features.

adjusting wiper arm speed for the interval mode, matching the current rain intensity. Whenever the wiper is in interval mode and heavy rain is detected—represented by the input action $hvyRn$ —the wiper arm adjusts its wiper arm speed automatically and starts with fast interval wiper arm movements, as specified by the process $Fast$. As soon as the rain intensity decreases—represented by the input action $ttlRn$ —the wiper switches back to regular interval mode. This more comfortable version of the windscreen wiper is realized by the process FWC (Front Wiper Comfort), whose entire PF-CCS specification is given in Figure 5.5.

Now, the question is what are the commonalities of both wiper variants? What are their common parts and how can we represent or even calculate them? Intuitively—simply by inspecting the corresponding PF-CCS program for FWS and FWC —we guess that both versions have at least the behavior of the permanent wiping mode in common. In addition, their interval modes seem to be similar, however, already here we probably can not specify the commonalities exactly anymore. We will address this challenge in the following and calculate the common parts of both versions in a formal way by applying the introduced algebraic laws.

In order to determine the common parts we join both wiper versions to form a family of front wipers. Therefore, we combine the corresponding start processes FWS and FWC as alternative variants under the same variation point, which we label with a fresh number, i.e. a number which is not already taken for a variation point in the programs of FWS or FWC . Since both systems FWS and FWC do not contain any variation points, yet, the new variation point is labeled with number 1.

$$FWFam \stackrel{\text{def}}{=} FWS \oplus_1 FWC \tag{5.1}$$

The Equation 5.1 together with the PF-CCS specifications of the two processes FWS and FWC constitutes the program $(\mathcal{E}, FWFam)$ which is the basis for our further calculation. Thereby, the set \mathcal{E} of equations is given as the union of the equations of FWS and FWC . However, before combining both sets of equations we have to ensure that the process identifiers of both programs do not (accidentally) overlap. In this example, we therefore index every clashing identifier of the specification of FWC with the letter 'c'. Note that overlapping is only problematic for process identifiers,

5.2. Calculating Commonalities: A Detailed Example

not for actions, since we assume that actions of the same name in different PF-CCS programs actually represent the same actions. In particular, this means that actions are not subject to renaming due to name clashes. The entire program $(\mathcal{E}, FWFam)$, where all name clashes are resolved, is shown in Figure 5.6a.

The calculation for determining the common parts is given in detail in Figure 5.7. The calculation restructures Equation 5.1 by applying the distributive laws (Theorems 5.2 and 5.3) in the direction from right to left, i.e. “pushing” the variation points deeper into the term hierarchy and thereby factoring out common parts. We consider it step by step. Line (1) to (2) is a simple expansion of the original process constants according to the defining equations of FWS and FWC given in Figure 5.6a. Line (2) to (3) is the substitution of the processes FWS and FWC by the term $FWS \oplus_1 FWC$ according to Theorem 5.4. Here, Theorem 5.4 is applicable since both wiper versions are combined as variants of the same variation point \oplus_i . As we will perform the same substitution again going from line (5) to (6), from now on we directly expand the process definitions using the equations of Figure 5.6b, as in these equations the substitution of FWS and FWC by $FWS \oplus_1 FWC$ is already performed. Line (3) to (4) shows the extraction of the common process $off.(FWS \oplus_1 FWC)$ out of a nondeterministic sum according to Theorem 5.2. Regarding Line (4) we can not straight away extract any further processes by means of the distributive laws, as the process identifiers are formally (syntactically) different. However, semantically the processes $Perm$ and $Perm_c$ are bisimilar and thus equivalent. We see the bisimulation easily when considering the corresponding process definitions for $Perm$ and $Perm_c$ shown in Figure 5.6b. Since (strong) bisimulation is a (strong) congruence relation for all operators of CCS [Mil95] and PF-CCS, we can substitute both processes for each other. W.l.o.g. we substitute $Perm_c$ by $Perm$, and can now directly apply the distributive law for $+$ (Theorem 5.2) resulting in the term shown in Line (5). For the remaining term $intv.(Intv \oplus_1 Intv_c)$ we can extract the common initial action $intv$ according to Theorem 5.1, resulting in Line (6). Since the processes $Intv$ and $Intv_c$ are not bisimilar, we expand them according to their process definitions shown in Figure 5.6b. Another application of the distributive law of $+$ in Line (7) allows us to extract the common process $off.(FWS \oplus_1 FWC)$ from both interval processes. Since the processes $WaitL$ and $WaitL_c$ are not bisimilar we can not factor out any further common parts using the distributive laws.

In summary, the calculation yields the following result.

$$\begin{aligned}
 FWS \oplus_1 FWC &= off.(FWS \oplus_1 FWC) + permOn.Perm + & (5.2) \\
 &intv. \left(off.(FWS \oplus_1 FWC) + \right. \\
 &\quad \left((\overline{wiper}.WaitL + intv2.Intv2) \right. \\
 &\quad \left. \left. \oplus_1 (\overline{wiper}.WaitL + hvyRn.Fast) \right) \right)
 \end{aligned}$$

5. Restructuring PF-CCS Software Product Families

$$\begin{aligned}
FWFam &\stackrel{def}{=} FWS \oplus_1 FWC \\
FWS &\stackrel{def}{=} \overline{off}.FWS + \text{intv}.Intv + \text{permOn}.Perm \\
Intv &\stackrel{def}{=} \overline{wipe}.WaitL + \text{intv2}.Intv2 + \overline{off}.FWS \\
WaitL &\stackrel{def}{=} (\overline{wait}.\overline{wait}.Intv \parallel \text{wait}.\text{wait}.Nil) \setminus \{wait\} \\
Intv2 &\stackrel{def}{=} \overline{wipe}.Wait + \text{intv}.Intv \\
Wait &\stackrel{def}{=} (\overline{wait}.Intv2 \parallel \text{wait}.Nil) \setminus \{wait\} \\
Perm &\stackrel{def}{=} \overline{wipe}.Perm + \overline{off}.FWS \\
FWC &\stackrel{def}{=} \overline{off}.FWC + \text{intv}.Intv_c + \text{permOn}.Perm_c \\
Intv_c &\stackrel{def}{=} \overline{wipe}.WaitL_c + \text{hvyRn}.Fast + \overline{off}.FWC \\
WaitL_c &\stackrel{def}{=} (\overline{wait}.\overline{wait}.Intv_c \parallel \text{wait}.\text{wait}.Nil) \setminus \{wait\} \\
Fast &\stackrel{def}{=} \overline{wipe}.Wait_c + \overline{off}.FWC + \text{ttlRn}.Intv_c \\
&\quad + \text{permOn}.Perm_c \\
Wait_c &\stackrel{def}{=} (\overline{wait}.Fast \parallel \text{wait}.Nil) \setminus \{wait\} \\
Perm_c &\stackrel{def}{=} \overline{wipe}.Perm_c + \overline{off}.FWC
\end{aligned}$$

(a) Program without substitution.

$$\begin{aligned}
FWFam &\stackrel{def}{=} FWS \oplus_1 FWC \\
FWS &\stackrel{def}{=} \overline{off}.(FWS \oplus_1 FWC) + \text{intv}.Intv + \text{permOn}.Perm \\
Intv &\stackrel{def}{=} \overline{wipe}.WaitL + \text{intv2}.Intv2 + \overline{off}.(FWS \oplus_1 FWC) \\
WaitL &\stackrel{def}{=} (\overline{wait}.\overline{wait}.Intv \parallel \text{wait}.\text{wait}.Nil) \setminus \{wait\} \\
Intv2 &\stackrel{def}{=} \overline{wipe}.Wait + \text{intv}.Intv \\
Wait &\stackrel{def}{=} (\overline{wait}.Intv2 \parallel \text{wait}.Nil) \setminus \{wait\} \\
Perm &\stackrel{def}{=} \overline{wipe}.Perm + \overline{off}.(FWS \oplus_1 FWC) \\
FWC &\stackrel{def}{=} \overline{off}.(FWS \oplus_1 FWC) + \text{intv}.Intv_c + \text{permOn}.Perm_c \\
Intv_c &\stackrel{def}{=} \overline{wipe}.WaitL_c + \text{hvyRn}.Fast + \overline{off}.(FWS \oplus_1 FWC) \\
WaitL_c &\stackrel{def}{=} (\overline{wait}.\overline{wait}.Intv_c \parallel \text{wait}.\text{wait}.Nil) \setminus \{wait\} \\
Fast &\stackrel{def}{=} \overline{wipe}.Wait_c + \overline{off}.(FWS \oplus_1 FWC) + \text{ttlRn}.Intv_c \\
&\quad + \text{permOn}.Perm_c \\
Wait_c &\stackrel{def}{=} (\overline{wait}.Fast \parallel \text{wait}.Nil) \setminus \{wait\} \\
Perm_c &\stackrel{def}{=} \overline{wipe}.Perm_c + \overline{off}.(FWS \oplus_1 FWC)
\end{aligned}$$

(b) Substitution of FWS and FWC by $FWS \oplus_1 FWC$ according to Theorem 5.4.

Figure 5.6.: The PF-CCS program $(\mathcal{E}, FWFam)$ specifying a family of front screen wipers. The program is the starting point for the calculation of the common parts of the two front wiper variants. Figure (b) shows the version of (a), where the process identifiers FWS and FWC are already substituted with the term $FWS \oplus_1 FWC$ according to Theorem 5.4.

5.2. Calculating Commonalities: A Detailed Example

$$\begin{aligned}
(1) \quad & FWFam \stackrel{def}{=} FWS \oplus_1 FWC \\
(2) \quad & = \left(\begin{array}{l} \text{off}.FWS + \text{intv}.Intv + \text{permOn}.Perm \\ \text{off}.FWC + \text{intv}.Intv_c + \text{permOn}.Perm_c \end{array} \right) \oplus_1 \\
(3) \quad & \stackrel{Th5.4}{=} \left(\begin{array}{l} \text{off}.(FWS \oplus_1 FWC) + \text{intv}.Intv + \text{permOn}.Perm \\ \text{off}.(FWS \oplus_1 FWC) + \text{intv}.Intv_c + \text{permOn}.Perm_c \end{array} \right) \oplus_1 \\
(4) \quad & \stackrel{Th5.2}{=} \text{off}.(FWS \oplus_1 FWC) + \\
& \left(\begin{array}{l} (\text{intv}.Intv + \text{permOn}.Perm) \oplus_1 \\ (\text{intv}.Intv_c + \text{permOn}.Perm_c) \end{array} \right) \\
(5) \quad & \stackrel{Perm \approx Perm_c; Th5.2}{=} \text{off}.(FWS \oplus_1 FWC) + \\
& \text{permOn}.Perm + \\
& \left(\text{intv}.Intv \oplus_1 \text{intv}.Intv_c \right) \\
(6) \quad & \stackrel{Th5.1}{=} \text{off}.(FWS \oplus_1 FWC) + \\
& \text{permOn}.Perm + \\
& \text{intv}.(Intv \oplus_1 Intv_c) \\
(7) \quad & = \text{off}.(FWS \oplus_1 FWC) + \\
& \text{permOn}.Perm + \\
& \text{intv}.\left(\begin{array}{l} \overline{\text{wipe}}.WaitL + \text{intv2}.Intv2 + \text{off}.(FWS \oplus_1 FWC) \\ \oplus_1 (\overline{\text{wipe}}.WaitL_c + \text{hvyRn}.Fast + \text{off}.(FWS \oplus_1 FWC)) \end{array} \right) \\
(8) \quad & \stackrel{Th5.2}{=} \text{off}.(FWS \oplus_1 FWC) + \\
& \text{permOn}.Perm + \\
& \text{intv}.\left(\begin{array}{l} \text{off}.(FWS \oplus_1 FWC) + \\ ((\overline{\text{wipe}}.WaitL + \text{intv2}.Intv2) \oplus_1 \\ (\overline{\text{wipe}}.WaitL_c + \text{hvyRn}.Fast)) \end{array} \right)
\end{aligned}$$

Figure 5.7.: Calculation of the commonalities of FWS and FWC .

5. Restructuring PF-CCS Software Product Families

This result provides a new representation of the product family. It means that in order to obtain one of the products FWS or FWC , instead of performing the configuration selection directly between these two processes using the variation point $FWS \oplus_1 FWC$, we can likewise select between the variants $(\overline{wiper}.WaitL + intv2.Intv2)$ and $(\overline{wiper}.WaitL + hvyRn.Fast)$ of the variation point \oplus_1 embedded in the process structure shown on the right side of Equation 5.2. Note that the (same) variation point \oplus_1 offers different pairs of variants, depending on the context/position in which it appears in the term structure. In the representation on the left side of Equation 5.2, the variation point is the uppermost token in the term structure, while in the representation on the right side of Equation 5.2, the variation point is embedded into a greater process term structure. It is the combination of the different pairs of variants together with the position of the variation point \oplus_1 itself that guarantees that the overall result of configuring the right side of Equation 5.2 will be the same as when configuring the left side for the same configuration. In particular, a configuration θ with $\theta_1 = L$ (respectively $\theta_1 = R$) will yield a product equivalent (bisimilar) to FWS (respectively FWC), independently from the representation (left or right side of Equation 5.2) which we use to perform the configuration.

In the representation shown on the right side of Equation 5.2, the variation point \oplus_1 is pushed as far into the term hierarchy as possible, thereby extracting the common part of both wiper variants. In this position, the variation point shows the difference between both products in its most “exact” form. In particular, the difference between both products can not be any “smaller” since both variants have no common part anymore. We see this easily since the variation point can not be pushed further into the term hierarchy by means of the distributive laws. The actual differences themselves, i.e. the way *how* both products differ, are precisely characterized by the two variants $(\overline{wiper}.WaitL + intv2.Intv2)$ and $(\overline{wiper}.WaitL + hvyRn.Fast)$. In order to visualize the differences better, we define a new process $Diff$ as an abbreviation of this variation point.

$$Diff \stackrel{def}{=} (\overline{wiper}.WaitL + intv2.Intv2) \oplus_1 (\overline{wiper}.WaitL + hvyRn.Fast) \quad (5.3)$$

In contrast to their difference, the common behavior of both wiper processes is represented by the process term which is pushed out during the calculation by the application of the distributive laws. In Equation 5.2 this is that part of the defining equation which is parsed until the variation point $(\overline{wiper}.WaitL + intv2.Intv2) \oplus_1 (\overline{wiper}.WaitL + hvyRn.Fast)$ is reached. A substitution of the process $FWS \oplus_1 FWC$ by $FWFam$ according to our initial constant definition (Equation 5.1), and the usage of the previously defined process $Diff$ yields a representation which shows the common part more clearly:

$$FWFam = off.FWFam + permOn.Perm + \quad (5.4) \\ intv. \left(off.FWFam + Diff \right)$$

5.2. Calculating Commonalities: A Detailed Example

$$\begin{aligned}
FWFam &\stackrel{\text{def}}{=} \text{off}.FWFam + \text{permOn}.Perm + \text{intv}.\left(\text{off}.FWFam + \text{Diff}\right) \\
Diff &\stackrel{\text{def}}{=} \left(\overline{\text{wipe}}.\text{WaitL} + \text{intv2}.\text{Intv2}\right) \oplus_1 \left(\overline{\text{wipe}}.\text{WaitL}_c + \text{hvyRn}.\text{Fast}\right) \\
Perm &\stackrel{\text{def}}{=} \overline{\text{wipe}}.Perm + \text{off}.FWFam \\
Intv &\stackrel{\text{def}}{=} \overline{\text{wipe}}.\text{WaitL} + \text{intv2}.\text{Intv2} + \text{off}.FWFam \\
WaitL &\stackrel{\text{def}}{=} \left(\overline{\text{wait}}.\overline{\text{wait}}.\text{Intv} \parallel \text{wait}.\text{wait}.\text{Nil}\right) \setminus \{\text{wait}\} \\
Intv2 &\stackrel{\text{def}}{=} \overline{\text{wipe}}.\text{Wait} + \text{intv}.\text{Intv} \\
Wait &\stackrel{\text{def}}{=} \left(\overline{\text{wait}}.\text{Intv2} \parallel \text{wait}.\text{Nil}\right) \setminus \{\text{wait}\} \\
Intv_c &\stackrel{\text{def}}{=} \overline{\text{wipe}}.\text{WaitL}_c + \text{hvyRn}.\text{Fast} + \text{off}.FWFam \\
WaitL_c &\stackrel{\text{def}}{=} \left(\overline{\text{wait}}.\overline{\text{wait}}.\text{Intv}_c \parallel \text{wait}.\text{wait}.\text{Nil}\right) \setminus \{\text{wait}\} \\
Fast &\stackrel{\text{def}}{=} \overline{\text{wipe}}.\text{Wait}_c + \text{off}.FWFam + \text{ttlRn}.\text{Intv}_c + \text{permOn}.Perm_c \\
Wait_c &\stackrel{\text{def}}{=} \left(\overline{\text{wait}}.\text{Fast} \parallel \text{wait}.\text{Nil}\right) \setminus \{\text{wait}\} \\
Perm_c &\stackrel{\text{def}}{=} \overline{\text{wipe}}.Perm_c + \text{off}.FWFam
\end{aligned}$$

Figure 5.8.: Restructured version of the product family of front screen wipers with a maximal degree of common parts.

The common part of two PF-CCS processes is the initial common behavior of both processes, until the first behavioral differences are encountered. The common part of the two front wipers comprises the entire process $FWFam$ as defined in Equation 5.4 until the process $Diff$ is executed. Thus, both derivable products—the system FWS corresponding to the projection of $FWFam$ according to a configuration θ with $\theta_i = L$, and system FWC corresponding to $\theta_i = R$ —share the common behavior of

- remaining in an initial mode ($FWFam$) until another action than the action off is performed,
- being capable of performing permanent wiper arm movements as described by the process $Perm$, initially triggered by the action $permOn$, and
- offering interval modes, which are for both products initiated by the action $intv$ and suspended by the action off , leading again to the initial mode $FWFam$.

The entire restructured representation of the product family can be derived from the result of the calculation. Instead of the original result (Equation 5.2) we use the equivalent version shown in Equation 5.4 which (i) uses the substitution of $FWS \oplus_1 FWC$ by the constant $FWFam$, and which (ii) shows the differences more illustratively using the process $Diff$ as defined in Equation 5.3. This equation together with the required original process definitions of Figure 5.6a constitute the entire restructured version of our product family. It is shown in Figure 5.8. Note that it is not necessary to include the process $Perm_c$, as we have seen that $Perm_c$ is bisimilar (and congruent) to the process $Perm$ and thus can be replaced by $Perm$ throughout the entire program. However, in order to improve the comprehensibility of Figure 5.8 we use both processes in the program.

5. Restructuring PF-CCS Software Product Families

Let us briefly recapitulate the calculation. It confirms what we initially have already expected, that (i) both wipers can perform the same functionality of wiping permanently, and that (ii) their behaviors in the interval modes are similar, but not identical. However, while outside the PF-CCS framework we are not able to specify this commonality formally, in PF-CCS we can not only specify the commonalities, but also calculate them in a schematic way based on well-defined laws. Certainly, the calculation of commonalities cannot be performed in a completely automated fashion. Similarly to a calculation in arithmetic, the calculation has to be performed in a guided way. However, compared to the situation without PF-CCS, we now can perform the restructuring according to formal rules, which allow to proof the correctness of restructuring operations. Thus, PF-CCS gives the basis to reason about a product family, and especially the commonalities and differences of its products in a formal way using formal methods.

5.3. Common Parts

The example in the preceding section has given us the right intuition for commonalities between products, and has also sketched a rudimentary procedure of how a calculation can take place. However, we have not yet precisely defined what a common part actually is. We will do so in the following.

Although we use the term representation of a PF-CCS program in order to perform the restructuring procedure, we actually define the common part of an arbitrary set of configurations in terms of a PF-LTS (representing the configured-transitions semantics of a product family). The conceptual structure of a PF-LTS, where the transitions are labeled with a set of configurations indicating when the transition is present, allows for a straight-forward definition of the common parts of a set of configurations. The definition is based on the simple idea that two (or more) configurations have a transition in common, if both (all) configurations are elements of the set of configuration labels of the transition. This allows to represent the common part of a set of products as a projection of the PF-LTS representing the entire product family. More precisely, the common part of a set of products is the projection of the product family which discards all those transitions (and states) that do not exist in all products.

In order to express this formally we introduce a slightly adjusted concept of projection. It extends the concept of projection as introduced in Definition 3.20 (cf. Chapter 3.2.3, Page 142) to deal with sets of configurations, instead of single configurations only.

Definition 5.1 (Projection of a PF-LTS According to a Set of Configurations). *Let $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \{\Longrightarrow_{\alpha} \mid \alpha \in \mathcal{A}\}, \sigma)$ be a PF-LTS (cf. Definition 3.18), and $C = \{\theta_i \mid \theta_i \in \{R, L, ?\}^n, i \in \mathbb{N}\}$ be a set of fitting configurations. The projection $\Pi_C(\mathcal{T})$ of \mathcal{T} according to C is defined as the PF-LTS*

$$\Pi_C(\mathcal{T}) = (\mathcal{S}_C, \mathcal{A}, \{\Longrightarrow_{\alpha}^C \mid \alpha \in \mathcal{A}\}, \sigma)$$

where

- $\Longrightarrow_{\alpha}^C = \{s \xrightarrow{\alpha, L'} s' \mid (s \xrightarrow{\alpha, L} s' \in \Longrightarrow_{\alpha}) \wedge (L' = \{\nu \in L \mid \forall \theta_i \in C : \theta_i \sqsubseteq \nu\} \wedge L' \neq \emptyset)\}$
- $\mathcal{S}_C \subseteq \mathcal{S}$ is the set of all states which are reachable from σ with respect to the transition relation $\Longrightarrow_{\alpha}^C$,
- \mathcal{A} is a set of communication actions,
- and $\sigma \in \mathcal{S}$ is the start state.

For a singleton set C containing only one complete configuration this kind of projection is equivalent to the original projection as introduced in Definition 3.20 (cf. Chapter 3.2.3, Page 142). Similarly to the original projection, this extended projection handles complete and incomplete configurations. However, while we have applied the original projection in Chapter 3 only with complete configurations in order to derive single products, we use the new kind of projection also with incomplete configurations, since we are not only interested in the common parts of complete products, but also in the common parts of sub-families (which still contain variation points themselves).

The transition relation which is preserved by the kind of projection of Definition 5.1 comprises only those transitions which actually exist in all specified configurations of C , while all other transitions are discarded. Regarding a single variation point \oplus_i , this kind of projection performed on the PF-LTS representing the configured-transitions semantics of the variation point allows to characterize the common part of its variants. More precisely, the projection of a PF-LTS $\llbracket P \oplus_i Q \rrbracket_{CT}$ according to a configuration set $C = \{\theta, \pi\}$, where $\theta_i = L$ and $\pi_i = R$, yields the PF-LTS which describes the common behavior of both variants of this particular variation point \oplus_i . This idea can be extended from single variation points to entire product families, which results in a definition of the common part of an entire product family for an arbitrary set of configurations. Thereby, it makes only sense to talk about the common part of a process which represents a true product family, i.e. which contains at least one variation point. In this light, the following definition defines the notion of common part for such a product family.

Definition 5.2 (Common Part of a Set of Configurations). *Let $Prog = (\mathcal{E}, P)$ be a product family containing n variation points. Given a set $C = \{\theta_i \mid \theta_i \in \{R, L, ?\}^n, i \in \mathbb{N}\}$ of fitting configurations, we call the PF-LTS*

$$\Pi_C(\llbracket Prog \rrbracket_{CT})$$

a common part of the configurations in C w. r. t. the product family $Prog$.

5. Restructuring PF-CCS Software Product Families

Although we have based the definition of common parts of a product family on the configured-transitions semantics, the concept of common parts can be directly transferred to the unfolded semantics, and defined in terms of the unfolded semantics, too. However, we have preferred the definition using the configured-transitions semantics, as it is the semantics which will be usually used “in practice”.

The common part of a set of configurations C is again a product family, i.e. the common part itself can still contain variable parts and variation points which are not configured, yet. Recall the front wiper example of the previous section (cf. Figure 5.8). In this example, the common part itself does not contain variation points anymore. However, this is not necessarily the case. Assume that the permanent wiping can be stopped not only by the driver by operating the turn switch (action *off*), but optionally also by the car itself whenever the car is parked and the engine is turned off (action *engOff*). Such a behavior is realized by the adjusted process $Perm$, which itself contains a second variation point \oplus_2 .

$$Perm \stackrel{def}{=} \overline{wipe}.Perm + off.FWS + (Nil \oplus_2 engOff.FWS)$$

Using this new version of $Perm$ (and of $Perm$, respectively) we can perform the same calculation as shown in Figure 5.7, and obtain the updated process $Perm$ as part of the commonalities of both wiper variants like in Figure 5.8. However, since the updated process $Perm$ now contains a variation point itself, i.e. variation point \oplus_2 , the common part now is a sub-family.

While we have defined common parts in terms of the semantic structure (PF-LTS) of a PF-CCS product family, the example in the preceding section shows that the commonalities between variants can be determined directly from the corresponding PF-CCS program representing the product family. According to Definition 5.2, the common part—being a sub-system of the original PF-LTS—“ends” in states whose incoming transition are labeled with a configuration set containing all relevant configurations, while the labels on any outgoing transition do not contain all relevant configurations anymore. Since the SOS rules for the variants operator (Figure 3.4b, Page 128) are the only rules which change the transition labels, such states can only be constructed as a consequence of parsing a subterm of the entire PF-CCS program which contains a variants operator. When determining the common part for a set C of configurations, a certain set of variation points is decisive in order to clearly specify this set C . The end of the common part of the configurations in C is reached in all states of the corresponding PF-LTS, which are constructed when the *last* of the decisive variants operators is parsed for the *first* time in the term hierarchy and the corresponding SOS rule is applied, respectively. Altogether, this gives us the connection between the formal definition of common parts based on a PF-LTS and its representation as a PF-CCS program. Using the PF-CCS representation, the common part of a set C of configurations starts with the start process and ends when the *last* decisive variants operator is parsed in the term hierarchy for the *first*

time. Coming back to the example in the preceding section, this now explains why we define the process shown in Equation 5.4 (Page 212) as a common part of both wiper variants.

The size and the structure of the common part of a set of configurations crucially depends on the representation of the product family which we use to determine the commonalities. In particular, the common part of a set of configurations is *not* unique for a product family. Recall the calculation of our example from the previous section shown in Figure 5.7. According to Definition 5.2, a common part of the two variants FWS and FWC using the initial representation (Line 1 in Figure 5.7) is given by

$$\Pi_{\{\langle L \rangle, \langle R \rangle\}}(\llbracket FWS \oplus_1 FWC \rrbracket_{CT})$$

However, constructing the corresponding PF-LTS shows that this common part consists only of the single state $(FWS \oplus_1 FWC, \langle ? \rangle)$, since already the outgoing transitions from this initial state of the PF-LTS are labeled with either $\langle L \rangle$ or $\langle R \rangle$, but not with both. Hence, no required outgoing transition exists for the start state, as all of them are discarded by the projection. In contrast, if we consider the common part of the same set of configurations using the representation which is the result of the restructuring , we get a common part given by

$$\Pi_{\{\langle L \rangle, \langle R \rangle\}}(\llbracket FWFam \rrbracket_{CT})$$

where the process $FWFam$ is specified as in Figure 5.8 (Page 213). This common part is obviously different as the construction of the corresponding PF-LTS $\llbracket FWFam \rrbracket_{CT}$ clearly shows. Here, the splitting of the transitions into some transitions labeled with $\langle L \rangle$ and $\langle R \rangle$ happens only after the application of some other SOS rules, finally caused by the SOS rules for the variants operator when parsing the subprocess

$$(\overline{wipe}.WaitL + intv2.Intv2) \oplus_1 (\overline{wipe}.WaitL + hvyRn.Fast)$$

shown in Line 2 of Figure 5.8.

Depending on the concrete representation which we use for a product family, the common part of a set of configurations varies. However, the common parts (stemming from different representations of the same product family) for the same set of configurations are related. Consider again the calculation shown in Figure 5.7 (Page 211). Each of the lines (2), (4), (5), (6), and (8) shows another representation of the same product family, where each one contains a common part. Following the calculation, we observe that the common part becomes “larger” from line to line, i.e. the common part comprises more and more processes when we compare the respective representations. In fact, processes which are already in the common part in one line are preserved in the common part in the following line, while the common part grows by adding further processes. Thereby, the “size” of the common part, i.e. the degree of commonalities, is determined by the nesting levels of the relevant variants

5. Restructuring PF-CCS Software Product Families

operators in the term hierarchy. This implies the existence of an ordering among the common parts of a set of configurations, and in particular suggests the notion of a *greatest* common part of a set of configurations. Since we perform the calculation of the greatest common part directly on a PF-CCS program we provide the characterization of the greatest common part also in terms of a PF-CCS program, and not on basis of the corresponding semantic domain, i.e. a PF-LTS.

Definition 5.3 (Representation Showing the Greatest Common Part). *Let $Prog = (\mathcal{E}, P)$ be a PF-CCS program representing a product family containing n variation points, and $C = \{\theta_i : \theta_i \in \{R, L, ?\}^n, i \in \mathbb{N}\}$ be a set of fitting configurations. We say that $Prog$ is the representation showing the greatest common part of C w.r.t. the corresponding product family, if we cannot apply any further distributive laws (Theorems 5.1 to 5.3) to the set of equations \mathcal{E} in order to factor out a variation point relevant for C .*

The ordering on the representations of the common parts of a set C of configurations can be observed on the corresponding PF-LTSs, too. Let $Prog_1$ and $Prog_2$ be two representations of the same product family, where $Prog_2$ is derived from $Prog_1$ by factoring out more common parts using the distributive laws. Then, the PF-LTS representing the common part according to representation $Prog_1$ is the “initial” part of the PF-LTS representing the common part according to $Prog_2$. We illustrate this using the example of the PF-LTSs shown in the uppermost line of Figure 5.3 on Page 201. The effect of factoring out the common part (here by applying the distributive law for the $+$ operator) corresponds to changing from the PF-LTS shown in Figure 5.3b to the one shown in Figure 5.3a. While all outgoing transitions from the initial state in Figure 5.3b have configuration labels which have concrete selections for the variants operator \oplus_i , the initial state in Figure 5.3a provides the transition α, ν whose transition label ν does not yet require a selection for one of the variants of the variants operator \oplus_i , as this selection decision is postponed to a successor of the state (P', ν) . This means that while the common part of the variants of the variation point \oplus_i in Figure 5.3b is only the initial state, the common part of the same variation point in Figure 5.3a is given by the initial state, the transition α, ν and the state (P', ν) . In particular, the PF-LTS representing the common part in Figure 5.3b is an initial subsystem¹ (modulo bisimulation) of the PF-LTS representing the common part in Figure 5.3a. The situation is similar for the remaining distributive laws. This means that by factoring out the common part using distributive laws, we continually enlarge the PF-LTS which represents the common part, until we can no longer apply any one of the distributive laws. Thus, in terms of the corresponding PF-LTSs, the common parts (according to different representations) of a set C of configurations are ordered in a subsystem¹ relation. In particular, any PF-LTS which represents a common part is a subsystem of the PF-LTS which is constructed using the representation of the greatest common part.

¹Let $\mathcal{T}_1 = (\mathcal{S}_1, \mathcal{A}, \rightarrow_1, \sigma_1)$ and $\mathcal{T}_2 = (\mathcal{S}_2, \mathcal{A}, \rightarrow_2, \sigma_2)$ be two PF-LTSs. We call \mathcal{T}_2 an *initial subsystem* of \mathcal{T}_1 iff $\mathcal{S}_2 \subseteq \mathcal{S}_1$, $\rightarrow_2 \subseteq \rightarrow_1$, and both system have the same start state, i.e. $\sigma_2 = \sigma_1$.

The ability to calculate the greatest common part of a set of configurations is the key to establish the connection between the two different views unto a product family: a view based purely on alternatives, i.e. emphasizing the information of the points *where* products differ, and a view using common and optional parts, i.e. emphasizing the information *how* products differ. While the “alternative” view corresponds to the concept of variations points (represented by the variants operator of PF-CCS), the “optional” view unto two variants is a special case of the alternative view, where one variant completely includes the behavior of the other one. The ability to calculate the greatest common part allows us to swap between both views arbitrarily, since the knowledge of the greatest common part allows to represent two alternative variants in a new form which consists of the common part of both, and those parts which are specific for each product. In the special case where one of the specific parts is *Nil*, we have the situation of optionality, where one variant comprises the entire behavior of the other variant (which has exactly the behavior of the common part) but additionally contains some extra behavior (the optional part), too. In a component-based development paradigm, the situation of having a characterization using such optional parts is extremely useful, as it allows to describe the differences between products as separate entities in a modular way. In particular, this is the basis to determine the atomic assets of which all products are built.

Let us close the chapter by emphasizing again the importance of the ability to calculate behavioral commonalities of products in a formal way. Since the beginning of the software crisis the idea of making use of the commonalities of programs and specifications is central to software engineering. Beside McIlroy, Dijkstra expresses this vision quite illustratively in the context of programs:

If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say, different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common.

Using our terminology, what Dijkstra calls the “common ancestor” matches exactly what we define as the greatest common part of the behavior of a set of PF-CCS processes. In this light, PF-CCS is an implementation of Dijkstra’s vision. However, while Dijkstra’s vision is more concerned with the (formal) representation of a common part, regarding the maximization of reuse, the ability to calculate the greatest common part in a schematic way is as important as the ability to specify it formally. Only with such a formally founded calculation mechanism is it possible to make use of a formal representation of the commonalities between variants in a realistic application scenario. With PF-CCS we have implemented this idea of restructuring as a central aspect beside the pure specification of commonalities.

Conclusion and Future Work

In this chapter we briefly recapitulate the main contributions of this thesis, and discuss them in the light of the challenges which we describe in the introduction of this thesis. In addition, we discuss some general design decisions of our theory, comprising for example the practical relevance of the PF-CCS framework, the relationship between PF-CCS and CCS, and the conceptual independence of product family specific concepts from a concrete underlying formalism like CCS. Finally, we present some of our ongoing and future work.

Contents

6.1. Discussion	225
6.2. Future and Ongoing Work	228

6. Conclusion and Future Work

With this thesis we contribute to the theoretical foundation of software product line engineering by (i) providing an axiomatization of general software product family concepts, and by (ii) introducing a framework, consisting of the process algebra PF-CCS and a multi-valued version of the modal μ -calculus, for the specification, verification, and restructuring of product families which correspond to the operational functionality of a family of software-intensive, reactive systems. The aim of the PF-CCS framework is to provide and demonstrate the fundamental concepts, and not to be applied overnight in the current development process of an automotive OEM. Accordingly, our work is a contribution to the theory of software product family development, rather than to the challenges of putting fundamental software product family concepts into practice. However, a conceptual and theoretical framework as we provide it in this thesis is the indispensable basis for the development of appropriate tools, a corresponding methodology, and the integration of software product family practices into the software development process.

The contributions that we have made in this thesis can be summarized in the following four major ones:

- We identify and formalize the general characteristics and fundamental construction concepts of software product families, independent of the concrete realization of the product family. We do this by elaborating an axiomatization of software product family concepts. Technically, the axiomatization is defined as an algebraic specification (cf. Chapter 2) which describes the specific operations, concepts, and laws of software product families in a formal way.
- We introduce the process algebraic framework PF-CCS (cf. Chapter 3) for the specification of the behavior of a family of similar systems in an implementation and platform independent way.
- We introduce the multi-valued modal μ -calculus $mv\text{-}\mathfrak{L}_\mu$ (cf. Chapter 4) which is tailored to specify properties of product families in general, and in particular of PF-CCS programs. Thereby, $mv\text{-}\mathfrak{L}_\mu$ properties can be evaluated using existing multi-valued model checking techniques.
- We introduce concepts and techniques to restructure a PF-CCS program (cf. Chapter 5). An important application of the restructuring concepts is to calculate the greatest common part (cf. Definition 5.3, Page 218) of a set of products.

The motivation for our approach, in particular for the PF-CCS framework, are challenges which have to be faced during the construction of complex, multi-functional systems. In Chapter 1.3 we have described these challenges using the concrete example of automotive systems and the automotive domain. Although these challenges become noticeable during practical system engineering, they stem from a lack

of fundamental, conceptual knowledge about the construction of software product families. With our contributions we realize the properties and concepts which we have identified in Chapter 1.2.1 as very relevant requirements for the improvement of the current development of families of similar systems. In the following we discuss the benefits of our contributions with respect to these requirements.

Benefit of the Axiomatization

With the axiomatization we formalize the fundamental concepts, and techniques that represent the underlying construction principle behind the development of a set of system as a software product family. In other words, the axiomatization defines precisely what a product family actually is, by formalizing the specific concepts and properties which hold in every software product family. In this sense, the axiomatization represents a standardization of software product family concepts which unifies the various notions of a software product family which exist in the software product line community, and which is the basis for a precise terminology.

Due to the realization as an algebraic specification, we can understand the axiomatization as a precise characterization of the class of computation structures that are “valid instances” of a software product family.

Moreover, although being given in an implementation-platform independent way, the axiomatization still guides the construction of new concrete product families, since it has an operational character in the sense that the axioms are all specified according to the scheme of primitive recursion. This means that the axiomatization can directly be implemented in a functional programming language such as for example ML [MTH90], and thus serve as a template for implementations.

Benefit of PF-CCS and its Restructuring Concept

PF-CCS provides the concepts for representing and modeling the operational functionality of families of software-intensive, reactive systems in a platform-independent way, and represents a specification formalism that supports the concept of behavioral variability in the deterministic sense of a software product family. Firstly, this means that we can use PF-CCS to specify the behavior of a family of functionally similar systems. For the automotive industry this offers completely new possibilities for the specification of the behavior of families of automotive systems, as the specification of a family of similar behaviors is not possible with the current state of the art in automotive software engineering.

PF-CCS allows to represent the connection between the operational behaviors of similar systems, facilitating to develop the operational behavior of similar systems in an coordinated way. In other words, PF-CCS provides all concepts to specify

6. Conclusion and Future Work

the operational behavior of a family of systems as a software product family. Due to the restructuring concept of PF-CCS we can determine for a given PF-CCS product family the common behavior of any (sub)set of its products. Thereby, the ability to restructure a product family is the essential property in order to work effectively with the commonalities of products. In fact, it is the fundamental requirement to benefit from commonalities. However, the shape and the size of a software product family are subject to frequent changes: new assets are added or replaced, old assets are removed, entire product families have to be merged, etc. These changes in the structure of a product family require techniques and methods to determine the commonalities of products or sub-families of a software product family at any time anew. As we have demonstrated in Chapter 5, the restructuring concept of PF-CCS realizes such a calculation of common parts. The idea of restructuring finally brings together the two “dual” views on the variability of a PF-CCS software product family: the view based on alternative behavior, and the view based on optional behavior. With the ability to calculate the greatest common part between two alternative PF-CCS products we can represent each PF-CCS product by a composition of the common part and a variant-specific part. This means that PF-CCS is a specification technique in which both concepts, optionality and alternativeness, can be used equally, since their relation is formally understood and precisely defined for PF-CCS software product families.

In PF-CCS, the ability to determine common behavioral parts is the basis for the realization of a reuse concept of operational functionality for families of similar systems. In particular, by the explicit separation between common parts and variant-specific parts (i.e. the differences between products), the PF-CCS formalism facilitates a planned reuse of artifacts. PF-CCS provides the conceptual basis to reuse (pieces of) implementation-independent, operational behavior. Instead of specifying the behaviors of two similar products separately, as for example by two individual CCS programs, PF-CCS facilitates to specify both products in an integrated way as a single PF-CCS program, where the points in which both products differ are explicitly modeled. In this scenario, reuse takes already place as part of the specification of the product family, since whenever a new variation point is added to an existing PF-CCS program, parts of the existing program, i.e. the common parts of the existing program and the new variants, are implicitly reused. With respect to the development process this means that PF-CCS allows reuse of artifacts in a very early stage of the development, long before implementation and platform specific details are considered. As we have discussed in the introduction, this matches the needs of industrial practice, since reuse is mainly functionality-driven, i.e. usually functionality has to be reused, independently of its concrete implementation.

Benefit of the Multi-Valued modal μ -Calculus

The multi-valued version of the modal μ -calculus which we have introduced in Chapter 4 allows to verify an entire PF-CCS product family, and to determine all those

products of a product family that fulfill a certain property. So far, without the multi-valued μ -calculus, regarding the operational behavior of a family of similar systems, this question can only be answered by inspecting the products individually, since standard two-valued logics (e.g. μ -calculus, LTL) cannot be used to reason about product families directly, as their semantics is only defined for regular Kripke structures but not for PF-LTSs representing product families. Thus, our main contribution in this context is to define a logic which operates directly on the model of a product family, which consequently allows to reason about commonalities and differences of the products, directly, and which most importantly yields not only a *true/false*-result, but the set of configurations that fulfill the desired property. Moreover, since the model of the product family is the central model in the PF-CCS approach—the models of every product can be derived automatically from the product family—it is essential to provide a logic that allows to reason about this central model.

Regarding the practical application of the multi-valued μ -calculus, the evaluation of formulae does not require new techniques or algorithms. As we have shown in Chapter 4.2, the evaluation can be done using existing model checking techniques which have to be adjusted only marginally. Since these techniques operate directly on the model of the product family, they exploit commonalities between the products for the verification.

6.1. Discussion

Operational Semantics of PF-CCS

The decision to provide an operational semantics for PF-CCS specifications was an intentional step towards the practical applicability of PF-CCS. For the practitioner, a central question is how to derive an implementation from the models that are used in a model-based development process. With an operational semantics which is given in terms of labeled transition systems (that can easily be understood as I/O-automata), a PF-CCS specification is already very close to an implementation. In fact, in a model-based development process it is a matter of generators to construct a platform-specific implementation from that kind of transition systems which form the semantic domain for the operational semantics of PF-CCS. As an alternative to an operational semantics, we could have also defined the meaning of PF-CCS programs by providing a denotational [ZLWL07] or an axiomatic semantics. Both kinds of semantics are frequently found in the area of process algebras, in particular for Hoare’s CSP, too. However, compared to an operational semantics, a denotational semantics describes the effects of performing a PF-CCS program in terms of mathematical objects, rather than defining a sequence of computational steps which

6. Conclusion and Future Work

reflect the execution of a PF-CCS program (like an operational semantics does). A similar situation is given for an axiomatic semantics, which aims rather at deriving new laws from existing axioms. In this light, an operational semantics is the closest to an implementation as we can get, and thus is the most interesting kind of semantics from a practical point of view.

Relationship between PF-CCS and CCS

With PF-CCS we have made the decision for one concrete process algebra, CCS. This means that PF-CCS takes over the specific concepts of CCS, for example synchronous communication, and consequently has to deal with all the specific challenges, concepts and problems of CCS. However, the question arises whether CCS is necessarily required in order to realize the software product family concepts, or whether others process algebras can also be the basis for a formalism like PF-CCS, as well. It was this question that motivated us to specify software product family concepts in a completely independent and self-contained way, using a mathematically solid approach: an axiomatization of software product family concepts, given as an algebraic specification of the abstract data type $\text{SPF } \alpha$ representing a software product family. With the axiomatization we have created a vehicle that allows us to formally demonstrate that the software product family concepts implemented in PF-CCS are independent of a concrete underlying process algebra—CCS in our case—and that the software product family concepts can be combined with other process algebras, like for example CSP, as well. Similarly, on basis of the axiomatization we can show that product family concepts are independent of the choice for synchronous or asynchronous communication.

The axiomatization characterizes the essential properties and concepts that constitute a software product family in an implementation and paradigm independent way, and specifies the fundamental requirements which any software product family has to fulfill, independently of its concrete realization. In this light, the PF-CCS framework is just one instance of a software product family that conforms to this axiomatization. In particular, the axiomatization provides a formal way of falsification: any formalism or modeling approach for which we can show that it violates one of the axioms of the algebraic specification is not an instance of a software product family formalism in our sense. In that sense the axiomatization represents a blueprint for the integration of software product family aspects into existing formalisms and modeling techniques.

With respect to the axiomatization, PF-CCS is a software product family approach. The variants operator of PF-CCS matches exactly the variants operator as characterized in the axiomatization. The remaining operators of the axiomatization are realized by the CCS portion of PF-CCS, i.e. we realize

- atomic assets, i.e. the function `asset` (Chapter 2.2.2), by the CCS concept of process and (process) constant definitions,
- the composition operator `||` (Chapter 2.2.2) by the composition operations `||`, `+`, and action prefixing `“.”`, respectively, and
- the neutral element `ntrl` (Chapter 2.2.3, Page 34) by the idle process *Nil*.

In particular, for these CCS concepts and operations, we have shown in the Chapters 3 and 5 that they realize the corresponding operators of the axiomatization in a way that respects the corresponding laws. If we compare other process algebras, for example CSP, with CCS, we see that CSP provides equivalent operations (with an agreeing semantics). Thus, an extension of CSP with the software product family concepts is also possible, such that this extension offers all operations and respects all laws required in the axiomatization, too. As the properties specified in the axiomatization do not require any particular communication paradigm, it is also possible to extend a specification technique based on asynchronous communication with software product family concepts. In this light, the decision for CCS was not mandatory.

Beside the “original” operators of CCS, PF-CCS provides only one more operator—the variants operator—in order to realize the modeling of alternative variants. Here, the question arises whether the variants operator is actually required as an additional, new operator, or whether we could have realized the concept of alternative variants with existing CCS operators and mechanisms, too. In particular, since the non-deterministic choice operator `+` represents also a kind of selection between different processes, can we “simulate” the variants operator by the non-deterministic choice operator? The answer is no. The kind of choice between two alternative variants which is represented by the variants operator is conceptually different to a non-deterministic choice: While the selection for one variant of the variants operator is always performed *in the same way* according to the corresponding configuration, the selection for one process of a non-deterministic choice can vary. As we have already discussed in detail in Chapter 3.2.2, in particular in the context of recursive process definitions, this is a fundamental difference. Consider for example the recursive program

$$\begin{aligned} P &\stackrel{def}{=} \alpha.T \oplus_1 \beta.T \\ T &\stackrel{def}{=} \gamma.P \end{aligned}$$

If we chose one variant, say the right variant $\beta.T$, the intention is that in any subsequent recursive pass of P we cannot undo the preceding configuration choice R and suddenly select the left variant $\alpha.T$. In contrast, the non-deterministic choice operator does not guarantee such a property. Consider for example the process

$$\begin{aligned} P &\stackrel{def}{=} \alpha.T + \beta.T \\ T &\stackrel{def}{=} \gamma.P \end{aligned}$$

6. Conclusion and Future Work

While in a first round the left process $\alpha.T$ can be selected non-deterministically, in a subsequent recursive entry the right process $\beta.T$ can be selected. Clearly, this contradicts the axiomatization of a software product family, in particular the requirement that a configuration selection has to be deterministic, resulting always in the same product, no matter when it is performed. Thus, the non-deterministic choice operator cannot simulate the variants operator.

Moreover, the distributive laws which we require for a variants operator do not hold for the non-deterministic choice operator. In Chapter 5.1.2 we have demonstrated that the laws

$$\alpha.P \oplus_i \alpha.Q = \alpha.(P \oplus_i Q) \quad (\text{cf. Theorem 5.1})$$

and

$$P \parallel (Q \oplus_i S) = (P \parallel Q) \oplus_i (P \parallel S) \quad (\text{cf. Theorem 5.3})$$

hold, while analogue laws for the non-deterministic choice, e.g. $\alpha.P + \alpha.Q = \alpha.(P + Q)$, do not hold. Thus, the non-deterministic choice cannot “simulate” the variants operator with respect to these laws, either. In summary, these fundamental conceptual differences between the concept of alternative variants in a software product family and the concept of a non-deterministic choice suggest the existence of a variants operator as an additional, independent operator.

6.2. Future and Ongoing Work

Our future work focuses on improving the applicability of PF-CCS by providing concepts to deal with, and verify large PF-CCS software product families. We have already made a first step in this direction with the introduction of a general abstraction concept [CGLT09] for software product families, which is in particular applicable for PF-CCS product families, too. The abstraction formalism establishes the theoretical basis to deal also with product families containing a high degree of variability by reducing the state space as well as the sets of possible configurations of a product family. We briefly introduce the abstraction concept in the following.

Abstraction Techniques for SPFs

As part of our ongoing work we focus on abstraction and refinement techniques for software product families. Abstraction techniques improve the scalability of the PF-CCS theory, but in particular also increase the practical applicability of verification techniques such as multi-valued model checking.

Usually, a PF-CCS software product family is a very large and comprehensive model which comprises the behavior of all of its products. As we have seen, in our PF-CCS approach, the product family is the central model for verification. However, for the verification of properties we are not always interested in all aspects of the entire product family in their very details, but very often can/want to abstract from some parts of the product family. In particular with respect to multi-valued model checking, abstraction is an effective remedy for the state explosion problem.

Abstraction techniques for two-valued transition systems usually base on the fact of joining states and thereby decreasing the size of the abstract system. In contrast to the two-valued setting, for a multi-valued system like a PF-CCS product family we identify two dimensions of abstraction for the corresponding PF-LTS: Abstraction by joining states of the PF-LTS as well as abstraction by joining configurations. In [CGLT09], we have introduced an abstraction concept for multi-valued transition systems which realizes these two dimensions. The abstraction yields an abstract product family, which itself can be represented as a multi-valued Kripke structure (i.e. a PF-LTS), and for which all model checking concepts apply in the same way as for the concrete product family or any other multi-valued Kripke structure.

The abstract product family follows both an *optimistic* and *pessimistic* account for each dimension of abstraction. The optimistic account corresponds to an over-approximation of the system, in which all those configurations are considered in which a transition *may* be present, while the pessimistic account corresponds to an under-approximation, in which transitions *must* be present for a given set of configurations. We show that this notion of abstraction is *conservative* in the following sense: The set of configurations which fulfill a property in a concrete system is “between” the optimistic and pessimistic assessment of the abstract system. Moreover, whenever the optimistic and pessimistic model checking result differ, the *cause* [CGLT09] for such an assessment is identified, allowing the abstraction to be refined to eventually yield a result for which both the optimistic and pessimistic assessment coincide.

From a practical point of view, the applicability of multi-valued model checking in a realistic, industrial environment greatly benefits from such an abstraction concept, as it allows to reduce the state space of the system dramatically. With respect to future work in the area of abstraction, our next steps will be to apply the abstraction concepts on a realistic case study in order to give a proof of concept, but also to measure the gain for model checking when using the abstract system. Along with such a case study we want to provide a complexity analysis of the multi-valued model checking algorithm which we use for the evaluation of $mv\text{-}\mathcal{L}_\mu$ -properties of PF-CCS programs as introduced in Chapter 4.

Tool Support

Finally, another important aspect for our PF-CCS approach is to come up with an appropriate tool support. Although we provide a solution for the specification, veri-

6. *Conclusion and Future Work*

fication and handling of software product families in this thesis on a conceptual level, these concepts and techniques will not be able to enter industrial practice without such an adequate tool support. In the context of PF-CCS this means to provide (i) a tool for the specification of a PF-CCS product family, and (ii) to integrate this tool with an implementation of a multi-valued model checking algorithm which is appropriate for such PF-CCS specifications. However, since the aim and the contribution of this thesis is to understand and model the idea and concepts behind a software product family, we have delayed the construction of tools to our future work.

Selected Algebraic Specifications

In the following we list the algebraic specifications of those sorts which are used and imported in the specification of the sort `SPF α` , given by the algebraic specification `SOFTWAREPRODUCTFAMILY` shown in Figure 2.11, Section 2.2.4 (Page 90).

The algebraic specifications of the sorts `Bool`, `Nat`, `Seq α` , `Set α` , and `MSet α` define the fundamental sorts of boolean values, of natural numbers, of sequences, of (ordinary) sets, and of multiset (bags), together with their respective operations and laws. They are given in Figures A.1, A.3, A.4, A.5, and A.6, respectively.

For the normal form which we introduce in Section 2.2.3.2 as part of the axiomatization we require some auxiliary functions on multisets. These auxiliary functions are specified as an extension to the specification `MULTISET` in the separate specification `EXT_MULTISET` shown in Figure A.7.

Within the specification `SOFTWAREPRODUCTFAMILY` we make use of a *conditional if* function. Such a conditional function can itself be defined in an algebraic fashion. The corresponding axiomatization is given in a stand-alone algebraic specification which is shown in Figure A.2.

A. Selected Algebraic Specifications

```

SPEC BOOL =
{ defines sort Bool,

      true, false : Bool,
      ¬ : Bool → Bool,           Prefix
      _ ∨ _ , _ ∧ _ : Bool, Bool → Bool,   Infix

  Bool generated_by true, false,

  not (true = false),

  ¬(true) = false,
  ¬(false) = true,
  ¬(¬(x)) = x,

  (false ∨ x) = (x ∨ false) = x,
  (true ∨ x) = (x ∨ true) = true,
  (true ∧ x) = (x ∧ true) = x,
  (false ∧ x) = (x ∧ false) = false
}

```

Figure A.1.: Algebraic specification of the sort Bool, representing boolean values.

```

SPEC CONDITIONAL_IF =
{ defines sort If-then-else,
  based_on BOOL,

  if : Bool, α, α → α,           Prefix

  if(true, x, y) = x,
  if(false, x, y) = y,
}

```

Figure A.2.: Algebraic specification of the sort If-then-else, representing the conditional if-function.

```

SPEC NAT =
{ defines sort Nat,
  based_on BOOL

    0 : Nat,
    succ : Nat → Nat,
    pred : Nat → Nat,
    iszero : Nat → Bool,
    _ + _ : Nat, Nat → Nat,      Infix
    _ * _ : Nat, Nat → Nat,      Infix
    =Nat : Nat, Nat → Bool,     Infix

  Nat generated_by 0, succ,

  iszero(0) = true,
  iszero(succ(x)) = false,

  pred(succ(x)) = x,

  0 + y = y,
  succ(x) + y = succ(x + y),
  0 * y = 0,
  succ(x) * y = y + (x * y),

  (0 =Nat 0) = true
  (succ(x) =Nat succ(y)) = (x =Nat y)
}

```

Figure A.3.: Algebraic specification of the sort `Nat`, representing the natural numbers.

```

SPEC SEQ =
{
  defines sort Seq  $\alpha$ ,
  based_on BOOL, NAT

       $\langle \rangle$  : Seq  $\alpha$ ,
       $\langle \_ \rangle$  :  $\alpha \rightarrow$  Seq  $\alpha$ ,   Mixfix
       $\_ \circ \_$  : Seq  $\alpha$ , Seq  $\alpha \rightarrow$  Seq  $\alpha$ ,   Infix
  iseseq : Seq  $\alpha \rightarrow$  Bool,
  first, last : Seq  $\alpha \rightarrow \alpha$ ,
  head, rest : Seq  $\alpha \rightarrow$  Seq  $\alpha$ ,
  get : Seq  $\alpha$ , Nat  $\rightarrow$  Seq  $\alpha$ ,

  Seq  $\alpha$  generated_by  $\langle \rangle$ ,  $\langle \_ \rangle$ ,  $\circ$ ,

  iseseq( $\langle \rangle$ ) = true
  iseseq( $\langle a \rangle$ ) = false
  iseseq( $x \circ y$ ) = (iseseq( $x$ )  $\wedge$  iseseq( $y$ ))

   $x \circ \langle \rangle = x = \langle \rangle \circ x$ 
  ( $x \circ y$ ) $\circ z = x \circ (y \circ z)$ 

  first( $\langle a \rangle \circ x$ ) =  $a$ 
  last( $x \circ \langle a \rangle$ ) =  $a$ 
  head( $x \circ \langle a \rangle$ ) =  $x$ 
  rest( $\langle a \rangle \circ x$ ) =  $x$ 

  get( $\langle \rangle$ ,  $n$ ) =  $\langle \rangle$ 
  get( $\langle a \rangle$ , 1) =  $a$ 
   $\neg$ iseseq( $x$ )  $\Rightarrow$  (get( $x$ ,  $n$ ) = get(rest( $x$ ), pred( $n$ )))
}

```

Figure A.4.: Algebraic specification of the sort `Seq α` , representing sequences of elements of sort α . The function `get` realizes an indexed access to the elements of the sequence, similarly to an array.

```

SPEC SET =
{
  defines sort Set  $\alpha$ ,
  based_on BOOL,

   $\emptyset$  : Set  $\alpha$ ,
  add : Set  $\alpha$ ,  $\alpha \rightarrow$  Set  $\alpha$ ,
  del : Set  $\alpha$ ,  $\alpha \rightarrow$  Set  $\alpha$ ,
  iseset : Set  $\alpha \rightarrow$  Bool,
  iselem : Set  $\alpha$ ,  $\alpha \rightarrow$  Bool,

  Set  $\alpha$  generated_by  $\emptyset$ , add,

  iseset( $\emptyset$ ) = true,
  iseset(add( $s$ ,  $x$ )) = false,

  iselem( $\emptyset$ ,  $x$ ) = false,
  iselem(add( $s$ ,  $x$ ),  $x$ ) = true,
  iselem( $s$ ,  $x$ ) = true  $\Rightarrow$  iselem(add( $s$ ,  $y$ ),  $x$ ) = true,
   $x \neq y \Rightarrow$  iselem(add( $s$ ,  $y$ ),  $x$ ) = iselem( $s$ ,  $x$ ),

  del( $\emptyset$ ,  $x$ ) =  $\emptyset$ ,
  del(add( $s$ ,  $x$ ),  $x$ ) = del( $s$ ,  $x$ ),
   $x \neq y \Rightarrow$  del(add( $s$ ,  $y$ ),  $x$ ) = add(del( $s$ ,  $x$ ),  $y$ ),

  add(add( $s$ ,  $x$ ),  $y$ ) = add(add( $s$ ,  $y$ ),  $x$ ),
  iselem( $s$ ,  $x$ ) = true  $\Rightarrow$  add( $s$ ,  $x$ ) =  $s$ ,
}

```

Figure A.5.: Algebraic specification of the sort Set α , representing the axiomatization of a set (of elements of the same sort).

```

SPEC MULTISSET =
{ defines sort MSet  $\alpha$ ,
  based_on BOOL,

     $\emptyset$  : MSet  $\alpha$ ,
    add : MSet  $\alpha$ ,  $\alpha \rightarrow$  MSet  $\alpha$ ,
    del : MSet  $\alpha$ ,  $\alpha \rightarrow$  MSet  $\alpha$ ,
    iseset : MSet  $\alpha \rightarrow$  Bool,
    iselem : MSet  $\alpha$ ,  $\alpha \rightarrow$  Bool,

  MSet  $\alpha$  generated_by  $\emptyset$ , add,

  iseset( $\emptyset$ ) = true,
  iseset(add( $s$ ,  $x$ )) = false,

  iselem( $\emptyset$ ,  $x$ ) = false,
  iselem(add( $s$ ,  $x$ ),  $x$ ) = true,
  iselem( $s$ ,  $x$ ) = true  $\Rightarrow$  iselem(add( $s$ ,  $y$ ),  $x$ ) = true,
   $x \neq y \Rightarrow$  iselem(add( $s$ ,  $y$ ),  $x$ ) = iselem( $s$ ,  $x$ ),

  add(add( $s$ ,  $x$ ),  $y$ ) = add(add( $s$ ,  $y$ ),  $x$ ),

  del( $\emptyset$ ,  $x$ ) =  $\emptyset$ ,
  del(add( $s$ ,  $y$ ),  $x$ ) =  $\begin{cases} s & , x = y \\ \text{add}(\text{del}(s, x), y) & , \text{else} \end{cases}$ 
}

```

Figure A.6.: Algebraic specification of the sort MSet α , representing the axiomatization of a multiset.

SPEC EXT_MULTISSET =

{ **based_on** MULTISSET,

$\{-\}$: $\alpha \rightarrow \text{MSet } \alpha$,

Represents the creation of singleton sets. The function is simply an abbreviation for adding a single element to the empty set.

$_{-}\cup_{-}$: $\text{MSet } \alpha, \text{MSet } \alpha \rightarrow \text{MSet } \alpha$, Default set union for multisets. Infix.

cmn : $\text{MSet } \alpha, \text{MSet } \alpha \rightarrow \text{MSet } \alpha$,

The operation $\text{cmn}(s, t)$ returns (a multiset of) those elements which appear in both multisets s and t . Thereby, the number of occurrences of single elements is preserved, i.e. $\text{cmn}(\{a, a, a\}, \{a, a\})$ yields $\{a, a\}$, and not only the singleton set $\{a\}$.

diff : $\text{MSet } \alpha, \text{MSet } \alpha \rightarrow \text{MSet } \alpha$,

The operation $\text{diff}(s, t)$ realizes a set minus operation on multisets, e.g. $\text{diff}(s, t)$ returns a multiset of those elements which appear in s but not in t .

$\{x\} = \text{add}(\emptyset, x)$,

$\emptyset \cup s = s$,

$\text{add}(s, a) \cup t = s \cup \text{add}(t, a)$,

$s \cup t = t \cup s$,

$\text{cmn}(s, t) = \text{cmn}(t, s)$,

$\text{cmn}(\emptyset, s) = \emptyset = \text{cmn}(t, \emptyset)$,

$\text{cmn}(\{a\}, \text{add}(t, b)) = \begin{cases} \{a\} & , a = b \\ \text{cmn}(\{a\}, t) & , a \neq b \end{cases}$,

$\neg\text{isaset}(s) \Rightarrow$

$(\text{cmn}(\text{add}(s, a), \text{add}(t, b)) = \text{cmn}(\{a\}, \text{add}(t, b)) \cup \text{cmn}(s, \text{del}(\text{add}(t, b), a)))$,

$\text{diff}(\emptyset, s) = \emptyset$,

$\text{diff}(s, \emptyset) = s$,

$\text{diff}(\text{add}(s, a), \{b\}) = \begin{cases} s & , a = b \\ \text{add}(\text{diff}(s, \{b\}), a) & , a \neq b \end{cases}$,

$\neg\text{isaset}(t) \Rightarrow$

$\text{diff}(\text{add}(s, a), \text{add}(t, b)) = \text{diff}(\text{diff}(\text{add}(s, a), \{b\}), t)$

}

Figure A.7.: Extension of the algebraic specification of sort MSet.

Uniqueness of the Normal Form: Proofs

In this section we present the main part of the proof of Theorem 2.2 (Chapter 2.2.3.2, Page 59), which states that the normal form realized by the function `NF` is unique. The essential idea of this proof is the fact that the function `norm` always yields the same term when applied to the left-hand and right-hand side of terms that represent a constructor axioms, respectively. We have decided to source out the main part of the proof to this chapter in the Appendix as it consists of five individual proofs which are rather long, standard structural inductions and thus purely technical. Each of the following proofs shows for the function `norm` one of the identities for the term representations that are established by a single constructor axiom as introduced in Theorem 2.2, respectively. Throughout the proofs we use the abbreviations *lhs*, and *rhs* to denote the left-hand, and the right-hand side of the identities, respectively. Numbers on top of equal signs refer to the laws of Section 2.2.3.2. The auxiliary Lemmas B.1–B.6 which we need during some of the proofs are presented at the end of this section.

Proof. We show the identity

$$\text{norm}(P \parallel \text{ntrl}) = \text{norm}(P)$$

of Theorem 2.2 by structural induction on the term structure of P .

Base cases:

B. Uniqueness of the Normal Form: Proofs

- $P = \text{ntrl}$: $lhs: \text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$
 $rhs: \text{norm}(\text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$
- $P = \text{asset}(a)$: $lhs: \text{norm}(\text{asset}(a) \parallel \text{ntrl})$
 $\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a))))$
 $\stackrel{2.16}{=} \text{reconv}(\text{sort}(\{\text{asset}(a)\}))$
 $\stackrel{\text{sorting}}{=} \text{reconv}(\langle \text{asset}(a) \rangle)$
 $\stackrel{2.37}{=} \text{asset}(a)$
 $rhs: \text{norm}(\text{asset}(a)) \stackrel{2.11}{=} \text{asset}(a)$

Inductive step:

- $P = Q \parallel R$: Showing: $\text{norm}((Q \parallel R) \parallel \text{ntrl}) = \text{norm}(Q \parallel R)$
 Induction hypothesis: $\text{norm}(Q \parallel \text{ntrl}) = \text{norm}(Q)$ and
 $\text{norm}(R \parallel \text{ntrl}) = \text{norm}(R)$

Case differentiation according to the cases of Law 2.12:

Case 1, $\text{norm}(Q \parallel R) = \text{ntrl}$:

$$lhs: \text{norm}((Q \parallel R) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$$

$$rhs: \text{norm}(Q \parallel R) = \text{ntrl} \text{ according to assumption.}$$

Case 2, $\text{norm}(Q \parallel R) \neq \text{ntrl}$:

$$lhs: \text{norm}((Q \parallel R) \parallel \text{ntrl}) = \text{reconv}(\text{sort}(\text{coll}(Q \parallel R)))$$

$$rhs: \text{norm}(Q \parallel R)$$

Case differentiation according to the 3 possible cases of Law 2.12:

Case i, $(\text{norm}(Q) \neq \text{ntrl}) \wedge (\text{norm}(R) \neq \text{ntrl})$:

$$lhs: \text{reconv}(\text{sort}(\text{coll}(Q \parallel R)))$$

$$rhs: \text{norm}(Q \parallel R) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \parallel R)))$$

Case ii, $(\text{norm}(Q) = \text{ntrl}) \wedge (\text{norm}(R) \neq \text{ntrl})$:

$$lhs: \text{reconv}(\text{sort}(\text{coll}(Q \parallel R)))$$

$$\stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(R)))$$

$$rhs: \text{norm}(Q \parallel R)$$

$$\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R)))$$

Case iii, $(\text{norm}(Q) \neq \text{ntrl}) \wedge (\text{norm}(R) = \text{ntrl})$:

$$lhs: \text{reconv}(\text{sort}(\text{coll}(Q \parallel R)))$$

$$\stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))$$

$$rhs: \text{norm}(Q \parallel R)$$

$$\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))$$

B. Uniqueness of the Normal Form: Proofs

- $P = \text{ntrl}$: lhs : $\text{norm}(\text{ntrl} \oplus_i \text{ntrl}) \stackrel{2.13}{=} \text{ntrl}$
since $\text{norm}(\text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$.
 rhs : $\text{norm}(\text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$
- $P = \text{asset}(a)$: lhs : $\text{norm}(\text{asset}(a) \oplus_i \text{asset}(a)) \stackrel{2.13}{=} \text{asset}(a)$
since $\text{norm}(\text{asset}(a)) \stackrel{2.11}{=} \text{asset}(a)$.
 rhs : $\text{norm}(\text{asset}(a)) \stackrel{2.11}{=} \text{asset}(a)$

Inductive step:

- $P = Q \parallel R$: Showing: $\text{norm}((Q \parallel R) \oplus_i (Q \parallel R)) = \text{norm}(Q \parallel R)$
Case differentiation according to the cases of Law 2.13:
Only the first case of Law 2.13 applies, since $\text{norm}(Q \parallel R) = \text{norm}(Q \parallel R)$.
This means that: $\text{norm}((Q \parallel R) \oplus_i (Q \parallel R)) \stackrel{2.13}{=} \text{norm}(Q \parallel R)$
- $P = Q \oplus_i R$:
The case follows with the same argumentation as in the case before.

□

Proof. We show the identity

$$\text{norm}(P \parallel Q) = \text{norm}(Q \parallel P)$$

of Theorem 2.2 by structural induction on the term structure of P .

Base cases:

- $P = \text{ntrl}$:
Case differentiation according to the two possible cases of Law 2.12.
Case 1, $Q = \text{ntrl}$:
 lhs : $\text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$
 rhs : $\text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$
Case 2, $Q \neq \text{ntrl}$:
 lhs : $\text{norm}(\text{ntrl} \parallel Q) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))$
 rhs : $\text{norm}(Q \parallel \text{ntrl}) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))$
- $P = \text{asset}(a)$:
Case differentiation according to the two possible cases of Law 2.12.

Case 1, $Q = \text{ntrl}$:

$$\begin{aligned} \text{lhs: } & \text{norm}(\text{asset}(a) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)))) \\ \text{rhs: } & \text{norm}(\text{ntrl} \parallel \text{asset}(a)) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)))) \end{aligned}$$

Case 2, $Q \neq \text{ntrl}$: W.l.o.g. we assume that $\text{coll}(Q) = \{q_1, \dots, q_n\}$ where $\forall i \in \{1, \dots, n\} : a <_{\alpha} q_i$.

$$\begin{aligned} \text{lhs: } & \text{norm}(\text{asset}(a) \parallel Q) \\ & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel Q))) \\ & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q))) \\ & \stackrel{2.16}{=} \text{reconv}(\text{sort}(\{\text{asset}(a)\} \cup \text{coll}(Q))) \\ & \text{properties} \\ & \text{of sorting} \\ & \stackrel{=}{=} \text{reconv}(\langle \text{asset}(a) \rangle \circ \text{sort}(\text{coll}(Q))) \\ \text{rhs: } & \text{norm}(Q \parallel \text{asset}(a)) \\ & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \parallel \text{asset}(a)))) \\ & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(\text{asset}(a)))) \\ & \stackrel{2.16}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \{\text{asset}(a)\})) \\ & \text{properties} \\ & \text{of sorting} \\ & \stackrel{=}{=} \text{reconv}(\langle \text{asset}(a) \rangle \circ \text{sort}(\text{coll}(Q))) \end{aligned}$$

Inductive step:

- $P = R \parallel S$: Showing: $\text{norm}((R \parallel S) \parallel Q) = \text{norm}(Q \parallel (R \parallel S))$
 Induction hypothesis: $\text{norm}(R \parallel Q) = \text{norm}(Q \parallel R)$ and $\text{norm}(S \parallel Q) = \text{norm}(Q \parallel S)$
 W.l.o.g. we assume that $\text{coll}(Q) = \{q_1, \dots, q_n\}$, $\text{coll}(R) = \{r_1, \dots, r_m\}$, and $\text{coll}(S) = \{s_1, \dots, s_l\}$ with the ordering $q_1 <_{\alpha} \dots <_{\alpha} q_n <_{\alpha} r_1 <_{\alpha} \dots <_{\alpha} r_m <_{\alpha} s_1 <_{\alpha} \dots <_{\alpha} s_l$.

Case differentiation according to the two possible cases of Law 2.12.

Case 1, $Q = \text{ntrl}$:

$$\begin{aligned} \text{lhs: } & \text{norm}((R \parallel S) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R \parallel S))) \\ \text{rhs: } & \text{norm}(\text{ntrl} \parallel (R \parallel S)) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R \parallel S))) \end{aligned}$$

Case 2, $Q \neq \text{ntrl}$:

$$\begin{aligned} \text{lhs: } & \text{norm}((R \parallel S) \parallel Q) \\ & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((R \parallel S) \parallel Q))) \\ & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(R \parallel S) \cup \text{coll}(Q))) \\ \text{rhs: } & \text{norm}(Q \parallel (R \parallel S)) \\ & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \parallel (R \parallel S)))) \\ & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R \parallel S))) \end{aligned}$$

Case differentiation according to the three cases of Law 2.17.

B. Uniqueness of the Normal Form: Proofs

Case i, $S = \text{ntrl} \wedge R \neq \text{ntrl}$:

$$\begin{array}{ll}
 \text{lhs:} & \text{reconv}(\text{sort}(\text{coll}(R \parallel \text{ntrl}) \cup \text{coll}(Q))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(R) \cup \text{coll}(Q))) \\
 & \text{sorting and} \\
 & \text{assumption} \\
 & \stackrel{=}{=} \text{reconv}(\langle q_1, \dots, q_n, r_1, \dots, r_m \rangle) \\
 \text{rhs:} & \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R \parallel \text{ntrl}))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R))) \\
 & \text{sorting and} \\
 & \text{assumption} \\
 & \stackrel{=}{=} \text{reconv}(\langle q_1, \dots, q_n, r_1, \dots, r_m \rangle)
 \end{array}$$

Case ii, $R = \text{ntrl} \wedge S \neq \text{ntrl}$:

The case follows with a similar argumentation as in the case above.

Case iii, $S \neq \text{ntrl} \wedge R \neq \text{ntrl}$:

$$\begin{array}{ll}
 \text{lhs:} & \text{reconv}(\text{sort}(\text{coll}(R \parallel S) \cup \text{coll}(Q))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(R) \cup \text{coll}(S) \cup \text{coll}(Q))) \\
 & \text{sorting and} \\
 & \text{assumption} \\
 & \stackrel{=}{=} \text{reconv}(\langle q_1, \dots, q_n, r_1, \dots, r_m, s_1, \dots, s_l \rangle) \\
 \text{rhs:} & \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R \parallel S))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R) \cup \text{coll}(S))) \\
 & \text{sorting and} \\
 & \text{assumption} \\
 & \stackrel{=}{=} \text{reconv}(\langle q_1, \dots, q_n, r_1, \dots, r_m, s_1, \dots, s_l \rangle)
 \end{array}$$

- $P = R \oplus_i S$: Showing: $\text{norm}((R \oplus_i S) \parallel Q) = \text{norm}(Q \parallel (R \oplus_i S))$
 Induction hypothesis: $\text{norm}(R \parallel Q) = \text{norm}(Q \parallel R)$ and $\text{norm}(S \parallel Q) = \text{norm}(Q \parallel S)$

$$\begin{array}{ll}
 \text{lhs:} & \text{norm}((R \oplus_i S) \parallel Q) \\
 \text{rhs:} & \text{norm}(Q \parallel (R \oplus_i S))
 \end{array}$$

Case differentiation according to the two possible cases of Law 2.12.

Case 1, $Q = \text{ntrl}$:

$$\begin{array}{ll}
 \text{lhs:} & \text{norm}((R \oplus_i S) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R \oplus_i S))) \\
 \text{rhs:} & \text{norm}(\text{ntrl} \parallel (R \oplus_i S)) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R \oplus_i S)))
 \end{array}$$

Case 2, $Q \neq \text{ntrl}$:

Case differentiation according to the two cases of Law 2.12 for $R \oplus_i S$.

Case i, $\text{norm}(R \oplus_i S) = \text{ntrl}$:

$$(\text{norm}(R \oplus_i S) = \text{ntrl}) \Leftrightarrow (\text{norm}(R) = \text{norm}(S) = \text{ntrl})$$

$$\begin{aligned}
lhs: & \quad \text{norm}((R \oplus_i S) \parallel Q) \\
& = \quad \text{norm}(\text{ntrl} \parallel Q) \\
& \stackrel{2.12}{=} \quad \text{reconv}(\text{sort}(\text{coll}(Q))) \\
rhs: & \quad \text{norm}(Q \parallel (R \oplus_i S)) \\
& = \quad \text{norm}(Q \parallel \text{ntrl}) \\
& \stackrel{2.12}{=} \quad \text{reconv}(\text{sort}(\text{coll}(Q)))
\end{aligned}$$

Case ii, $\text{norm}(R \oplus_i S) \neq \text{ntrl}$:

$$\begin{aligned}
lhs: & \quad \text{norm}((R \oplus_i S) \parallel Q) \\
& \stackrel{2.12}{=} \quad \text{reconv}(\text{sort}(\text{coll}(R \oplus_i S) \cup \text{coll}(Q))) \\
rhs: & \quad \text{norm}(Q \parallel (R \oplus_i S)) \\
& \stackrel{2.12}{=} \quad \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R \oplus_i S)))
\end{aligned}$$

$lhs = rhs$, since both sets $\text{coll}(R \oplus_i S) \cup \text{coll}(Q)$ and $\text{coll}(Q) \cup \text{coll}(R \oplus_i S)$ contain the same elements. Thus, sorting and reconstructing both sets, i.e. the application of $\text{reconv}(\text{sort}(\dots))$ on both sets, yields the same term.

□

Proof. We show the identity

$$\text{norm}(P \parallel (Q \parallel R)) = \text{norm}((P \parallel Q) \parallel R)$$

of Theorem 2.2 by structural induction on the term structure of P .

Base cases:

- $P = \text{ntrl}$:

$$\begin{aligned}
rhs: & \quad \text{norm}((\text{ntrl} \parallel Q) \parallel R) \\
& \stackrel{2.12}{=} \quad \text{reconv}(\text{sort}(\text{coll}(Q \parallel R))) \\
& \stackrel{\text{Lemma B.1}}{=} \quad \text{norm}(Q \parallel R)
\end{aligned}$$

$$lhs: \text{norm}(\text{ntrl} \parallel (Q \parallel R))$$

Case differentiation according to Law 2.12.

Case 1, $(\text{norm}(R) = \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl})$:

From $\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl}$ we conclude that $\text{norm}(Q) = \text{ntrl}$.

$$lhs: \text{norm}(\text{ntrl} \parallel (Q \parallel R)) = \text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$$

$$rhs: \text{norm}(Q \parallel R) = \text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$$

B. Uniqueness of the Normal Form: Proofs

Case 2, $(\text{norm}(R) = \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl})$:

From $\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl}$ we conclude that $\text{norm}(Q) \neq \text{ntrl}$.

$$\begin{aligned}
 lhs: & \quad \text{norm}((\text{ntrl} \parallel Q) \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl} \parallel Q))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q))) \\
 rhs: & \quad \text{norm}(Q \parallel \text{ntrl}) \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))
 \end{aligned}$$

Case 3, $(\text{norm}(R) \neq \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl})$:

From $\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl}$ we conclude that $\text{norm}(Q) = \text{ntrl}$.

$$\begin{aligned}
 lhs: & \quad \text{norm}((\text{ntrl} \parallel \text{ntrl}) \parallel R) = \text{reconv}(\text{sort}(\text{coll}(R))) \\
 rhs: & \quad \text{norm}(\text{ntrl} \parallel R) = \text{reconv}(\text{sort}(\text{coll}(R)))
 \end{aligned}$$

Case 4, $(\text{norm}(R) \neq \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl})$:

From $\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl}$ we conclude that $\text{norm}(Q) \neq \text{ntrl}$.

$$\begin{aligned}
 lhs: & \quad \text{norm}((\text{ntrl} \parallel Q) \parallel R) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl} \parallel Q) \cup \text{coll}(R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R))) \\
 rhs: & \quad \text{norm}(Q \parallel R) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \parallel R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(Q) \cup \text{coll}(R)))
 \end{aligned}$$

- $P = \text{asset}(a)$:

$$rhs: \text{norm}(\text{asset}(a) \parallel (Q \parallel R))$$

Case differentiation according to Law 2.12.

Case 1, $\text{norm}(Q \parallel R) = \text{ntrl}$:

The assumption implies that $\text{norm}(Q) = \text{norm}(R) = \text{ntrl}$.

$$\begin{aligned}
 rhs: & \quad \text{norm}(\text{asset}(a) \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)))) \\
 & \stackrel{\text{Lemma B.1}}{=} \text{norm}(\text{asset}(a)) \\
 & \stackrel{2.11}{=} \text{asset}(a) \\
 lhs: & \quad \text{norm}((\text{asset}(a) \parallel Q) \parallel R) \\
 & = \text{norm}(\text{asset}(a) \parallel \text{ntrl} \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel \text{ntrl}))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)))) \\
 & \stackrel{\text{Lemma B.1}}{=} \text{norm}(\text{asset}(a)) \\
 & \stackrel{2.11}{=} \text{asset}(a)
 \end{aligned}$$

Case 2, $\text{norm}(Q \parallel R) \neq \text{ntrl}$:

$$\begin{aligned}
 \text{rhs: } & \text{norm}(\text{asset}(a) \parallel (Q \parallel R)) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel (Q \parallel R)))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \parallel R)))
 \end{aligned}$$

$$\text{lhs: } \text{norm}((\text{asset}(a) \parallel Q) \parallel R)$$

Case differentiation according to Law 2.12, where only the two cases where $\text{norm}(\text{asset}(a) \parallel Q) \neq \text{ntrl}$ have to be considered.

Case i, $\text{norm}(R) = \text{ntrl}$:

This implies that $\text{norm}(Q) \neq \text{ntrl}$ since according to the assumption the property $\text{norm}(Q \parallel R) \neq \text{ntrl}$ holds.

$$\begin{aligned}
 \text{lhs: } & \text{norm}((\text{asset}(a) \parallel Q) \parallel \text{ntrl}) \\
 & = \text{norm}(\text{asset}(a) \parallel \text{ntrl} \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel Q))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q))) \\
 \text{rhs: } & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \parallel \text{ntrl}))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q)))
 \end{aligned}$$

Case ii, $(\text{norm}(R) \neq \text{ntrl}) \wedge (\text{norm}(\text{asset}(a) \parallel Q) \neq \text{ntrl})$:

$$\begin{aligned}
 \text{lhs: } & \text{norm}((\text{asset}(a) \parallel Q) \parallel R) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((\text{asset}(a) \parallel Q) \parallel R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel Q) \cup \text{coll}(R)))
 \end{aligned}$$

Case differentiation according to Law 2.17.

Case i, $Q = \text{ntrl}$:

$$\begin{aligned}
 \text{lhs: } & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel \text{ntrl}) \cup \\
 & \qquad \qquad \qquad \text{coll}(R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(R))) \\
 \text{rhs: } & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \\
 & \qquad \qquad \qquad \text{coll}(\text{ntrl} \parallel R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(R)))
 \end{aligned}$$

Case ii, $Q \neq \text{ntrl}$:

$$\begin{aligned}
 \text{lhs: } & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel Q) \cup \\
 & \qquad \qquad \qquad \text{coll}(R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \\
 & \qquad \qquad \qquad \text{coll}(Q) \cup \text{coll}(R))) \\
 \text{rhs: } & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \\
 & \qquad \qquad \qquad \text{coll}(Q \parallel R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \\
 & \qquad \qquad \qquad \text{coll}(Q) \cup \text{coll}(R)))
 \end{aligned}$$

B. Uniqueness of the Normal Form: Proofs

Inductive step:

- $P = S \parallel T$: Showing: $\text{norm}((S \parallel T) \parallel (Q \parallel R)) = \text{norm}(((S \parallel T) \parallel Q) \parallel R)$
 I. H. : $\text{norm}(S \parallel (Q \parallel R)) = \text{norm}((S \parallel Q) \parallel R)$ and
 $\text{norm}(T \parallel (Q \parallel R)) = \text{norm}((T \parallel Q) \parallel R)$

lhs: $\text{norm}((S \parallel T) \parallel (Q \parallel R))$

Case differentiation according to Law 2.12.

Case 1, $(\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q \parallel R) = \text{ntrl})$:

This assumption implies that $\text{norm}(Q) = \text{norm}(R) = \text{ntrl}$.

lhs: $\text{norm}(\text{ntrl} \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$

rhs: $\text{norm}((\text{ntrl} \parallel \text{ntrl}) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl}$

Case 2, $(\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q \parallel R) \neq \text{ntrl})$:

This assumption implies that $\text{norm}(S) = \text{norm}(T) = \text{ntrl}$.

lhs: $\text{norm}(\text{ntrl} \parallel (Q \parallel R)) \stackrel{2.12}{=} \text{norm}(Q \parallel R)$

rhs: $\text{norm}((\text{ntrl} \parallel Q) \parallel R)$

Case differentiation according to Law 2.12.

Case i, $(\text{norm}(R) = \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl})$:

The proof of the base case $P = \text{ntrl}$, Case 1, applies analogously to this case.

Case ii, $(\text{norm}(R) = \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl})$:

The proof of the base case $P = \text{ntrl}$, Case 2, applies analogously to this case.

Case iii, $(\text{norm}(R) \neq \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) = \text{ntrl})$:

The proof of the base case $P = \text{ntrl}$, Case 3, applies analogously to this case.

Case iv, $(\text{norm}(R) \neq \text{ntrl}) \wedge (\text{norm}(\text{ntrl} \parallel Q) \neq \text{ntrl})$:

The proof of the base case $P = \text{ntrl}$, Case 4, applies analogously to this case.

Case 3, $(\text{norm}(S \parallel T) \neq \text{ntrl}) \wedge (\text{norm}(Q \parallel R) = \text{ntrl})$:

The proof for the previous Case 2 applies analogously for this case, where only $(S \parallel T)$ and $(Q \parallel R)$ swap their roles.

Case 4, $(\text{norm}(S \parallel T) \neq \text{ntrl}) \wedge (\text{norm}(Q \parallel R) \neq \text{ntrl})$:

This assumption implies that $\neg((\text{norm}(S) = \text{ntrl}) \wedge (\text{norm}(T) = \text{ntrl}))$
 and $\neg((\text{norm}(Q) = \text{ntrl}) \wedge (\text{norm}(R) = \text{ntrl}))$.

$$\begin{aligned}
lhs: & \quad \text{norm}((S \parallel T) \parallel (Q \parallel R)) \\
& \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \parallel T) \parallel (Q \parallel R)))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(Q \parallel R)))
\end{aligned}$$

Case differentiation according to Law 2.17 (Note that the case $(Q = \text{ntrl}) \wedge (R = \text{ntrl})$ is not possible due to the assumption).

Case i, $(Q = \text{ntrl}) \wedge (R \neq \text{ntrl})$:

$$\begin{aligned}
lhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(\text{ntrl} \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(R))) \\
rhs: & \quad \text{norm}(((S \parallel T) \parallel \text{ntrl}) \parallel R) \\
& \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(((S \parallel T) \parallel \text{ntrl}) \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}((S \parallel T) \parallel \text{ntrl}) \cup \text{coll}(R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(R)))
\end{aligned}$$

Case ii, $(Q \neq \text{ntrl}) \wedge (R = \text{ntrl})$:

The proof for the previous Case i applies analogously for this case, where only Q and R swap their roles.

Case iii, $(Q \neq \text{ntrl}) \wedge (R \neq \text{ntrl})$:

$$\begin{aligned}
lhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(Q \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(Q) \cup \text{coll}(R))) \\
rhs: & \quad \text{norm}(((S \parallel T) \parallel Q) \parallel R) \\
& \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(((S \parallel T) \parallel Q) \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}((S \parallel T) \parallel Q) \cup \text{coll}(R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T) \cup \text{coll}(Q) \cup \text{coll}(R)))
\end{aligned}$$

- $P = S \oplus_i T$: Showing: $\text{norm}((S \oplus_i T) \parallel (Q \parallel R)) = \text{norm}(((S \oplus_i T) \parallel Q) \parallel R)$
Ind. hyp. : $\text{norm}(S \parallel (Q \parallel R)) = \text{norm}((S \parallel Q) \parallel R)$ and $\text{norm}(T \parallel (Q \parallel R)) = \text{norm}((T \parallel Q) \parallel R)$

$$lhs: \text{norm}((S \oplus_i T) \parallel (Q \parallel R))$$

Case differentiation according to Law 2.12.

Case 1, $(\text{norm}(S \oplus_i T) = \text{ntrl}) \wedge (\text{norm}(Q \parallel R) = \text{ntrl})$:

$lhs = rhs$ follows directly from Law 2.12.

Case 2, $(\text{norm}(S \oplus_i T) = \text{ntrl}) \wedge (\text{norm}(Q \parallel R) \neq \text{ntrl})$:

$lhs = rhs$, where the proof proceeds analogously to the one for Case 2 of the previous constructor case $P = S \parallel T$.

B. Uniqueness of the Normal Form: Proofs

Case 3, $(\text{norm}(S \oplus_i T) \neq \text{ntrl}) \wedge (\text{norm}(Q \parallel R) = \text{ntrl})$:

The assumption for this case implies that $\text{norm}(S) \neq \text{ntrl}$, $\text{norm}(T) \neq \text{ntrl}$, and $\text{norm}(Q) = \text{norm}(R) = \text{ntrl}$.

$$\begin{aligned}
 lhs: & \quad \text{norm}((S \oplus_i T) \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T))) \\
 rhs: & \quad \text{norm}(((S \oplus_i T) \parallel \text{ntrl}) \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel \text{ntrl}))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T)))
 \end{aligned}$$

Case 4, $(\text{norm}(S \oplus_i T) \neq \text{ntrl}) \wedge (\text{norm}(Q \parallel R) \neq \text{ntrl})$:

The assumption implies that $\neg((\text{norm}(S) = \text{ntrl}) \wedge (\text{norm}(T) = \text{ntrl}))$ and $\neg((\text{norm}(Q) = \text{ntrl}) \wedge (\text{norm}(R) = \text{ntrl}))$.

$$\begin{aligned}
 lhs: & \quad \text{norm}((S \oplus_i T) \parallel (Q \parallel R)) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel (Q \parallel R)))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(Q \parallel R)))
 \end{aligned}$$

$$rhs: \text{norm}(((S \oplus_i T) \parallel Q) \parallel R)$$

Case differentiation according to Law 2.12, where due to the assumption only the two cases where $\text{norm}((S \oplus_i T) \parallel Q) \neq \text{ntrl}$ have to be considered (since the part $\text{norm}(S \oplus_i T) \neq \text{ntrl}$ according to the assumption).

Case i, $\text{norm}(R) = \text{ntrl}$:

This implies that $\text{norm}(Q) \neq \text{ntrl}$ since according to the assumption the property $\text{norm}(Q \parallel R) \neq \text{ntrl}$ holds.

$$\begin{aligned}
 rhs: & \quad \text{norm}(((S \oplus_i T) \parallel Q) \parallel \text{ntrl}) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel Q))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(Q))) \\
 lhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(Q \parallel \text{ntrl}))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(Q)))
 \end{aligned}$$

Case ii, $\text{norm}(R) \neq \text{ntrl}$:

$$\begin{aligned}
 rhs: & \quad \text{norm}(((S \oplus_i T) \parallel Q) \parallel R) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(((S \oplus_i T) \parallel Q) \parallel R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel Q) \cup \text{coll}(R)))
 \end{aligned}$$

Case differentiation according to Law 2.17.

Case a, $(Q = \text{ntrl}) \wedge (\text{norm}(S \oplus_i T) \neq \text{ntrl})$:

$$\begin{aligned}
 rhs: & \quad \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel \text{ntrl}) \cup \\
 & \quad \quad \quad \text{coll}(R))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(R)))
 \end{aligned}$$

$$\begin{aligned}
lhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \\
& \quad \quad \quad \text{coll}(\text{ntrl} \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \text{coll}(R)))
\end{aligned}$$

Case b, $(Q \neq \text{ntrl}) \wedge (\text{norm}(S \oplus_i T) \neq \text{ntrl})$:

$$\begin{aligned}
rhs: & \quad \text{reconv}(\text{sort}(\text{coll}((S \oplus_i T) \parallel Q) \cup \\
& \quad \quad \quad \text{coll}(R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \\
& \quad \quad \quad \text{coll}(Q) \cup \text{coll}(R)))
\end{aligned}$$

$$\begin{aligned}
lhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \\
& \quad \quad \quad \text{coll}(Q \parallel R))) \\
& \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_i T) \cup \\
& \quad \quad \quad \text{coll}(Q) \cup \text{coll}(R)))
\end{aligned}$$

□

Proof. We show the identity

$$\text{norm}((P \parallel Q) \oplus_i (P \parallel R)) = \text{norm}(P \parallel (Q \oplus_i R))$$

of Theorem 2.2 by structural induction on the term structure of P .

Base cases:

- $P = \text{ntrl}$:

$$lhs: \text{norm}((\text{ntrl} \parallel Q) \oplus_i (\text{ntrl} \parallel R))$$

$$rhs: \text{norm}(\text{ntrl} \parallel (Q \oplus_i R))$$

Case differentiation according to the three cases of Law 2.13.

Case 1, $\text{norm}(\text{ntrl} \parallel Q) = \text{norm}(\text{ntrl} \parallel R)$:

This implies that $\text{norm}(Q) = \text{norm}(R)$

$$lhs: \text{norm}((\text{ntrl} \parallel Q) \oplus_i (\text{ntrl} \parallel R)) \stackrel{2.13}{=} \text{norm}(\text{ntrl} \parallel Q)$$

Case differentiation according to the Law 2.12.

Case i, $Q = \text{ntrl}$:

This implies for this case that $\text{norm}(Q) = \text{norm}(R) = \text{ntrl}$.

$$lhs: \text{norm}(\text{ntrl} \parallel \text{ntrl})$$

$$\stackrel{2.12}{=} \text{ntrl}$$

$$rhs: \text{norm}(\text{ntrl} \parallel (\text{ntrl} \oplus_i \text{ntrl}))$$

$$\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl} \oplus_i \text{ntrl})))$$

$$\stackrel{2.13}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl})))$$

$$\stackrel{\text{Lemma B.1}}{=} \text{norm}(\text{ntrl})$$

$$\stackrel{2.10}{=} \text{ntrl}$$

B. Uniqueness of the Normal Form: Proofs

Case ii, $Q \neq \text{ntrl}$:

This implies for this case that $\text{norm}(R) \neq \text{ntrl}$.

$$\begin{array}{l}
 \text{lhs:} \quad \text{norm}(\text{ntrl} \parallel Q) \\
 \quad \quad \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q))) \\
 \text{rhs:} \quad \quad \quad \text{norm}(\text{ntrl} \parallel (Q \oplus_i R)) \\
 \quad \quad \quad \quad \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R))) \\
 \quad \quad \quad \quad \quad \stackrel{\text{Lemma B.1}}{=} \text{norm}(Q \oplus_i R) \\
 \quad \quad \quad \quad \quad \stackrel{2.13,}{=} \text{norm}(Q) = \\
 \quad \quad \quad \quad \quad \stackrel{\text{norm}(R)}{=} \text{norm}(Q) \\
 \quad \quad \quad \quad \quad \stackrel{\text{Lemma B.1}}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))
 \end{array}$$

Case 2, $\left(\text{norm}(\text{ntrl} \parallel Q) \neq \text{norm}(\text{ntrl} \parallel R) \right) \wedge$
 $\left(\text{cmn}(\text{coll}(\text{ntrl} \parallel Q), \text{coll}(\text{ntrl} \parallel R)) = \emptyset \right)$:

Since $\text{norm}(\text{ntrl} \parallel Q) \neq \text{norm}(\text{ntrl} \parallel R)$ this means that $\text{norm}(Q) \neq \text{norm}(R)$.

$$\begin{array}{l}
 \text{lhs:} \quad \quad \quad \text{norm}\left((\text{ntrl} \parallel Q) \oplus_i (\text{ntrl} \parallel R)\right) \\
 \quad \quad \quad \quad \quad \stackrel{2.13}{=} \text{norm}(\text{ntrl} \parallel Q) \oplus_i \text{norm}(\text{ntrl} \parallel R) \\
 \quad \quad \quad \quad \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q))) \oplus_i \text{reconv}(\text{sort}(\text{coll}(R))) \\
 \quad \quad \quad \quad \quad \stackrel{\text{Lemma B.1}}{=} \text{norm}(Q) \oplus_i \text{norm}(R) \\
 \text{rhs:} \quad \quad \quad \text{norm}(\text{ntrl} \parallel (Q \oplus_i R)) \\
 \quad \quad \quad \quad \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R))) \\
 \quad \quad \quad \quad \quad \stackrel{\text{Lemma B.1}}{=} \text{norm}(Q \oplus_i R) \\
 \quad \quad \quad \quad \quad \stackrel{2.13}{=} \text{norm}(Q) \oplus_i \text{norm}(R)
 \end{array}$$

Case 3, $\left(\text{norm}(\text{ntrl} \parallel Q) \neq \text{norm}(\text{ntrl} \parallel R) \right) \wedge$
 $\left(\text{cmn}(\text{coll}(\text{ntrl} \parallel Q), \text{coll}(\text{ntrl} \parallel R)) \neq \emptyset \right)$:

We can assume that $\text{norm}(Q) \neq \text{norm}(R)$ since $\text{norm}(\text{ntrl} \parallel Q) \neq \text{norm}(\text{ntrl} \parallel R)$, and $\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset$ since $\text{cmn}(\text{coll}(\text{ntrl} \parallel Q), \text{coll}(\text{ntrl} \parallel R)) \neq \emptyset$.

$$\begin{aligned}
lhs: & \quad \text{norm}\left(\left(\text{ntrl} \parallel Q\right) \oplus_i \left(\text{ntrl} \parallel R\right)\right) \\
& \stackrel{2.13}{=} \text{reconv}\left(\text{sort}\left(X \cup Y\right)\right) \\
& \text{where } X \text{ and } Y \text{ are defined according to Law 2.13 as} \\
& \quad X = \text{cmn}\left(\text{coll}\left(\text{ntrl} \parallel Q\right), \text{coll}\left(\text{ntrl} \parallel R\right)\right) \\
& \quad \stackrel{2.17}{=} \text{cmn}\left(\text{coll}\left(Q\right), \text{coll}\left(R\right)\right) \\
& \text{and} \\
& \quad Y = \left\{ \begin{array}{l} \text{reconv}\left(\text{sort}\left(\text{diff}\left(\text{coll}\left(\text{ntrl} \parallel Q\right), \right.\right.\right. \\ \left.\left.\left.\text{coll}\left(\text{ntrl} \parallel R\right)\right)\right)\right) \\ \oplus_i \\ \text{reconv}\left(\text{sort}\left(\text{diff}\left(\text{coll}\left(\text{ntrl} \parallel R\right), \right.\right.\right. \\ \left.\left.\left.\text{coll}\left(\text{ntrl} \parallel Q\right)\right)\right)\right) \end{array} \right\} \\
& \stackrel{2.17}{=} \left\{ \begin{array}{l} \text{reconv}\left(\text{sort}\left(\text{diff}\left(\text{coll}\left(Q\right), \text{coll}\left(R\right)\right)\right)\right) \\ \oplus_i \\ \text{reconv}\left(\text{sort}\left(\text{diff}\left(\text{coll}\left(R\right), \text{coll}\left(Q\right)\right)\right)\right) \end{array} \right\} \\
rhs: & \quad \text{norm}\left(\text{ntrl} \parallel \left(Q \oplus_i R\right)\right) \\
& \stackrel{2.12}{=} \text{reconv}\left(\text{sort}\left(\text{coll}\left(Q \oplus_i R\right)\right)\right) \\
& \stackrel{\text{Lemma B.1}}{=} \text{norm}\left(Q \oplus_i R\right) \\
& \stackrel{2.13}{=} \text{reconv}\left(\text{sort}\left(X \cup Y\right)\right) \\
& \text{where } X \text{ and } Y \text{ are exactly as specified above for the } lhs.
\end{aligned}$$

- $P = \text{asset}(a)$:

$$lhs: \text{norm}\left(\left(\text{asset}(a) \parallel Q\right) \oplus_i \left(\text{asset}(a) \parallel R\right)\right)$$

$$rhs: \text{norm}\left(\text{asset}(a) \parallel \left(Q \oplus_i R\right)\right)$$

Case differentiation according to the cases of Law 2.13, where due to the common element $\text{asset}(a)$ only the first and the third case apply.

Case 1, $\text{norm}(\text{asset}(a) \parallel Q) = \text{norm}(\text{asset}(a) \parallel R)$:

This implies that $\text{norm}(Q) = \text{norm}(R)$.

$$lhs: \text{norm}\left(\left(\text{asset}(a) \parallel Q\right) \oplus_i \left(\text{asset}(a) \parallel R\right)\right)$$

$$\stackrel{2.13}{=} \text{norm}\left(\text{asset}(a) \parallel Q\right)$$

Case differentiation according to the cases of Law 2.12.

Case i, $Q = \text{ntrl}$:

$$lhs: \text{norm}\left(\text{asset}(a) \parallel \text{ntrl}\right)$$

$$\stackrel{2.12}{=} \text{reconv}\left(\text{sort}\left(\text{coll}\left(\text{asset}(a)\right)\right)\right)$$

$$\stackrel{\text{Lemma B.1}}{=} \text{norm}\left(\text{asset}(a)\right)$$

$$rhs: \text{In this case we have } \text{norm}(Q) = \text{norm}(R) = \text{ntrl}.$$

B. Uniqueness of the Normal Form: Proofs

$$\begin{aligned}
& \text{norm}(\text{asset}(a) \parallel (\text{ntrl} \oplus_i \text{ntrl})) \\
\stackrel{2.12}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)))) \\
& \text{since } \text{norm}(\text{ntrl} \oplus_i \text{ntrl}) = \text{ntrl}. \\
\stackrel{\text{Lemma B.1}}{=} & \text{norm}(\text{asset}(a))
\end{aligned}$$

Case ii, $Q \neq \text{ntrl}$:

$$\begin{aligned}
\text{lhs: } & \text{norm}(\text{asset}(a) \parallel Q) \\
\stackrel{2.12}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel Q))) \\
\stackrel{2.17}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q))) \\
\text{rhs: } & \text{norm}(\text{asset}(a) \parallel (Q \oplus_i R)) \\
\stackrel{2.12}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a) \parallel (Q \oplus_i R)))) \\
\stackrel{2.17}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R))) \\
& \text{norm}(Q) = \\
& \text{norm}(R), \\
\stackrel{2.18}{=} & \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q)))
\end{aligned}$$

Case 2, $(\text{norm}(\text{asset}(a) \parallel Q) \neq \text{norm}(\text{asset}(a) \parallel R)) \wedge$
 $(\text{cmn}(\text{coll}(\text{asset}(a) \parallel Q), \text{coll}(\text{asset}(a) \parallel R)) \neq \emptyset)$:

This case implies that $\text{norm}(Q) \neq \text{norm}(R)$.

$$\begin{aligned}
\text{lhs: } & \text{norm}((\text{asset}(a) \parallel Q) \oplus_i (\text{asset}(a) \parallel R)) \\
\stackrel{2.13}{=} & \text{reconv}(\text{sort}(X \cup Y))
\end{aligned}$$

where X and Y are defined according to Law 2.13 as

$$\begin{aligned}
X & = \text{cmn}(\text{coll}(\text{asset}(a) \parallel Q), \text{coll}(\text{asset}(a) \parallel R)) \\
\stackrel{\text{Lemma B.3}}{=} & \{\text{asset}(a)\} \cup \text{cmn}(\text{coll}(Q), \text{coll}(R))
\end{aligned}$$

and

$$\begin{aligned}
Y & = \left\{ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(\text{asset}(a) \parallel Q), \right. \right. \right. \right. \\
& \qquad \qquad \qquad \left. \left. \left. \text{coll}(\text{asset}(a) \parallel R) \right) \right) \right) \\
& \oplus_i \\
& \left. \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(\text{asset}(a) \parallel R), \right. \right. \right. \right. \right. \\
& \qquad \qquad \qquad \left. \left. \left. \text{coll}(\text{asset}(a) \parallel Q) \right) \right) \right) \right\}
\end{aligned}$$

$$\begin{aligned}
\stackrel{\text{Lem. B.5}}{=} & \left\{ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(Q), \text{coll}(R) \right) \right) \right) \right. \\
& \oplus_i \\
& \left. \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(R), \text{coll}(Q) \right) \right) \right) \right\}
\end{aligned}$$

$$\begin{aligned}
\stackrel{\text{Lem. B.6}}{=} & \left\{ \text{reconv} \left(\text{sort}(\text{coll}(Q)) \right) \right. \\
& \oplus_i \\
& \left. \text{reconv} \left(\text{sort}(\text{coll}(R)) \right) \right\}
\end{aligned}$$

rhs: $\text{norm}(\text{asset}(a) \parallel (Q \oplus_i R))$
 $\stackrel{2.12}{=}$ $\text{reconv}(\text{sort}(\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R)))$
as $\text{norm}(Q \oplus_i R) \neq \text{ntrl}$, since due to the assumption of this case Q and R have common parts which are different to ntrl , themselves.

$lhs = rhs$, if the corresponding sets $X \cup Y$ and $\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R)$ contain the same elements.

Case differentiation for the set $\text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R)$ in the *rhs* according to the cases of Law 2.18.

Case i, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$:

Consider the set which is part of the *rhs*:

$$\begin{aligned} & \text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R) \\ & \stackrel{2.18}{=} \text{coll}(\text{asset}(a)) \cup \{\text{norm}(Q) \oplus_i \text{norm}(R)\} \\ & \stackrel{2.16}{=} \{\text{asset}(a)\} \cup \{\text{norm}(Q) \oplus_i \text{norm}(R)\} \\ & \stackrel{\text{Lemma B.1}}{=} \{\text{asset}(a)\} \cup \left\{ \text{reconv}(\text{sort}(\text{coll}(Q))) \oplus_i \right. \\ & \qquad \qquad \qquad \left. \text{reconv}(\text{sort}(\text{coll}(R))) \right\} \end{aligned}$$

This result is equivalent to the set which appears in the *lhs*, since the part $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$ due to the case restriction, which means that the set X of the *lhs* reduces to $\{\text{asset}(a)\}$. Together with the set Y we get the same set as here for the *rhs*.

Case ii, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset$:

Consider the set which is part of the *rhs*:

$$\begin{aligned} & \text{coll}(\text{asset}(a)) \cup \text{coll}(Q \oplus_i R) \\ & \stackrel{2.18}{=} \text{coll}(\text{asset}(a)) \cup \text{coll}(\text{norm}(Q \oplus_i R)) \\ & \stackrel{2.13}{=} \text{coll}(\text{asset}(a)) \cup \text{coll}(\text{reconv}(\text{sort}(X \cup Y))) \\ & \stackrel{2.16}{=} \{\text{asset}(a)\} \cup \text{coll}(\text{reconv}(\text{sort}(X \cup Y))) \end{aligned}$$

Thus $lhs = rhs$, since this is exactly the set which exists in the *lhs*.

Inductive step:

- $P = S \parallel T$: Showing: $\text{norm}\left(\left((S \parallel T) \parallel Q\right) \oplus_i \left((S \parallel T) \parallel R\right)\right) =$
 $\text{norm}\left((S \parallel T) \parallel (Q \oplus_i R)\right)$
Ind. hyp. : $\text{norm}\left((S \parallel Q) \oplus_i (S \parallel R)\right) =$
 $\text{norm}\left(S \parallel (Q \oplus_i R)\right)$ and
 $\text{norm}\left((T \parallel Q) \oplus_i (T \parallel R)\right) =$
 $\text{norm}\left(T \parallel (Q \oplus_i R)\right)$

B. Uniqueness of the Normal Form: Proofs

$$lhs: \text{norm}\left(\left((S \parallel T) \parallel Q\right) \oplus_i \left((S \parallel T) \parallel R\right)\right)$$

Case differentiation according to the three cases of Law 2.13.

Case 1, $\text{norm}((S \parallel T) \parallel Q) = \text{norm}((S \parallel T) \parallel R)$:

The assumption for this case implies that $\text{norm}(Q) = \text{norm}(R)$.

$$lhs: \text{norm}\left(\left((S \parallel T) \parallel Q\right) \oplus_i \left((S \parallel T) \parallel R\right)\right) \\ \stackrel{2.13}{=} \text{norm}((S \parallel T) \parallel Q)$$

Case differentiation according to Law 2.12:

Case i, $\text{norm}(Q) = \text{ntrl}$:

Due to the assumption $\text{norm}(Q) = \text{norm}(R)$ this implies that $\text{norm}(R) = \text{ntrl}$.

$$lhs: \text{norm}((S \parallel T) \parallel Q) \\ \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T))) \\ rhs: \text{norm}((S \parallel T) \parallel (\text{ntrl} \oplus_i \text{ntrl})) \\ \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(S \parallel T)))$$

Case ii, $\text{norm}(S \parallel T) = \text{ntrl}$:

$$lhs: \text{norm}((S \parallel T) \parallel Q) \\ \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q))) \\ rhs: \text{norm}((S \parallel T) \parallel (Q \oplus_i R)) \\ \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R))) \\ \stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(Q)))$$

Case iii, $(\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q) = \text{ntrl})$:

Due to the assumption $\text{norm}(Q) = \text{norm}(R)$ this implies that $\text{norm}(R) = \text{ntrl}$.

$$lhs: \text{norm}((\text{ntrl} \parallel \text{ntrl}) \parallel \text{ntrl}) \stackrel{2.12}{=} \text{ntrl} \\ rhs: \text{norm}(\text{ntrl} \parallel (\text{ntrl} \oplus_i \text{ntrl})) \stackrel{2.12}{=} \text{ntrl}$$

Case 2, $\left(\text{norm}((S \parallel T) \parallel Q) \neq \text{norm}((S \parallel T) \parallel R)\right) \wedge$
 $\left(\text{cmn}(\text{coll}((S \parallel T) \parallel Q), \text{coll}((S \parallel T) \parallel R)) = \emptyset\right)$:

The assumption of this case implies that $\text{norm}(S \parallel T) = \text{ntrl}$ and $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$, since only then can the two elements $((S \parallel T) \parallel Q)$ and $((S \parallel T) \parallel R)$ have no common elements.

$$rhs: \text{norm}((S \parallel T) \parallel (Q \oplus_i R))$$

Case differentiation according to Law 2.12. But due to the assumption

only the case $(\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) \neq \text{ntrl})$ applies, since $\text{norm}(Q \oplus_i R) = \text{ntrl}$ would only be possible if both $\text{norm}(Q)$ and $\text{norm}(R)$ equal ntrl , which is contrary to the higher-level assumption for this case.

$$\begin{aligned}
& \text{Case i, } (\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) \neq \text{ntrl}): \\
& \text{rhs: } \quad \text{norm}(\text{ntrl} \parallel (Q \oplus_i R)) \\
& \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R))) \\
& \quad \stackrel{2.18}{=} \text{reconv}(\text{sort}(\{ \text{norm}(Q) \oplus_i \text{norm}(R) \})) \\
& \quad = \text{norm}(Q) \oplus_i \text{norm}(R) \\
& \quad \text{since the set contains only one element.} \\
& \text{lhs: } \quad \text{norm}(\text{ntrl} \parallel Q) \oplus_i \text{norm}(\text{ntrl} \parallel R) \\
& \quad \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q))) \oplus_i \\
& \quad \text{reconv}(\text{sort}(\text{coll}(R))) \\
& \quad \stackrel{\text{Lemma B.1}}{=} \text{norm}(Q) \oplus_i \text{norm}(R)
\end{aligned}$$

$$\begin{aligned}
& \text{Case 3, } (\text{norm}((S \parallel T) \parallel Q) \neq \text{norm}((S \parallel T) \parallel R)) \wedge \\
& (\text{cmn}(\text{coll}((S \parallel T) \parallel Q), \text{coll}((S \parallel T) \parallel R)) \neq \emptyset):
\end{aligned}$$

The assumption for this case implies that $\text{norm}(Q) \neq \text{norm}(R)$.

$$\begin{aligned}
& \text{lhs: } \quad \text{norm}(((S \parallel T) \parallel Q) \oplus_i ((S \parallel T) \parallel R)) \\
& \quad \stackrel{2.13}{=} \text{reconv}(\text{sort}(X \cup Y)) \\
& \quad \text{where } X \text{ and } Y \text{ are defined according to Law 2.13 as} \\
& \quad X = \text{cmn}(\text{coll}((S \parallel T) \parallel Q), \text{coll}((S \parallel T) \parallel R)) \\
& \quad \stackrel{\text{B.3}}{=} \text{coll}(S \parallel T) \cup \text{cmn}(\text{coll}(Q), \text{coll}(R)) \\
& \quad \text{and} \\
& \quad Y = \left\{ \begin{aligned} & \text{reconv}(\text{sort}(\text{diff}(\text{coll}((S \parallel T) \parallel Q), \\ & \quad \text{coll}((S \parallel T) \parallel R)))) \\ & \oplus_i \\ & \text{reconv}(\text{sort}(\text{diff}(\text{coll}((S \parallel T) \parallel R), \\ & \quad \text{coll}((S \parallel T) \parallel Q)))) \end{aligned} \right\} \\
& \quad \stackrel{\text{Lem. B.5}}{=} \left\{ \begin{aligned} & \text{reconv}(\text{sort}(\text{diff}(\text{coll}(Q), \text{coll}(R)))) \\ & \oplus_i \\ & \text{reconv}(\text{sort}(\text{diff}(\text{coll}(R), \text{coll}(Q)))) \end{aligned} \right\}
\end{aligned}$$

B. Uniqueness of the Normal Form: Proofs

$$rhs: \text{norm}\left((S \parallel T) \parallel (Q \oplus_i R)\right)$$

Case differentiation according to Law 2.12.

Due to the assumption $\text{norm}(Q) \neq \text{norm}(R)$ the situation where $\text{norm}(Q) = \text{ntrl} = \text{norm}(R)$ cannot exist, which implies that only those cases of Law 2.12 have to be considered where $\text{norm}(Q \oplus_i R) \neq \text{ntrl}$.

Case i, $(\text{norm}(S \parallel T) = \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) \neq \text{ntrl})$:

$$rhs: \text{norm}\left(\text{ntrl} \parallel (Q \oplus_i R)\right)$$

$$\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R)))$$

Case differentiation according to Law 2.13.

Case a, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$:

$$rhs: \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R)))$$

$$\stackrel{Lem. = B.1}{=} \text{norm}(Q \oplus_i R)$$

$$\stackrel{2.13}{=} \text{norm}(Q) \oplus_i \text{norm}(R)$$

$lhs = rhs$, since $X = \text{coll}(\text{ntrl}) \cup \text{ntrl} = \text{ntrl}$ according to the assumptions for this case. Thus the set $X \cup Y$ reduces to Y which leaves the lhs as:

$$lhs: \text{reconv}(\text{sort}(Y))$$

one
elem.

$$\stackrel{list =}{=} \text{reconv}\left(\text{sort}\left(\text{diff}(\text{coll}(Q), \text{coll}(R))\right)\right)$$

\oplus_i

$$\text{reconv}\left(\text{sort}\left(\text{diff}(\text{coll}(R), \text{coll}(Q))\right)\right)$$

$$\stackrel{B.6}{=} \text{reconv}\left(\text{sort}(\text{coll}(Q))\right)$$

\oplus_i

$$\text{reconv}\left(\text{sort}(\text{coll}(R))\right)$$

$$\stackrel{B.1}{=} \text{norm}(Q) \oplus_i \text{norm}(R)$$

Case b, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset$:

$$rhs: \text{reconv}(\text{sort}(\text{coll}(Q \oplus_i R)))$$

$$\stackrel{Lem. = B.1}{=} \text{norm}(Q \oplus_i R)$$

$$\stackrel{2.13}{=} \text{reconv}(\text{sort}(X' \cup Y'))$$

where X' and Y' are defined according to Law 2.13 as

$$X' = \text{cmn}(\text{coll}(Q), \text{coll}(R))$$

and

$$Y' = \left\{ \begin{array}{l} \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(Q), \text{coll}(R) \right) \right) \right) \\ \oplus_i \\ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(R), \text{coll}(Q) \right) \right) \right) \end{array} \right\}$$

$lhs = rhs$ since $Y = Y'$, and due to the assumption $(\text{norm}(S \parallel T) = \text{ntrl})$ we have $X = X'$.

Case ii, $(\text{norm}(S \parallel T) \neq \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) \neq \text{ntrl})$:

$$\begin{aligned} rhs: & \text{norm} \left((S \parallel T) \parallel (Q \oplus_i R) \right) \\ & \stackrel{2.12}{=} \text{reconv} \left(\text{sort} \left(\text{coll} \left((S \parallel T) \parallel (Q \oplus_i R) \right) \right) \right) \\ & \stackrel{2.17}{=} \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \text{coll} (Q \oplus_i R) \right) \right) \end{aligned}$$

Case differentiation according to Law 2.18.

Case a, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$:

$$\begin{aligned} rhs: & \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \text{coll} (Q \oplus_i R) \right) \right) \\ & \stackrel{2.18}{=} \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \right. \right. \\ & \quad \left. \left. \{ \text{norm}(Q) \oplus_i \text{norm}(R) \} \right) \right) \end{aligned}$$

$rhs = lhs$ since $\text{coll}(S \parallel T) \cup \{ \text{norm}(Q) \oplus_i \text{norm}(R) \}$ equals the set $X \cup Y$, as $Y = \{ \text{norm}(Q) \oplus_i \text{norm}(R) \}$, and due to the assumption for this case we have $X = \text{coll}(S \parallel T) \cup \emptyset = \text{coll}(S \parallel T)$.

Case b, $\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset$:

$$\begin{aligned} rhs: & \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \text{coll} (Q \oplus_i R) \right) \right) \\ & \stackrel{2.18}{=} \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \right. \right. \\ & \quad \left. \left. \text{coll}(\text{norm}(Q \oplus_i R)) \right) \right) \\ & \stackrel{2.13}{=} \text{reconv} \left(\text{sort} \left(\text{coll} (S \parallel T) \cup \right. \right. \\ & \quad \left. \left. \text{coll}(\text{reconv}(\text{sort}(\tilde{X} \cup \tilde{Y}))) \right) \right) \end{aligned}$$

where \tilde{X} and \tilde{Y} are defined according to Law 2.13 as

$$\begin{aligned} \tilde{X} &= \text{cmn}(\text{coll}(Q), \text{coll}(R)) \\ \text{and} \\ \tilde{Y} &= \left\{ \begin{array}{l} \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(Q), \text{coll}(R) \right) \right) \right) \\ \oplus_i \\ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(R), \text{coll}(Q) \right) \right) \right) \end{array} \right\} \end{aligned}$$

$rhs = lhs$, since

$\text{coll} (S \parallel T) \cup \text{coll}(\text{reconv}(\text{sort}(\tilde{X} \cup \tilde{Y}))) = X \cup Y$
as the subsequence $\text{coll}(\text{reconv}(\text{sort}(\tilde{X} \cup \tilde{Y})))$
contains the same elements as the subsequence
 $\text{cmn}(\text{coll}(Q), \text{coll}(R)) \cup Y$.

Case iii, $(\text{norm}(Q) \neq \text{norm}(R)) \wedge (\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset)$:

$rhs: \text{norm}(Q \oplus_i R) \stackrel{2.13}{=} \text{reconv}(\text{sort}(X \cup Y))$, where

$X = \text{cmn}(\text{coll}(Q), \text{coll}(R))$

$Y = \left\{ \begin{array}{l} \text{reconv}(\text{sort}(\text{diff}(\text{coll}(Q), \text{coll}(R)))) \oplus_i \\ \text{reconv}(\text{sort}(\text{diff}(\text{coll}(R), \text{coll}(Q)))) \end{array} \right\}$

$lhs: \text{norm}((\text{ntrl} \parallel Q) \oplus_i (\text{ntrl} \parallel R))$

$\stackrel{2.13}{=} \text{reconv}(\text{sort}(X' \cup Y'))$, where

$X' = \text{cmn}(\text{coll}(\text{ntrl} \parallel Q), \text{coll}(\text{ntrl} \parallel R))$

$\stackrel{2.17}{=} \text{cmn}(\text{coll}(Q), \text{coll}(R))$

$= X$

$Y' = \left\{ \begin{array}{l} \text{reconv}(\text{sort}(\text{diff}(\text{coll}(\text{ntrl} \parallel Q), \text{coll}(\text{ntrl} \parallel R)))) \oplus_i \\ \text{reconv}(\text{sort}(\text{diff}(\text{coll}(\text{ntrl} \parallel R), \text{coll}(\text{ntrl} \parallel Q)))) \end{array} \right\}$

$\stackrel{2.17}{=} \left\{ \begin{array}{l} \text{reconv}(\text{sort}(\text{diff}(\text{coll}(Q), \text{coll}(R)))) \oplus_i \\ \text{reconv}(\text{sort}(\text{diff}(\text{coll}(R), \text{coll}(Q)))) \end{array} \right\}$

$= Y$

$lhs = rhs$ since $X = X'$ and $Y = Y'$.

Case 3, $(\text{norm}(S \oplus_j T) \neq \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) = \text{ntrl})$:

The proof of the previous Case 2 applies analogously for this case, where $S \oplus_j T$ and $Q \oplus_i R$ swap their roles.

Case 4, $(\text{norm}(S \oplus_j T) \neq \text{ntrl}) \wedge (\text{norm}(Q \oplus_i R) \neq \text{ntrl})$:

The assumption of this case implies that $\neg((\text{norm}(S) = \text{ntrl}) \wedge (\text{norm}(S) = \text{ntrl}))$ and $\neg((\text{norm}(Q) = \text{ntrl}) \wedge (\text{norm}(R) = \text{ntrl}))$.

$rhs: \text{norm}((S \parallel T) \parallel (Q \oplus_i R))$

$\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_j T) \parallel (Q \oplus_i R))))$

$\stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q \oplus_i R)))$

Case differentiation according to the cases of Law 2.18.

Case i, $\text{norm}(Q) = \text{norm}(R)$:

$rhs: \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q \oplus_i R)))$

$\stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q)))$

$lhs: \text{norm}(((S \oplus_j T) \parallel Q) \oplus_i ((S \oplus_j T) \parallel R))$

$\stackrel{2.13, \text{Lemma B.2}}{=} \text{norm}((S \oplus_j T) \parallel Q)$

B. Uniqueness of the Normal Form: Proofs

Case a, $Q = \text{ntrl}$:

$$\begin{aligned}
 lhs: & \quad \text{norm}((S \oplus_j T) \parallel Q) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T))) \\
 rhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(\text{ntrl}))) \\
 & \stackrel{2.15}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \emptyset)) \\
 & = \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T)))
 \end{aligned}$$

Case b, $Q \neq \text{ntrl}$:

$$\begin{aligned}
 lhs: & \quad \text{norm}((S \oplus_j T) \parallel Q) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_j T) \parallel Q))) \\
 & \stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q))) \\
 rhs: & \quad \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q)))
 \end{aligned}$$

Case ii, $(\text{norm}(Q) \neq \text{norm}(R)) \wedge (\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset)$:

$$\begin{aligned}
 rhs: & \quad \text{norm}((S \oplus_j T) \parallel (Q \oplus_i R)) \\
 & \stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}((S \oplus_j T) \parallel (Q \oplus_i R)))) \\
 & \stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \text{coll}(Q \oplus_i R))) \\
 & \stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(S \oplus_j T) \cup \\
 & \quad \{ \text{norm}(Q) \oplus_i \text{norm}(R) \}))
 \end{aligned}$$

$$\begin{aligned}
 lhs: & \quad \text{norm}(((S \oplus_j T) \parallel Q) \oplus_i ((S \oplus_j T) \parallel R)) \\
 & \stackrel{2.13}{=} \text{reconv}(\text{sort}(X \cup Y))
 \end{aligned}$$

where X and Y are defined according to Law 2.13 as

$$\begin{aligned}
 X & = \text{cmn}(\text{coll}((S \oplus_j T) \parallel Q), \text{coll}((S \oplus_j T) \parallel R)) \\
 & \stackrel{B.3}{=} \text{coll}(S \oplus_j T) \cup \text{cmn}(\text{coll}(Q), \text{coll}(R))
 \end{aligned}$$

and

$$\begin{aligned}
Y &= \left\{ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} \left((S \oplus_j T) \parallel Q \right), \right. \right. \right. \right. \\
&\quad \left. \left. \left. \text{coll} \left((S \oplus_j T) \parallel R \right) \right) \right) \right) \\
&\quad \oplus_i \\
&\quad \left. \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} \left((S \oplus_j T) \parallel R \right), \right. \right. \right. \right. \\
&\quad \left. \left. \left. \text{coll} \left((S \oplus_j T) \parallel Q \right) \right) \right) \right) \right\} \\
\stackrel{\text{B.5}}{=} &\left\{ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(Q), \text{coll}(R) \right) \right) \right) \right. \\
&\quad \oplus_i \\
&\quad \left. \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(R), \text{coll}(Q) \right) \right) \right) \right\} \\
\stackrel{\text{B.6}}{=} &\left\{ \text{reconv} \left(\text{sort} \left(\text{coll}(Q) \right) \right) \right. \\
&\quad \oplus_i \\
&\quad \left. \text{reconv} \left(\text{sort} \left(\text{coll}(R) \right) \right) \right\} \\
\stackrel{\text{B.1}}{=} &\left\{ \text{norm}(Q) \oplus_i \text{norm}(R) \right\}
\end{aligned}$$

$rhs = lhs$, since the set X reduces completely to its element $\text{coll}(S \oplus_j T)$, as $\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset$ due to the assumption for this case.

Case iii, $(\text{norm}(Q) \neq \text{norm}(R)) \wedge (\text{cmn}(\text{coll}(Q), \text{coll}(R)) \neq \emptyset)$:

$$\begin{aligned}
rhs: &\quad \text{norm} \left((S \oplus_j T) \parallel (Q \oplus_i R) \right) \\
&\stackrel{2.12}{=} \text{reconv} \left(\text{sort} \left(\text{coll} \left((S \oplus_j T) \parallel (Q \oplus_i R) \right) \right) \right) \\
&\stackrel{2.17}{=} \text{reconv} \left(\text{sort} \left(\text{coll}(S \oplus_j T) \cup \text{coll}(Q \oplus_i R) \right) \right) \\
&\stackrel{2.18}{=} \text{reconv} \left(\text{sort} \left(\text{coll}(S \oplus_j T) \cup \text{coll}(\text{norm}(Q \oplus_i R)) \right) \right) \\
&\stackrel{2.13}{=} \text{reconv} \left(\text{sort} \left(\text{coll}(S \oplus_j T) \cup \right. \right. \\
&\quad \left. \left. \text{coll} \left(\text{reconv} \left(\text{sort} \left(X' \cup Y' \right) \right) \right) \right) \right)
\end{aligned}$$

where X' and Y' are defined according to Law 2.13 as

$$\begin{aligned}
X' &= \text{cmn}(\text{coll}(Q), \text{coll}(R)) \\
\text{and} \\
Y' &= \left\{ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(Q), \text{coll}(R) \right) \right) \right) \right. \\
&\quad \oplus_i \\
&\quad \left. \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll}(R), \text{coll}(Q) \right) \right) \right) \right\}
\end{aligned}$$

$$\begin{aligned}
lhs: &\quad \text{norm} \left((S \oplus_j T) \parallel Q \oplus_i ((S \oplus_j T) \parallel R) \right) \\
&\stackrel{2.13}{=} \text{reconv} \left(\text{sort} \left(X \cup Y \right) \right)
\end{aligned}$$

where X and Y are defined according to Law 2.13 as

$$\begin{aligned}
X &= \text{cmn} \left(\text{coll} \left((S \oplus_j T) \parallel Q \right), \text{coll} \left((S \oplus_j T) \parallel R \right) \right) \\
&\stackrel{\text{B.3}}{=} \text{coll}(S \oplus_j T) \cup \text{cmn}(\text{coll}(Q), \text{coll}(R))
\end{aligned}$$

and

B. Uniqueness of the Normal Form: Proofs

$$\begin{aligned}
Y &= \left\{ \begin{array}{l} \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} \left((S \oplus_j T) \parallel Q \right), \right. \right. \right. \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \left. \left. \left. \text{coll} \left((S \oplus_j T) \parallel R \right) \right) \right) \right) \\ \oplus_i \\ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} \left((S \oplus_j T) \parallel R \right), \right. \right. \right. \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \left. \left. \left. \text{coll} \left((S \oplus_j T) \parallel Q \right) \right) \right) \right) \end{array} \right\} \\
&\stackrel{\text{B.5}}{=} \left\{ \begin{array}{l} \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} (Q), \text{coll} (R) \right) \right) \right) \\ \oplus_i \\ \text{reconv} \left(\text{sort} \left(\text{diff} \left(\text{coll} (R), \text{coll} (Q) \right) \right) \right) \end{array} \right\}
\end{aligned}$$

$rhs = lhs$, as the set

$$\text{coll}(S \oplus_j T) \cup \text{coll}(\text{reconv}(\text{sort}(X' \cup Y')))$$

contains the same elements as the set $X \cup Y$, since the subsets $\text{coll}(\text{reconv}(\text{sort}(X' \cup Y')))$ and $X' \cup Y'$ contain the same elements.

□

B.1. Auxiliary Lemmata

For some of the preceding proofs, e.g. for the case $\text{norm}(P \parallel \text{ntrl}) = \text{norm}(P)$ or $\text{norm}((P \parallel Q) \oplus_i (P \parallel R)) = \text{norm}(P \parallel (Q \oplus_i R))$, we make use of the following lemmata.

Lemma B.1.

$$\text{norm}(Q) = \text{reconv}(\text{sort}(\text{coll}(Q)))$$

Proof. Structural induction on the term structure of Q .

Base cases:

- $Q = \text{ntrl}$: $lhs: \text{norm}(\text{ntrl}) \stackrel{2.10}{=} \text{ntrl}$
 $rhs: \text{reconv}(\text{sort}(\text{coll}(\text{ntrl}))) \stackrel{2.15}{=} \text{reconv}(\text{sort}(\emptyset))$
 $\stackrel{\text{sorting}}{=} \text{reconv}(\langle \rangle)$
 $\stackrel{2.36}{=} \text{ntrl}$
- $Q = \text{asset}(a)$: $lhs: \text{norm}(\text{asset}(a)) \stackrel{2.11}{=} \text{asset}(a)$
 $rhs: \text{reconv}(\text{sort}(\text{coll}(\text{asset}(a))))$
 $\stackrel{2.16}{=} \text{reconv}(\text{sort}(\{\text{asset}(a)\}))$
 $\stackrel{\text{sorting}}{=} \text{reconv}(\langle \text{asset}(a) \rangle)$
 $\stackrel{2.37}{=} \text{asset}(a)$

Inductive step:

- $Q = R \parallel S$: Showing: $\text{norm}(R \parallel S) = \text{reconv}(\text{sort}(\text{coll}(R \parallel S)))$
 Ind. hypothesis: $\text{norm}(R) = \text{norm}(\text{reconv}(\text{sort}(\text{coll}(R))))$ and
 $\text{norm}(S) = \text{norm}(\text{reconv}(\text{sort}(\text{coll}(S))))$

Case differentiation according to the cases of Law 2.12:

Case 1, $R = S = \text{ntrl}$:

$$\begin{aligned} \text{lhs: } \text{norm}(\text{ntrl} \parallel \text{ntrl}) &\stackrel{2.10}{=} \text{ntrl} \\ \text{rhs: } &\text{reconv}(\text{sort}(\text{coll}(\text{ntrl} \parallel \text{ntrl}))) \\ &\stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl}))) \\ &\stackrel{\text{cf. base case}}{=} \text{ntrl} \end{aligned}$$

Case 2, $R \neq \text{ntrl} \wedge S = \text{ntrl}$:

$$\begin{aligned} \text{lhs: } \text{norm}(R \parallel \text{ntrl}) &\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(R))) \\ \text{rhs: } &\text{reconv}(\text{sort}(\text{coll}(R \parallel \text{ntrl}))) \\ &\stackrel{2.17}{=} \text{reconv}(\text{sort}(\text{coll}(R))) \end{aligned}$$

Case 3, $R = \text{ntrl} \wedge S \neq \text{ntrl}$: Similarly to Case 2.

Case 4, $R \neq \text{ntrl} \wedge S \neq \text{ntrl}$:

$$\begin{aligned} \text{lhs: } \text{norm}(R \parallel S) &\stackrel{2.12}{=} \text{reconv}(\text{sort}(\text{coll}(\text{ntrl} \parallel R \parallel S))) \\ \text{rhs: } &\text{reconv}(\text{sort}(\text{coll}(R \parallel S))) \end{aligned}$$

- $Q = R \oplus_i S$: Showing: $\text{norm}(R \oplus_i S) = \text{reconv}(\text{sort}(\text{coll}(R \oplus_i S)))$
 Ind. hypothesis: $\text{norm}(R) = \text{norm}(\text{reconv}(\text{sort}(\text{coll}(R))))$ and
 $\text{norm}(S) = \text{norm}(\text{reconv}(\text{sort}(\text{coll}(S))))$

Case differentiation according to the three cases of Law 2.13:

Case 1, $\text{norm}(R) = \text{norm}(S)$:

$$\begin{aligned} \text{lhs: } \text{norm}(R \oplus_i S) &\stackrel{2.13}{=} \text{norm}(R) \\ \text{rhs: } &\text{reconv}(\text{sort}(\text{coll}(R \oplus_i S))) \\ &\stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(R))) \\ &\stackrel{I.H.}{=} \text{norm}(R) \end{aligned}$$

Case 2, $(\text{norm}(R) \neq \text{norm}(S)) \wedge (\text{cmn}(\text{coll}(R), \text{coll}(S)) = \emptyset)$:

$$\begin{aligned} \text{lhs: } \text{norm}(R \oplus_i S) &\stackrel{2.13}{=} \text{norm}(R) \oplus_i \text{norm}(S) \\ \text{rhs: } &\text{reconv}(\text{sort}(\text{coll}(R \oplus_i S))) \\ &\stackrel{2.18}{=} \text{reconv}(\text{sort}(\{\text{norm}(R) \oplus_i \text{norm}(S)\})) \\ &\stackrel{\text{sorting}}{=} \text{reconv}(\langle \text{norm}(R) \oplus_i \text{norm}(S) \rangle) \\ &\stackrel{2.37}{=} \text{norm}(R) \oplus_i \text{norm}(S) \end{aligned}$$

B. Uniqueness of the Normal Form: Proofs

Case 3, $(\text{norm}(R) \neq \text{norm}(S)) \wedge (\text{cmn}(\text{coll}(R), \text{coll}(S)) \neq \emptyset)$:

$$\text{lhs: } \text{norm}(R \oplus_i S) \stackrel{2.13}{=} \text{reconv}(\text{sort}(X \cup Y))$$

where X, Y represent the common parts and the reduced variation point as defined in Equation 2.13.

$$\text{rhs: } \text{reconv}(\text{sort}(\text{coll}(R \oplus_i S)))$$

$$\stackrel{2.18}{=} \text{reconv}(\text{sort}(\text{coll}(\text{norm}(R \oplus_i S))))$$

$$\stackrel{2.13}{=} \text{reconv}(\text{sort}(\text{coll}(\text{reconv}(\text{sort}(X \cup Y))))$$

$$= \text{reconv}(\text{sort}(\text{coll}(X \cup Y)))$$

since **reconv** and **sort** do not affect the number and the kind of elements in the sequence on which they are applied. □

Lemma B.2. *If the components of two compound element have identical normal forms, respectively, then also the two compound elements themselves have the identical normal form, i.e.*

$$\left((\text{norm}(P) = \text{norm}(P')) \wedge (\text{norm}(Q) = \text{norm}(Q')) \right) \Rightarrow \left(\text{norm}(P \parallel Q) = \text{norm}(P' \parallel Q') \right)$$

Proof. Structural induction on the term structure of P . □

Lemma B.3. *For the function **cmn** (cf. Figure A.7, Page 237) we observe the law*

$$\text{cmn}(\text{coll}(P \parallel Q), \text{coll}(P \parallel R)) = \text{coll}(P) \cup \text{cmn}(\text{coll}(Q), \text{coll}(R))$$

Proof. Structural induction on the term structure of P . □

Lemma B.4. *For the function **cmn** (cf. Figure A.7, Page 237) we observe the law*

$$\left(\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset \right) \Rightarrow \left(\text{cmn}(\text{coll}(P \parallel Q), \text{coll}(P \parallel R)) = \text{coll}(P) \right)$$

Proof. Structural induction on the term structure of P . □

Lemma B.5. *Let **diff** be specified as shown in Figure A.7 on Page 237. We observe the law*

$$\text{diff}(\text{coll}(P \parallel Q), \text{coll}(P \parallel R)) = \text{diff}(\text{coll}(Q), \text{coll}(R))$$

Proof. Structural induction on the term structure of P . □

Lemma B.6. *If two product families have no common elements, “subtracting” one product family from the other always yields the original product family, i.e.*

$$\left(\text{cmn}(\text{coll}(Q), \text{coll}(R)) = \emptyset \right) \Leftrightarrow \left(\text{diff}(\text{coll}(Q), \text{coll}(R)) = \text{coll}(Q) \right)$$

Proof. Structural induction on the term structure of Q . □

Lattices

The following definitions follow [DP90]. An algebraic structure $(\mathcal{L}, \sqcap, \sqcup)$ consisting of a set \mathcal{L} , a binary operation $\sqcap : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$ called *meet* and a binary operation $\sqcup : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$ called *join* is a *lattice* if it satisfies the following equations for all elements $x, y, z \in \mathcal{L}$:

- $x \sqcap y = y \sqcap x$ and $x \sqcup y = y \sqcup x$, *(Commutative laws)*
- $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ and $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$, *(Associative laws)*
- $x \sqcap (y \sqcup x) = x$ and $x \sqcup (y \sqcap x) = x$, *(Absorption laws)*
- $x \sqcap x = x$ and $x \sqcup x = x$, *(Idempotent laws)*

Note that the idempotent laws can be derived from the absorption laws.

Equivalently to the definition as an algebraic structure, a lattice can be defined as a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where for each $x, y \in \mathcal{L}$, there exists

1. a unique *greatest lower bound* (glb), which is called the *meet* of x and y , denoted by $x \sqcap y$,
2. and a unique *least upper bound* (lub), which is called the *join* of x and y , denoted by $x \sqcup y$.

C. Lattices

The definitions of glb and lub extend to finite sets of elements $A \subseteq \mathcal{L}$ as expected, which are then denoted by $\prod A$ and $\sqcup A$, respectively. Depending on the usage we use one or the other form for dealing with lattices. To simplify the notation, we denote the partially order set \mathcal{L} as well as the mathematical structure $(\mathcal{L}, \sqcap, \sqcup)$ both with the same symbol \mathcal{L} .

We call a lattice *bounded* if it has a greatest element called top (denoted by \top), and least element called bottom (denoted by \perp). Thus, a bounded lattice is an algebraic structure of the form $(\mathcal{L}, \sqcap, \sqcup, \top, \perp)$ such that $(\mathcal{L}, \sqcap, \sqcup)$ is a lattice, and \perp and \top are the identity elements for the join operation \sqcup , and meet operation \sqcap , respectively. A lattice is called *finite* iff \mathcal{L} is finite. Every (non-empty) finite lattice is bounded, i.e. it has a least element \perp , and a greatest element \top . A lattice $(\mathcal{L}, \sqcap, \sqcup)$ is called *complete* if all subsets of \mathcal{L} have both a meet and a join. In particular, this definition comprises also infinite meets and joins. Every finite lattice is complete, and every complete lattice is also bounded.

For the purpose of model checking software product families as introduced in Chapter 4 we are interested in lattices which provide a notion of complement. The following types of lattices are useful for such a purpose. A bounded lattice in which every element x has a complement, i.e. an element y such that $x \sqcup y = \top$ and $x \sqcap y = \perp$, is called a *complemented lattice*. If a complemented lattice is also distributive, every element x has a (necessarily) *unique* complement denoted by $\neg x$. A lattice is *distributive*, iff $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, and, dually, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$. A complete distributive lattice is called *Boolean* iff for all elements $x \in \mathcal{L}$ we have $x \sqcup \neg x = \top$ and $x \sqcap \neg x = \perp$. Some properties of a Boolean lattice are emphasized in a de Morgan lattice. In a *de Morgan* lattice (also: *orthocomplemented lattice*), every element x has a unique *dual* element $\neg x$, such that $\neg \neg x = x$ and $x \sqsubseteq y$ implies $\neg y \sqsubseteq \neg x$. As the negation is essential we denote a de Morgan lattice as a quadruple $(\mathcal{L}, \sqcap, \sqcup, \neg)$. Note that a Boolean lattice is a special case of a de Morgan lattice.

A typical Boolean lattice is the one induced by the power of some non-empty finite set S , i.e. the lattice $(\mathcal{P}(S), \sqsubseteq)$ where the ordering is given by set inclusion. Here, the entire set S takes the role of \top , while the empty set \emptyset takes the role of \perp . The lattice operation meet \sqcap is given by set intersection \cap , join \sqcup is given by set union \cup , and the lattice complement operation \neg is given by set complement $\bar{}$, respectively. In particular, in the context of a PF-CCS software product family we can represent sets of configuration as elements of such a powerset lattice: In a PF-CCS software product family with $N \in \mathbb{N}$ variation points we associate every complete configuration with an element of S (in a unique way). Thus, the set S has $|S| = 2^N$ elements which represent all N possible configurations of the software product family. Then, any element $s \in \mathcal{P}(S)$ of the powerset lattice over S corresponds to a (sub)set of configurations. In the context of model checking software product families, as we introduce it in Chapter 4, we require such a powerset lattice $\mathcal{P}(S)$ over a set of configurations in order to determine and represent those configurations in which certain properties hold.

The Modal μ -Calculus

The modal μ -calculus—in the form we use it—was introduced by Dexter Kozen in 1983 [Koz83]. The set \mathfrak{L}_μ of syntactically correct formulae of the modal μ -calculus is defined by the grammar in Chapter 4.1.1 on Page 163. Note that this is the same syntax as formulae of our multi-valued modal μ -calculus $mv\text{-}\mathfrak{L}_\mu$.

A \mathfrak{L}_μ -formula is interpreted over a Kripke structure (see e.g. [CGP99]). An action-labeled Kripke structure

$$\mathcal{T} = (States, \overset{\alpha}{\rightarrow}, L)$$

consists of a finite set *States* of states, a total transition relation $\overset{\alpha}{\rightarrow} \subseteq States \times \mathcal{A} \times States$ where individual transitions are labeled with actions in \mathcal{A} , and a labeling function $L : States \rightarrow \mathcal{P}(\mathcal{P})$ which associates every state with a subset of atomic propositions in \mathcal{P} , which hold in this state. With $\mathcal{T}.s$ we denote the state s of \mathcal{T} . If the set $I \subseteq States$ of initial states has to be considered explicitly, we add it to the definition of a Kripke structure \mathcal{T} and write $\mathcal{T} = (States, \overset{\alpha}{\rightarrow}, L, I)$. In the case of I being the singleton set $I = \{\sigma\}$, we omit the set notation and only write $\mathcal{T} = (States, \overset{\alpha}{\rightarrow}, L, \sigma)$. Note that a *product family LTS* (PF-LTS) as we introduce it in Chapter 3.12 (Page 125) is actually a special kind of a Kripke structure, where the state labeling function is not explicitly given.

The semantics of the modal μ -calculus is defined in terms of sets of states of a Kripke structure \mathcal{T} . Thus, formulae of the modal μ -calculus \mathfrak{L}_μ are ascribed to states of \mathcal{T} . A state s has property φ , denoted by $s \models_\mu \varphi$, if φ holds in that particular state. If

D. The Modal μ -Calculus

a state does not have the property φ we write $s \not\models_{\mu} \varphi$. We write $\mathcal{T} \models_{\mu} \varphi$ if a Kripke structure (also a PF-LTS) \mathcal{T} fulfills a property φ in any of its start states σ , i.e. iff $\mathcal{T}.\sigma \models_{\mu} \varphi$.

Let \mathcal{V} denote the set of all Variables. In a \mathfrak{L}_{μ} formula, the fixpoint operators μ and ν are the only variable binders. The valuation of free variables is given by the function

$$V : \mathcal{V} \rightarrow \mathcal{P}(\text{States})$$

which defines a *variable environment*. It yields for every free variable $Z \in \mathcal{V}$ the set of states in which this variable holds. The construct $V[Z \mapsto S]$ denotes the updated valuation environment where the variable Z is mapped to the set $S \subseteq \text{States}$, and where all other variable valuations are according to V . In order to emphasize that a satisfaction relation uses the environment V we also write \models_{μ}^V .

The satisfaction relation \models_{μ}^V with respect to a variable environment V is defined inductively on the structure of the formula as shown below, where $\varphi, \psi \in \mathfrak{L}_{\mu}$ are formulae, $q \in \mathcal{P}$ is an atomic proposition, and $\alpha \in \mathcal{A}$ is an action. Note that this is just a brief summary of the semantics of the μ -calculus as given in [Sti01] (adjusted to the setting of states instead of processes).

$$\begin{array}{ll}
s \models_{\mu}^V \mathbf{true} & \text{for all states } s \in \text{States} \\
s \not\models_{\mu}^V \mathbf{false} & \text{for all states } s \in \text{States} \\
s \models_{\mu}^V q & \text{iff } q \in L(s) \\
s \models_{\mu}^V \varphi \wedge \psi & \text{iff } s \models_{\mu}^V \varphi \text{ and } s \models_{\mu}^V \psi \\
s \models_{\mu}^V \varphi \vee \psi & \text{iff } s \models_{\mu}^V \varphi \text{ or } s \models_{\mu}^V \psi \\
s \models_{\mu}^V \langle \alpha \rangle \varphi & \text{iff } \exists t \in \{s' \mid \exists \alpha : s \xrightarrow{\alpha} s'\} : t \models_{\mu}^V \varphi \\
s \models_{\mu}^V [\alpha] \varphi & \text{iff } \forall t \in \{s' \mid \exists \alpha : s \xrightarrow{\alpha} s'\} : t \models_{\mu}^V \varphi \\
s \models_{\mu}^V Z & \text{iff } s \in V(Z) \\
s \models_{\mu}^V \nu Z. \varphi & \text{iff } s \in \bigcup \left\{ S \subseteq \text{States} \mid S \subseteq {}_{\mu} \llbracket \varphi \rrbracket_{V[Z \mapsto S]}^{\mathcal{T}} \right\} \\
s \models_{\mu}^V \mu Z. \varphi & \text{iff } s \in \bigcap \left\{ S \subseteq \text{States} \mid {}_{\mu} \llbracket \varphi \rrbracket_{V[Z \mapsto S]}^{\mathcal{T}} \subseteq S \right\}
\end{array}$$

We denote the set of all states $s \in \text{States}$ in which a formula φ interpreted over a Kripke structure \mathcal{T} holds by

$${}_{\mu} \llbracket \varphi \rrbracket_V^{\mathcal{T}} \stackrel{\text{def}}{=} \{s \in \text{States} : s \models_{\mu}^V \varphi \text{ under valuation } V\}$$

where the left subscript μ emphasizes that the semantics is for the standard μ -calculus. If it is clear which structure we mean, we omit the superscript \mathcal{T} , respectively.

Bibliography

- [AGM⁺06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. Refactoring product lines. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE'06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 201–210. ACM, 2006.
- [AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
- [Ake78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), June 1978.
- [AKGL10] Sven Apel, Christian Kästner, Armin Gröbinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010. 10.1007/s10515-010-0066-8.
- [All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
- [ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In José Meseguer and Grigore Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

Bibliography

- [ALMK10] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.
- [App10] Apple. Mac OS X Snow Leopard. <http://www.apple.com/de/macosx/>, January 2010.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.
- [AtBGF09] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Deontic logics for modeling behavioural variability. In David Benavides, Andreas Metzger, and Ulrich W. Eisenecker, editors, *VaMoS*, volume 29 of *ICB Research Report*, pages 71–76. Universität Duisburg-Essen, January 2009.
- [Bae05] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131 – 146, 2005. Process Algebra.
- [Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.
- [BB87] Tommaso Bolognesi and Ed Brinksmma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.
- [BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, 1996.
- [BC96] L. Brownsword and P. Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University, 1996.
- [BCD02] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002.
- [BCKB03] Len Bass, Paul Clements, Rick Kazmann, and Lisa Brownsword. Cel-siustech - a case study in product line development. *Software Architecture in Practice, Second Edition*, pages 369–399, 2003. ISBN 0-321-15495-9.

- [BDG⁺05] Peter Braun, Peter Dornbusch, Alexander Gruler, Alfred Helmerich, Patrick Keil, Nora Koch, Roland Leisibach, Luis Mandel, Jan Romberg, Bernhard Schätz, Thomas Wild, and Guido Wimmel. Study of worldwide trends and r&d programmes in embedded systems in view of maximising the impact of a technology platform in the area. Online at ftp://ftp.cordis.lu/pub/ist/docs/embedded/final-study-181105_en.pdf, November 2005. Study for the European Commission.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1st edition, 2000.
- [Bel08] Michael Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley and Sons, 2008. ISBN 978-0-470-14111-3.
- [BFG⁺93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.
- [BFG⁺93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.
- [BFG⁺08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme. Technical Report TUM-I0816, Technische Universität München, June 2008.
- [BG04] Glenn Bruns and Patrice Godefroid. Model checking with multi-valued logics. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 281–293. Springer, 2004.
- [BHK89] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. Addison-Wesley, New York, 1989. ISBN 0-201-41635-2.
- [BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

Bibliography

- [BK96] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. *Lecture Notes in Computer Science*, 1061:75–??, 1996.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008. ISBN-10 0-262-02649-X, ISBN-13 978-0-262-02649-9.
- [BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, 16(1), 2007. Available at <http://doi.acm.org/10.1145/1189748.1189753>.
- [BKPS07] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmänn. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb. 2007.
- [BKT84] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 76–95, Pittsburgh, PA, July 1984. Springer.
- [BLL⁺95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In *Proceedings of the Workshop on Verification and Control of Hybrid Systems III*, number 1066 in *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, October 1995.
- [BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT—structured assertion language for temporal logic. In *ICFEM’06: Proceedings of the 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, September 2006.
- [BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, 1998.
- [BO92] Don Batory and Sean O’Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.
- [Boc09] Otto Bock. Knee joints. Homepage, December 2009. Available online http://www.ottobock.com/cps/rde/xchg/ob_com_en/hs.xsl/1943.html.

- [Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Pearson Education (Addison-Wesley & ACM Press), May 2000.
- [BR05] Manfred Broy and Andreas Rausch. Das neue V-modell ® XT. *Informatik Spektrum*, 28(3):220–229, 2005.
- [Bro98] Manfred Broy. *Informatik. Eine grundlegende Einführung. Band I: Programmierung und Rechnerstrukturen, 2. Auflage*. Springer, Berlin, 1998. ISBN 3-540-64392-3.
- [Bro05] Manfred Broy. Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures - The **Janus** Approach. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, pages 47–81. Springer Verlag, July 2005.
- [Bru91] Glenn Bruns. A language for value-passing CCS. Internal Report ECS-LFCS-91-175, University of Edinburgh, August 1991.
- [BS01a] J. Bradfield and C. Stirling. Modal logics and mu-calculi. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 293–332. Elsevier, North-Holland, 2001.
- [BS01b] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces and Refinement*. Springer, New York, 2001. ISBN 0-387-95073-7.
- [Cam88] Michelle M. Campbell. A microcomputer-based knee controller. Technical Report 88/317/29, University of Calgary, July 1988.
- [CCG⁺02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv version 2: An opensource tool for symbolic model checking. In *CAV'02: Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, July 2002. Springer.
- [CE81a] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer.

Bibliography

- [CE81b] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, May 1981.
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CGLT09] Alarico Campetelli, Alexander Gruler, Martin Leucker, and Daniel Thoma. Don't know for multi-valued systems. In Zhiming Liu and Anders P. Ravn, editors, *ATVA'09: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, number 5799 in *Lecture Notes in Computer Science*, pages 289–305. Springer, 2009.
- [CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [Cha09] Robert N. Charette. This car runs on code. *IEEE Spectrum Online*, February 2009, 2009. <http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code#>.
- [CHS⁺10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE 2010, Proceedings of the 32nd International Conference on Software Engineering*, pages 335–344. ACM, May 2010. Acceptance rate: 13.7
- [CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.
- [Cle93] Rance Cleaveland. The concurrency workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
- [CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Aug 2001.

- [Com09] European Commission. ecall: Time saved turns into lives saved. Website of the European Commission, Information Society, 2009. Online at http://ec.europa.eu/information_society/activities/esafety/ecall/index_en.htm, Accessed 12-2009.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–521, December 1985.
- [CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *SPLC'07: Proceedings of the 11th International Software Product Line Conference*, pages 23–34, 2007.
- [dAFH⁺05] Luca de Alfaro, Marco Faella, Thomas A. Henzinger, Rupak Majumdar, and Mariëlle Stoelinga. Model checking discounted temporal properties. *Theoretical Computer Science*, 345(1):139–170, 2005.
- [dAH01] L. de Alfaro and T.A. Henzinger. Interface automata. In *FSE'01: Proceedings of the 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.
- [Dam94] Mads Dam. CTL* and ECTL* as fragments of the modal μ -calculus. *Theoretical Computer Science*, 126(1):77–96, 04 1994.
- [DCB09] Benjamin Delaware, William R. Cook, and Don Batory. Fitting the pieces together: a machine-checked model of safe composition. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09*, pages 243–252, New York, NY, USA, 2009. ACM.
- [DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press, 1972.
- [Die08] Bob Diertens. A process algebra software engineering environment. *CoRR*, abs/0806.2730, 2008. informal publication.
- [Dij68] E. W. Dijkstra. GOTO statements considered harmful. *Communications of the ACM*, 11:147–148, 1968.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

Bibliography

- [dKP92] F. deBoer, J. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *LICS'92: Proceedings of the 7th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1992.
- [DMN68] Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard. SIMULA 67. common base language. Technical Report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.
- [DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [EC81] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *ICALP'80: Proceedings of the 7th International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Berlin, 1981. Springer-Verlag.
- [EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for fragments of mu-calculus. In C. Courcoubetis, editor, *CAV'93: Proceedings of the 5th International Computer-Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.
- [EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ -calculus. In *LICS'86: Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.
- [EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1*. Springer, Berlin, 1985.
- [Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, August 2005. ISBN 0-13-185858-0.
- [Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
- [Fok00] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2000.
- [GBR08] AUTOSAR GBR. Autosar - automotive open system architecture. Official Website, 2008.

- [GHH07] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *ECBS'07: Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 349–358. IEEE CNF, March 2007.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Massachusetts, 2005.
- [GLS08a] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Calculating and modeling common parts of software product lines. In Birgit Geppert and Klaus Pohl, editors, *SPLC'08: Proceedings of the 12th International Software Product Line Conference*, pages 203–212. IEEE, 2008.
- [GLS08b] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In Gilles Barth and Frank de Boer, editors, *FMOODS'08: Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2008.
- [Gmb02] Robert Bosch GmbH. *Adaptive Fahrgeschwindigkeitsregelung ACC*. Christiani, Konstanz, Germany, April 2002. ISBN 978-3865220189.
- [Gut75] John Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Department of Computer Science, October 1975.
- [Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [HKM06] Peter Höfner, Ridha Khédri, and Bernhard Möller. Feature algebra. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.
- [HKM09] Peter Höfner, Ridha Khedri, and Bernhard Möller. An algebra of product families. *Software and Systems Modeling*, Special Issue, 2009.
- [HM84] Ellis Horowitz and John B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10(5):477–487, 1984.

Bibliography

- [Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985. Available online at: <http://www.usingcsp.com/>.
- [Hol03] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.
- [Inc09] The MathWorks Inc. Simulink. <http://www.mathworks.com/index.shtml>, January 2009. Visited 2009-01.
- [IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
- [ISS06] Fraunhofer ISST. Modellbasierte entwicklung in der automobilindustrie - das MOSES-projekt. Technical Report ISST-Bericht 77/06, Fraunhofer-Gesellschaft, ISST, April 2006.
- [IT96] ITU-TS. ITU-TS recommendation Z.120: Message sequence chart 1996 (MSC96), 1996.
- [Jur09] Ronald Jurgen. *X-by-Wire automotive systems*. SAE International, 2009. ISBN-13 978-0768021004.
- [KA08] Christian Kästner and Sven Apel. Type-checking software product lines - A formal approach. In *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 258–267. IEEE, 2008.
- [Kar53] M. Karnaugh. The map method for synthesis of combinational logic circuits. *AIEE Transactions, Part I Communication and Electronics*, 72:593–599, November 1953.
- [Kay93] A. C. Kay. The early history of Smalltalk. In J. A. N. Lee and J. E. Sammet, editors, *Proceedings of the 2nd Conference on History of Programming Languages, Cambridge (MA), USA*, special issue of ACM SIGPLAN Notices , (28)3, pages 69–95. ACM Press, New York (NY), USA, 1993.

- [KHNP90] Sholom G. Cohen Kyo C. Kang, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature oriented design analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21-ESD-90/TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [Koz83] Dexter Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27(3):333 – 354, 1983.
- [Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
- [Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [LFT09] Kim G. Larsen, Uli Fahrenberg, and Claus Thrane. A quantitative characterization of weighted kripke structures in temporal logic. In Petr Hlinený, Václav Matyáš, and Tomáš Vojnar, editors, *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany.
- [LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *ESOP'07: Proceedings of the 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79, April 2007.
- [Loe87] Jacques Loeckx. Algorithmic specifications: a constructive specification method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 9(4):646–661, 1987.
- [LPT09] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. *ASE'09: Proceedings of the 24rd IEEE/ACM International Conference on Automated Software Engineering*, 0:269–280, 2009.
- [LS93] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Benjamin-Cummings, Redwood Cliffs, CA., 1993.
- [LT88] K. G. Larsen and B. Thomsen. A modal process logic. In *LICS '88: Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 203–211, Washington, D.C., USA, July 1988. IEEE Computer Society Press.

Bibliography

- [LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
- [LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
- [LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, April 1974.
- [Man02] Mike Mannion. Using first-order logic for product line model validation. In Gary J. Chastek, editor, *SPLC*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2002.
- [MC01] Mila E. Majster-Cederbaum. Underspecification for a simple process algebra of recursive processes. *Theoretical Computer Science*, 266(1-2):935–950, 2001.
- [McI69] M. D. McIlroy. “Mass produced” software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.
- [MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *ESEC’95: Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer-Verlag, September 1995.
- [Mey87] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, 1987.
- [Mey97] B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second edition, 1997.
- [Mil80] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
- [Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, May 1999.

- [MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, August 1990.
- [New05] News@SEI. Faqs: An introduction to software product lines. Interview with Paul Clements, Online at <http://www.sei.cmu.edu/news-at-sei/columns/software-product-lines/2005/3/software-product-lines-2005-3.htm>, 2005. Visited 01-2009.
- [NR69] P. Naur and B. Randell. Software engineering report of a conference sponsored by the NATO science committee, Garmisch, Germany, 7th–11th October 1968, Jan 1969.
- [Ort09] Ed Ort. Introducing the Java EE 6 platform. <http://java.sun.com/developer/technicalArticles/JavaEE/JavaEE6Overview.html>, December 2009.
- [Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.
- [Par81] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.
- [Par08] David Lorge Parnas. Multi-dimensional software families: Document defined partitions on a set of products. Available online at <http://www.lero.ie/download.aspx?f=SPLC08.thumbs.pdf>, September 2008. Keynote Talk, 12th International Software Product Line Conference, Limerick, Ireland.
- [Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, Germany, 1962. (In German).

Bibliography

- [Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Long Beach, Ca., USA, October 1977. IEEE Computer Society Press.
- [Pro99] B. J. Pronk. Medical product line architectures – 12 years of experience. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 357–367, San Antonio, TX, USA, February 1999. Kluwer Academic Publishers.
- [PS08] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 347–350. IEEE, 2008.
- [QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
- [RC04] Nelson Souto Rosa and Paulo Roberto Freire Cunha. A software architecture-based approach for formalising middleware behaviour. *Electronic Notes in Theoretical Computer Science*, 108:39–51, 2004.
- [Ros05] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, New York, online edition 2005 edition, 1998, Revised 2005. Online available at <http://web.comlab.ox.ac.uk/people/Bill.Roscoe/publication/68b.pdf>.
- [Sch98] Bernhard Schätz. *Ein methodischer Übergang von asynchron zu synchron kommunizierenden Systemen*. PhD thesis, Fakultät für Informatik, Technische Universität München, 1998. in German.
- [Sch99] Douglas C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report, SIGS*, 11(1), Jan 1999.
- [Sch01] Uwe Schöning. *Algorithmik*. Spektrum Akademischer Verlag GmbH, Heidelberg-Berlin, 2001.
- [Sch08] Kathrin Scheidemann. *Verifying Families of System Configurations*. PhD thesis, Fakultät für Informatik, Technische Universität München, April 2008.

- [SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49(7):717 – 739, 2007.
- [SEI] Carnegie Mellon University Software Engineering Institute. Software product line hall of fame. Website, http://www.sei.cmu.edu/productlines/plp_hof.html. Visited 12-2008.
- [SG05] Sharon Shoham and Orna Grumberg. Multi-valued model checking games. In Doron Peled and Yih-Kuen Tsay, editors, *ATVA '05: Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 354–369. Springer, October 2005.
- [SS71] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings, 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also, Programming Research Group Technical Monograph PRG–6, Oxford University.
- [SSP07] Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The java module system: core design and semantic definition. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr, editors, *OOPSLA '07*., pages 499–514. ACM, 2007.
- [ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3):229–269, 1997.
- [Sta84] Thomas A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.
- [Sti95] Colin Stirling. Local model checking games. In *CONCUR'95: Proceedings of the 6th International Conference on Concurrency Theory*, pages 1–11, London, UK, 1995. Springer-Verlag.
- [Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer, 2001.
- [Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

Bibliography

- [Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE'09: Proceedings of the 31th International Conference on Software Engineering*. IEEE Computer Society, May 2009.
- [Tof92] C.M.N. Tofts. Describing social insect behaviour using process algebra. *Transactions of the Society for Computer Simulation*, pages 227–283, 1992.
- [vHL89] Ivo van Horebeek and Johann Lewi. *Algebraic Specifications in Software Engineering: An Introduction*. Springer-Verlag, New York, NY, 1989. ISBN 3-540-51626-3, 0-387-51626-3.
- [VN98] Simone Vegliani and Rocco De Nicola. Possible worlds for process algebras. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 179–193. Springer, September 1998.
- [VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, January 1994.
- [WG04] Diana L. Webber and Hassan Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305 – 331, 2004. Software Variability Management.
- [Wir90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, NY, 1990.
- [WNSSL05] Cédric Wilwert, Nicolas Navet, Yeqiong Song, and Françoise Simonot-Lion. Design of automotive X-by-wire systems, January 18 2005.
- [Wol] Pierre Wolper. A translation from full branching time temporal logic to one letter propositional dynamic logic with looping. unpublished manuscript.
- [Zav93] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–30, Aug 1993. DOI 10.1109/2.223539.

- [Zil74] S. N. Zilles. Algebraic specification of abstract data types. Computation structures group memo 119, Laboratory for Computer Science, MIT, 1974.

- [ZLWL07] Guang Zheng, Shaorong Li, Jinzhao Wu, and Lian Li. A non-interleaving denotational semantics of value passing CCS with action refinement. In Franco P. Preparata and Qizhi Fang, editors, *FAW*, volume 4613 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2007.

Bibliography