Feature Constraint Propagation along Configuration Links for Advanced Feature Models

vorgelegt von
Diplom-Informatiker
Alexander Rein-Jury
(Berlin)

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuss :

Vorsitzender:	Prof. Dr. Uwe Nestmann
Gutachter:	Prof. Dr. Stefan Jähnichen
Gutachter:	Prof. Dr. Klaus Pohl

Tag der wissenschaftlichen Aussprache: 19. November 2013

Berlin 2013
D 83

Acknowledgment

I would like to thank all people who supported and motivated me during the last few years.

In particular I would like to thank...

...Prof. Dr. Stefan Jähnichen for giving me the opportunity to work in his software engineering group at Technische Universität Berlin and to write this thesis.

...my colleague Dr. Mark-Oliver Reiser for his continuous professional support, his suggestions, stimulating discussions and proofreading of many parts of this thesis. Special thanks for pushing me to further elaborate the experimental study and for always answering my questions and giving me constructive feedback.

...my colleague Alexandra Mehlhase for giving suggestions for improvement and feedback, especially for the structure of the thesis and related talks. Your different point of view was always very helpful for me.

...the rest of my colleagues for their critical questions and emerging discussions during and after my talks.

...Anastasia Cmyrev of Daimler AG for inviting me to Daimler (Research and Advanced Engineering) in Böblingen and providing me the possibility to present my work there.

...Daniel Hopp of Daimler AG for the industrial case study and his support before, during and after this study. You were at all times a nice, competent and helpful contact person for me. In addition, I would like to thank you for the guided tour.

...my wonderful and lovely wife Sabrina Jury for proofreading this thesis and for her great understanding and patience with me. Thank you for long and productive conversations and discussions and for bearing my seemingly never-ending monologues about this work. You opened my eyes for a lot of small things making life so beautiful. I enjoy every day, every hour and every minute with you. I love you!

...my great parents Bärbel and Eberhard Rein for supporting me in every circumstance and in every of my projects. You provided me with the foundation of everything I achieved and everything I became in my life. You are always there for me and send me your love. Thank you for being such great parents. I cannot imagine better ones. I love you!

Danksagung

Ich möchte mich bei allen bedanken, die mich während der letzten paar Jahre unterstützt und motiviert haben.

Speziell möchte ich mich bedanken bei...

...Prof. Dr. Stefan Jähnichen für die Möglichkeit in seiner Softwaretechnik-Gruppe bei der Technischen Universität Berlin zu arbeiten und diese Arbeit zu schreiben.

...meinem Kollegen Dr. Mark-Oliver Reiser für seine kontinuierliche fachliche Unterstützung, seine Ratschläge, anregende Diskussionen und das Korrekturlesen von vielen Teilen der Arbeit. Vielen Dank, dass du mich dazu gedrängt hast, die experimentelle Studie weiter auszuarbeiten und dafür, dass du immer meine Fragen beantwortet hast und mir viel konstruktives Feedback gegeben hast.

...meiner Kollegin Alexandra Mehlhase für ihre Verbesserungsvorschläge und ihr Feedback, speziell zu der Struktur der Arbeit und damit verbundene Vorträge. Deine andere Betrachtungsweise war stets sehr hilfreich für mich.

...dem Rest meiner Kollegen für ihre kritischen Fragen und aufkommende Diskussionen während und nach meinen Vorträgen.

...Anastasia Cmyrev von der Daimler AG für die Einladung zu Daimler (Forschung und Vorentwicklung) in Böblingen und die Möglichkeit dort meine Arbeit vorzustellen.

...Daniel Hopp von der Daimler AG für die industrielle Fallstudie und die Unterstützung vor, während und nach der Studie. Du warst jederzeit ein netter, hilfsbereiter und kompetenter Ansprechpartner für mich. Außerdem möchte ich mich bei dir für die Werksführung bedanken.

...meiner wundervollen und reizenden Frau Sabrina Jury für das Korrekturlesen dieser Arbeit und ihre große Unterstützung und Geduld mit mir. Danke für die langen und produktiven Gespräche und Diskussionen und dafür, dass du meine scheinbar endlosen Monologe über diese Arbeit ertragen hast. Du hast mir die Augen für viele kleine Dinge geöffnet, die das Leben so wundervoll machen. Ich genieße jeden Tag, jede Stunde und jede Minute mit dir. Ich liebe dich!

...meinen großartigen Eltern Bärbel und Eberhard Rein für ihre Unterstützung in jeder Lebenslage und in jedem meiner Vorhaben. Ihr habt mir die Voraussetzungen für alles gegeben, was ich im Leben erreicht habe und geworden bin. Ihr seid immer für mich da und lasst mir stets eure Liebe zukommen. Ich danke euch, dass ihr so tolle Eltern seid. Ich kann mir keine Besseren vorstellen. Ich habe euch lieb!

"The highest reward for a person's toil is not what they get for it, but what they become by it."

John Ruskin (08 February 1819 — 20 January 1900),
British writer, artist, art historian and social philosopher

„Der höchste Lohn für unsere Bemühungen ist nicht das, was wir dafür bekommen, sondern das, was wir dadurch werden."

John Ruskin (08. Februar 1819 — 20. Januar 1900),
britischer Schriftsteller, Maler, Kunsthistoriker und Sozialphilosoph

Abstract

Product line engineering is faced with several challenges when it comes to dealing with highly complex variability. Several approaches to tackle these challenges have been proposed in recent years. Among others, some of them provide the possibility to decompose a large-scale product line and to describe the variability in distinct variability models, tailored to individual stakeholders. These variability models are then hierarchically related in order to form the overall product line by integrating different views and different levels of abstraction. Configuration links [Rei08] are a concept for this purpose. A configuration link is a directed relation between two feature models and allows to automatically derive a configuration of the lower-level feature model from a given configuration of the higher-level feature model.

However, the use of configuration links raises some new challenges if constraints are contained in the individual feature models. Constraints forbid certain configurations of a feature model and occur almost in every realistic model. In combination with configuration links, constraints of one feature model do not necessarily affect only this individual model but can also influence higher-level feature models. This can result in locally valid configurations of a higher-level feature model which are invalid with respect to the overall product line because they lead to derived configurations of lower-level feature models which violate their constraints.

This dissertation presents the technique of feature constraint propagation, a new way to deal with constraints in such hierarchically organized product lines. Feature constraint propagation transforms constraints of a lower-level feature model into constraints of a higher-level feature model, making all their implicit effects explicit in the higher-level model. All configurations fulfilling the propagated constraints lead then to derived configurations fulfilling the original constraints and vice versa. Feature constraint propagation thus reveals the actual meaning of constraints of a lower-level model within the taxonomy of a higher-level model, which makes the higher-level model self-contained and helps to detect errors in the product line specification.

The proposed technique is defined on a conceptual level, formalized, verified, implemented and evaluated. The evaluation comprises an experimental case study (automated functional and performance tests) and an industrial case study (at Daimler AG).

Keywords: feature constraint propagation, configuration link, feature constraint, feature link, feature model, highly complex variability, product line engineering

Zusammenfassung

Produktlinienentwicklung steht verschiedenen Herausforderungen gegenüber, wenn die vorhandene Variabilität sehr hoch ist. In den letzten Jahren wurden verschiedene Ansätze vorgestellt, um diese Herausforderungen anzugehen. Unter anderem bieten einige von ihnen die Möglichkeit, eine große Produktlinie zu zerlegen und die Variabilität in verschiedenen Variabilitätsmodellen, die auf einzelne Stakeholder zugeschnitten sind, zu beschreiben. Diese Variabilitätsmodelle werden dann hierarchisch in Beziehung gesetzt, um die übergeordnete Produktlinie durch die Integration verschiedener Sichten und Abstraktionsebenen zu bilden. Configuration Links [Rei08] sind ein Konzept dafür. Ein Configuration Link ist eine gerichtete Beziehung zwischen zwei Feature Modellen und erlaubt die automatische Ableitung einer Konfiguration des untergeordneten Feature Modells von einer gegebenen Konfiguration des übergeordneten Feature Modells.

Allerdings wirft die Benutzung von Configuration Links auch einige neue Herausforderungen auf, wenn Constraints in den einzelnen Feature Modellen enthalten sind. Constraints verbieten bestimmte Konfigurationen eines Feature Modells und kommen in fast jedem realistischen Modell vor. In Kombination mit Configuration Links beeinflussen Constraints eines Feature Modells nicht unbedingt nur dieses einzelne Modell, sondern können auch Auswirkungen auf übergeordnete Modelle haben. Das kann lokal gültige Konfigurationen eines übergeordneten Feature Modells zur Folge haben, die ungültig bezüglich der gesamten Produktlinie sind, weil sie zu abgeleiteten Konfigurationen von untergeordneten Feature Modellen führen, die deren Constraints verletzen.

Diese Dissertation präsentiert die Technik der Feature Constraint Propagierung, einen neuen Weg, um mit Constraints in solchen hierarchisch angeordneten Produktlinien umzugehen. Die Feature Constraint Propagierung transformiert Constraints eines untergeordneten Feature Modells in Constraints eines übergeordneten Feature Modells, was all ihre impliziten Effekte in dem übergeordneten Modell explizit macht. Alle Konfigurationen, die die propagierten Constraints erfüllen, führen dann zu abgeleiteten Konfigurationen, die die ursprünglichen Constraints erfüllen und umgekehrt. Die Feature Constraint Propagierung deckt also die tatsächliche Bedeutung der Constraints eines untergeordneten Modells in der Taxonomie eines übergeordneten Modells auf, was das übergeordnete Modell in sich schließt und hilft, Fehler in der Produktlinien-Spezifikation aufzudecken.

Die vorgestellte Technik ist auf konzeptioneller Ebene definiert, formalisiert, verifiziert, implementiert und evaluiert. Die Evaluation umfasst eine experimentelle Fallstudie (automatisierte Funktions- und Performanztests) und eine industrielle Fallstudie (bei der Daimler AG).

Schlüsselwörter: Feature Constraint Propagierung, Configuration Link, Feature Constraint, Feature Link, Feature Modell, hoch komplexe Variabilität, Produktlinienentwicklung

Contents

Chapter 1

Introduction

Industrial domains of software and system development, like the automotive or aerospace industries, are characterized by a concurrence of complex product variation (model ranges, end-customer configuration, country variants, special equipment, etc.) and an extremely complex organizational context in which development takes place (large, global corporations, manufacturer / supplier interaction, long product life-cycles, etc.). The paradigm of *product line engineering* [CN02, PBL05] is perfectly suited for these domains. Its basic idea is the reuse of development artifacts in various products of a product line. A *product line* is a set of products with common and variable properties such that it is advantageous to study the common properties before analyzing the individual products [Par76]. Product line engineering separates two processes – domain engineering and application engineering – which consist of several sub-processes each. The main goal of the *domain engineering* process is the establishment of the reusable platform. During this process, the variabilities and commonalities of the product line are gathered and documented in some kind of variability model, e.g. a *feature model* [KCH$^+$90, CBUE02]. A variability model describes all products of a product line. This means that it defines *constraints* that all products have to fulfill. On completion of domain engineering, concrete products are derived by exploiting the defined variabilities and commonalities of the product line in the *application engineering* process.

In industrial domains, product line engineering is primarily applied to systems with small or medium size or only to individual subsystems of large-scale product lines. The application of product line engineering and variability management techniques to the entire product range on a global scale is still faced with several significant challenges [BLPW04, Rei08]. The first and obvious one is the resulting overall complexity of the variability. Then, many manufacturers, for example in the automotive domain, incorporate a large number of subsystems of external suppliers, each forming an independent product line with its own variability, turning the manufacturer's product line into an aggregate of a multitude of lower-level, subordinate product lines with diverse scopes and target groups [vOvdLKM00, WW03, RKW09]. But even within a single company, manufacturer or supplier, several different views on variability are often required due to distinct viewpoints of stakeholders and diverse life-cycles of subcomponents or individual development projects. Also heterogeneous development methods and tools often require different forms of variability

representations. All these considerations suggest that a single, monolithic variability model is inappropriate in highly complex use cases of product line engineering. Besides other approaches to manage complex variability [MHP+07, HHS+11, ACLF12], Reiser [Rei08] proposes to split the whole product line into several smaller subordinate product lines, each equipped with its own variability model. The individual variability models are then related to form the *overall product line* by integrating different views and different levels of abstraction.

The idea of managing the variability in different related variability models, however, reveals some new challenges. The constraints of one variability model can have implicit impacts on related variability models. These implicit constraints are hidden from the engineers and can outweigh the benefits of this approach or have at least an adverse effect on it. The main goal of this thesis is to introduce the technique of *feature constraint propagation* that tackles this challenge by making the implicit effects explicit and thus visible for the engineers.

Section 1.1 briefly introduces the approach for hierarchical organization of product line artifacts, which forms the foundation for this thesis, and discusses the arising challenges in more detail. The proposed solution to tackle these challenges – the technique of feature constraint propagation – is introduced and motivated subsequently in Section 1.2. The main contributions of this thesis are formulated in Section 1.3. Then, Section 1.4 differentiates our approach of feature constraint propagation from other constraint propagation approaches. Finally, Section 1.5 gives an overview on the complete structure of this thesis.

1.1 Hierarchical Organization with Configuration Links and arising Challenges

Configuration links [Rei08] are a concept for relating different variability models in hierarchically organized product lines. This concept (a) provides a means to partition a complex product line into several smaller, subordinate product lines – so-called *artifact lines* – and (b) allows to introduce orthogonal, i.e. differently structured, views on a product line's variability. Thus, the application of two classical principles of computer science to product line engineering becomes possible: information hiding and divide and conquer. Configuration links are centered around traditional feature modeling [KCH+90]. In addition, several extensions of feature modeling, as e.g. feature cardinalities, feature attributes and feature inheritance, are supported. We call feature models with these extensions *advanced feature models*. Subordinate product lines and orthogonal views are each represented by a feature model and configuration links are used to relate them. Each of these configuration links defines how to configure the feature model representing a lower-level, subordinate product line based on a given configuration of a higher-level product line's feature model. Accordingly, configuration links integrate parts of the configuration process into domain engineering: parts of the configurations are already predefined during development time and product configuration is no longer only a manual and interactive process taking place when an individual product is created.

Fig. 1.1 shows an example of a fictitious hierarchically organized car product line. The superordinate product line is represented by the so-called core feature model *CoreFM*. Each other node in the figure depicts another feature model that represents either an orthogonal view on the product line's variability (*CustomerFM*) or the subordinate product line (artifact line) of an individual subsystem of the car (*BodyElectronicsFM*, *EngineCtrlFM*, etc.). The arrows denote configuration links connecting these feature models and thus relating the different views and subordinate product lines. For example, the configuration link from *CoreFM* to *TelematicsFM* defines how to configure the telematics subsystem depending on a given configuration of the superordinate product line.

In addition to the creation of artifact lines, configuration links also allow to model the variability of compositional structures in a hierarchical manner. This is called *compositional variability management* [Rei08, RKW09]. For example, in component-based design every component can be equipped with its own feature model and configuration links can be defined from this feature model to all feature models of subcomponents. This allows to derive configurations of (the feature models of) subcomponents from a given configuration of (the feature model of) the higher-level component.

The effort when configuring a product decreases drastically if configuration links are used since only the topmost feature model has to be configured manually and the configurations of the lower-level feature models are derived automatically. This means that parts of the system's variability are hidden on topmost level (*configuration hiding*). Note that the topmost model can but do not have to configure the lower-level models completely. Some lower-level models could also need manual intervention.

Figure 1.2 depicts the two topmost feature models of the car product line and the configuration link between them. The customer-oriented feature model only allows to choose a market and a model. The configuration link *CL* consists of two configuration decisions, depicted in the dotted box. Each of them, in turn, consists of a criterion and an effect denoted in the form $\langle criterion \rangle \mapsto \langle effect \rangle$.

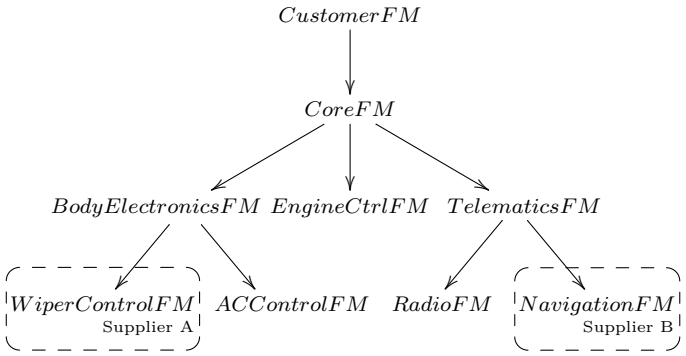


Figure 1.1: A hierarchically organized car product line

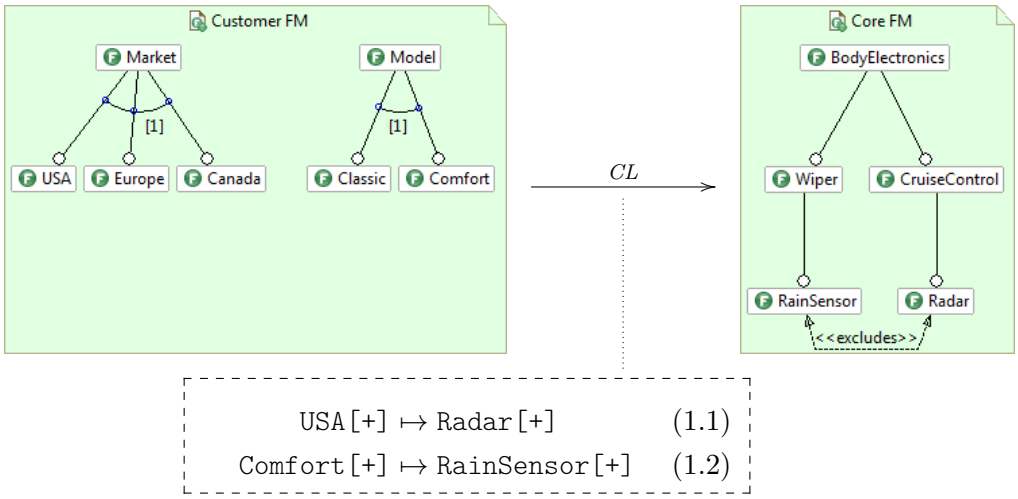


Figure 1.2: A configuration link from a customer-oriented (left) to a technical feature model (right) in a car product line

The criterion specifies under which conditions the effect is applied. Configuration decision 1.1 states that cars for the U.S. market have a radar and configuration decision 1.2 specifies that comfort cars are equipped with rain sensors. Consider the configuration with the features `Europe` and `Comfort` being selected. Then the application of the configuration link leads to a partial configuration of the core feature model in which only the feature `RainSensor` is selected. `CruiseControl` and `Radar` remain unconfigured and can be selected or deselected manually.

The concept of configuration links is stable, has been implemented in tools [AJL+10, CVM12] and applied in case studies and projects. However, there are still some challenges that arise when constraints, i.e. dependencies between features, are added to lower-level feature models. In the context of a hierarchically organized product line like that of the example, constraints defined in one feature model do not necessarily affect only this single model but can also indirectly influence connected feature models. This means that valid configurations of higher-level feature models could thus lead to invalid configurations of lower-level feature models. In the example, the selection of the features `USA` and `Comfort` of the customer-oriented feature model would lead, through the application of the configuration link, to a configuration of the core feature model with `Radar` and `RainSensor` being selected. However, the excludes relationship between these features (denoted by the dotted line between them) forbids this configuration. This means that the defined technical dependency of the core model makes the entire comfort package indirectly unavailable for the U.S. market, which certainly indicates an error within the variability definition.

How can we deal with these implicit effects? One possibility is to formulate the overall variability specification, i.e. all feature models with their constraints and the

configuration link, as a constraint satisfaction problem [Tsa95] and use a constraint solver to employ some analyses. For example, this would allow

- to check a given product for validity or
- to check the product line for satisfiability, i.e. whether a valid product exists.

While this is a perfectly feasible and effective approach for many use cases, it is less suitable in some other use cases because of the following shortcomings.

1. access to the entire variability specification is required
2. the actual meaning of a target feature model's constraint within the source feature model's taxonomy is not revealed

№ 1 poses a critical problem whenever parts of the product line are not accessible, maybe an artifact line is part of the internal development of a separate legal entity, e.g. an external supplier. But even if the whole variability specification is accessible, it is often very helpful to be able to clearly separate the different views and levels of abstraction from one another and to be able to temporarily work with one feature model without regarding the whole variability representation. For example, if the models are maintained and evolved by independent organizational units within a single company. The significance of № 2 is best illustrated with the example above: the target-side constraint indirectly rendered the `Comfort` package entirely unavailable for the U.S. market, which is surely unacceptable from a product management and marketing point of view. Without making this indirect effect of the core feature model's constraint explicit within the customer-oriented model, such information is hidden and practically unascertainable for the stakeholders involved.

Because of these shortcomings of constraint solving techniques in connection with configuration links, we propose the technique of feature constraint propagation, a fundamentally different approach, to deal with the challenges mentioned above.

1.2 Feature Constraint Propagation as Solution

The technique of *feature constraint propagation* complements the concept of configuration links and provides a new way to deal with constraints within hierarchically organized product lines. It transforms constraints of lower-level feature models backwards along configuration links into constraints of higher-level feature models. While being entirely redundant with respect to the overall variability specification, the propagated constraints serve to reveal the impacts of the constraints of lower-level models within the taxonomy of the higher-level model. The technique shall complement – not replace – existing reasoning based approaches to avoid the above-mentioned shortcomings, which is discussed later in this thesis. In the example of Figure 1.2, the propagation of the depicted constraint of the core feature model would reveal the implicit constraint that the comfort package is unavailable for the U.S. market. Without feature constraint propagation, the identification of such implicit effects is, already in small models, a very error-prone and time-consuming task. This is because configuration decisions of configuration links can, on the one

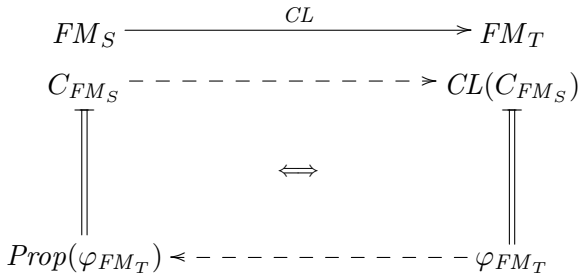


Figure 1.3: Correctness and minimality of feature constraint propagation

hand, contain complex criteria and effects and, on the other hand, be redundant or contradictory to other configuration decisions. These facts can lead to very complex dependencies between source and target features. In large-scale product lines the manual identification of the impacts of constraints of lower-level models on higher-level models is thus almost impossible.

Feature constraint propagation is able to propagate arbitrary constraints that can be expressed by propositional logic along arbitrary configuration links. A propagated constraint covers exactly the meaning of the original constraint in the higher-level model. We express this by two essential properties of feature constraint propagation: *correctness* and *minimality*. Figure 1.3 illustrates them. It depicts two feature models FM_S and FM_T with a configuration link CL between them. Given an arbitrary source configuration C_{FM_S} and an arbitrary target-side constraint φ_{FM_T}. The application of the configuration link to the source configuration leads to a derived target configuration $CL(C_{FM_S})$ and the application of feature constraint propagation to the target-side constraint delivers a source-side constraint $Prop(\varphi_{FM_T})$. Correctness means that the satisfaction of the propagated constraint on the source-side is a sufficient condition for the satisfaction of the original constraints on the target-side. The fact that the mentioned property is also a necessary condition is called minimality.

$$C_{FM_S} \models Prop(\varphi_{FM_T}) \Longrightarrow CL(C_{FM_S}) \models \varphi_{FM_T} \qquad \text{correctness}$$
$$C_{FM_S} \models Prop(\varphi_{FM_T}) \Longleftarrow CL(C_{FM_S}) \models \varphi_{FM_T} \qquad \text{minimality}$$

The term ‘minimality’ was chosen for this property since the propagated constraint is minimal, i.e. as weak as possible.

The consequences of correctness and minimality are illustrated in Figure 1.4. It depicts the sets of all source and all target configurations (the outer circles). The set of target configurations is narrowed to a subset (the green circle on the right side) by a set of constraints. The propagation of these constraints leads to a narrowed set of source configurations (the green circle on the left side). Correctness means that all configurations of this subset lead, through the application of the configuration link, to target configurations fulfilling the original constraints (the intersection on the right side). Vice versa, all source configurations which are not in this subset lead to target configurations violating the original constraints. If the configuration link does

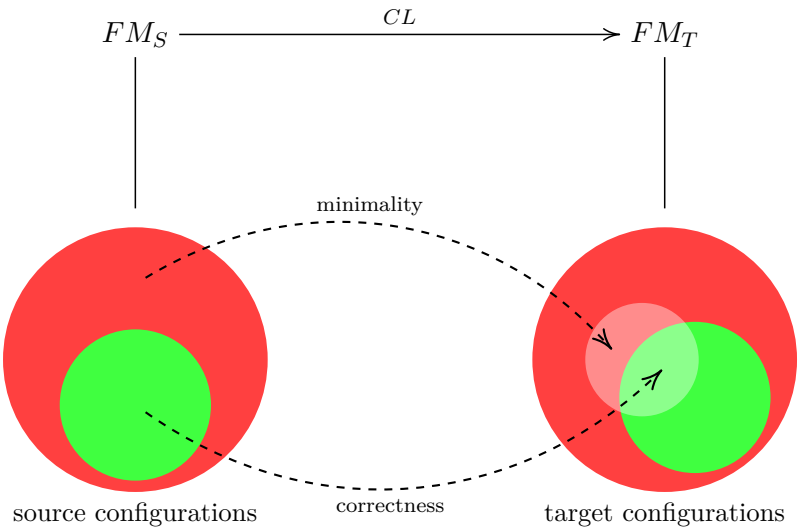


Figure 1.4: Consequences of correctness and minimality of feature constraint propagation

not produce configurations violating a cardinality restriction, it holds that all valid source configurations also lead to valid target configurations after the propagation. Vice versa, all source configurations that are invalid because they violate a source constraint also lead to invalid target configurations.

One might wonder how this can hold because there are target-side constraints that do not affect the higher-level feature model at all. These constraints lead to the propagated constraint `true`. So how can correctness and minimality hold? Although this question is answered in detail during the introduction of the concept, at this point we want to anticipate the answer in order to avoid any misunderstanding. Correctness and minimality can be ensured because (1) we do not take manual configuration of the derived configuration into account and (2) we distinguish only between the satisfaction and the violation of a constraint (even in the case of partial configurations). We do not consider the case that a constraint cannot be evaluated and is thus neither satisfied nor violated, as some other feature modeling approaches do. Consider the example in Figure 1.2, but assume the configuration link to contain only configuration decision 1.1. Then the constraint `true` is propagated. All (partial) configurations of the source feature model lead, through the application of the configuration link, to valid configurations of the target model since the feature `RainSensor` is never selected and the constraint is thus fulfilled by any derived configuration. Obviously, manual configuration of the unconfigured feature `RainSensor` can then lead to the violation of the constraint. However, such cases can be analyzed locally with existing tools (e.g. feature configurators). Our motivation to exclude manual configuration is substantiated later in this thesis.

Now we know what feature constraint propagation is. But why should we use it? Why can it be so important to make implicit constraints explicit? The main benefits of feature constraint propagation are enumerated in the following.

1. **Feature constraint propagation allows some stakeholders to (temporarily) disregard other feature models and configuration links.** With feature constraint propagation, constraints of all feature models of a hierarchically organized product line can be propagated stepwise bottom-up to the top-level and added to the individual feature models. Then, the engineers of a subsystem can entirely disregard all other feature models and configuration links since all implicit and hidden effects resulting from related feature models are made explicit in every model. This can be reasonable since all constraints are then expressed in the taxonomy of "their" feature model and the complexity of the overall product line is hidden or because they might not have access to the other feature models and their constraints for reasons of intellectual property rights. After the propagation, engineers can create arbitrary (valid) configurations of a feature model because they can be sure that these configurations lead to valid configurations on all levels of variability. This means that all locally valid configurations are also valid with respect to the overall product line. Since feature constraint propagation only propagates constraints based on propositional logic, this property only holds if the configuration link does not produce configurations violating a cardinality restriction.

2. **Feature constraint propagation helps to detect errors in product line specifications.** It is imaginable that the propagation of a constraint leads to anomalies or, in the worst-case, to a non-satisfiable feature model (i.e. a feature model that does not represent any products). The last-mentioned case would mean that all configurations of a higher-level feature model result in invalid configurations of a lower-level feature model, through the application of the configuration link. This indicates an error in the product line specification. But also constraints that do not lead to anomalies or non-satisfiable feature models can pose errors in product line specifications. In the example of Figure 1.2, the propagated constraint that the comfort package is unavailable in the U.S. (caused by a technical dependency) is surely unacceptable from a product management and marketing point of view.

3. **Feature constraint propagation allows to validate properties defined by configuration links.** Feature constraints cannot only be used to propagate constraints contained in the variability specification but also to propagate artificially created constraints to validate certain properties of the configuration link. This is best illustrated by the example in Figure 1.2. We can check, for example, whether all comfort cars are equipped with rain sensors. For this purpose, we simply propagate the artificially created constraint that `RainSensor` is selected. The propagation leads to the artificially source constraint that `Comfort` is selected. This means that the property that all comfort cars are equipped with rain sensors holds. This may look trivial in the example. However, in more complex examples, it can be very helpful to check whether some properties are captured correctly. Naturally, the artificially created constraints are not "really added" to the models. If the configuration link does not fulfill a required property, the variability definition is incorrect (see Nº 2). In addition, the propagation of artificially created constraints can be used

for analysis purposes to reveal under which conditions a certain feature gets selected.

4. **Feature constraint propagation automates the check whether changes in a lower-level feature model have impacts on a higher-level feature model.** If an engineer of a lower-level feature model performs some changes (e.g. he adds a new constraint), he can use the technique of feature constraint propagation to identify the implicit constraints in all higher-level feature models. If the propagated constraints before the changes differ from the propagated constraints after the changes, the local changes do not only affect the lower-level feature model itself but also a higher-level feature model. Such changes could be unacceptable for the overall product line. Note that not only the adding of new constraints could affect a higher-level feature model but also different changes like, for example, restructuring of features. This is because constraints can address the tree-structure of the feature model. We do not want to discuss this here in detail because this requires a deeper understanding of constraints, their semantics and configuration links.

The use of feature constraint propagation does not necessarily mean that the propagated constraints are actually added to the individual feature models. Naturally, only tolerable constraints should really be added to make implicit effects accessible (№ 1 above). If the propagation leads to non-acceptable constraints (№ 2 above) or some required properties are not fulfilled by the configuration link (№ 3 above), the product line specification has to be revised. Afterwards, feature constraint propagation should be used again to check whether the errors are resolved.

1.3 The Main Contributions of this Thesis

Up to now, we have motivated feature constraint propagation, explained what it is and presented its benefits. Furthermore, we mentioned that this thesis introduces the technique of feature constraint propagation. However, what does "introduce" mean in this context? It means that the technique is defined, formalized, verified, implemented and evaluated. The main contributions of this thesis are summarized in the following.

1. introduction of a new way to deal with constraints in hierarchically organized product lines (Section 1.2)
2. feature constraint propagation is rigorously defined on a conceptual level (Chapter 5)
3. feature constraint propagation is formalized and embedded into an existing formal framework (Chapter 6)
4. correctness and minimality of feature constraint propagation are proven (Chapter 6)
5. feature constraint propagation is implemented and integrated into a feature modeling tool (Chapter 7)

6. feature constraint propagation is evaluated by an experimental case study (Chapter 7)
7. feature constraint propagation is evaluated by an industrial case study at Daimler AG (Chapter 7)

In the conclusion of this thesis we revisit and refine these contributions (cf. Section 8.2).

1.4 The Term 'Constraint Propagation'

The term 'constraint propagation' is not new. It denotes a very general and efficient technique used in many constraint solvers. Constraint propagation is a form of inference and not a form of search. Values or combinations of values for some variables in a constraint satisfaction problem are excluded by actively using a constraint or a set of constraints [Bes06]. This is best illustrated by a small example. Consider a crossword-puzzle in which the name of a German city with six digits has to be found. If we now figure out that the first letter of the city is an 'A', we can automatically discard BERLIN and SOLTAU from the set of possible answers. We just propagated a constraint. The constraints "German city" and "six digits" lead to the set of all German cities with six digits as possible solutions. The adding of the constraint "the first letter is an 'A'" allows, through constraint propagation, to exclude all cities not beginning with an 'A'. Note that we did not search a solution by trial and error. Instead, we narrowed the solution space by inference, which is more efficient and natural.

The term 'constraint propagation' is also used in feature modeling, in particular, in product configuration. In this context, it means the application of constraint propagation techniques as described above to product configuration. Consider the technical feature model in Figure 1.2. The manual selection of a rain sensor has the effect that the radar cannot be selected anymore because of the excludes relationship. This can be identified by constraint propagation and an appropriate tool could automatically forbid to select the radar (e.g. by graying out the feature or deselecting it). Constraint propagation is therefore a common technique to support the user during product configuration.

The technique of feature constraint propagation is fundamentally different from constraint propagation techniques introduced in this section. It also serves to narrow a set of possible solutions (resp. configurations) but it does not evaluate or solve constraints. Instead, feature constraint propagation is a pure transformation of constraints from one variability representation into another. The result of feature constraint propagation is therefore a constraint and not a configuration or a set of configurations. Feature constraint propagation can be applied to constraints of feature models and does not consider concrete configurations. It exclusively affects the feature models. Their configurations are only affected indirectly by the propagated constraints. The terms 'constraint propagation' and 'propagation' in this thesis always mean our technique of feature constraint propagation unless otherwise stated. We only mentioned the other constraint propagation approaches at this point in order to avoid ambiguities.

1.5 Structure of this Thesis

After this introductory chapter, the thesis continues with the introduction of the required basics as variability management, feature modeling and the concept of configuration links in Chapter 2. Then, approaches and concepts related to feature constraint propagation are presented in Chapter 3. The subsequent part of the thesis contains the main contributions. First of all, a detailed problem analysis and the basic idea of feature constraint propagation is given in Chapter 4, which results in a set of requirements for the technique. Basing on these results, the technique is then rigorously defined on a conceptual level in Chapter 5. Subsequently, feature constraint propagation is formalized in Chapter 6 and correctness and minimality are proven. Chapter 7 presents the implementation and the evaluation of the technique, consisting of an experimental case study and an industrial case study. Finally, the thesis ends with Chapter 8, which contains the conclusion.

Chapter 2

Variability Management and Feature Modeling

Variability management is the key activity in product line engineering. However, what exactly means variability management? Different definitions of this term can be found in literature. According to [RA12], it comprises activities related to the identification, expression and binding of common and variable features included in the scope of a product line. During the identification of the variability, the information has to be captured somehow. This activity is called variability modeling. In general, two different ways of variability modeling can be distinguished. The first one is to integrate the variability into development models. In this approach the variability is captured by extending or annotating the development models. Admittedly, this approach has some drawbacks, as e.g. the increasing size and complexity of the development models, the variable constraints are not expressed in a clear and uniform manner, the definition of the variability is spread and the communication of the variability to some stakeholders is hindered since not all of them are able to understand complex development models. These drawbacks are tackled by the second way of variability modeling: the documentation of the variability in a first-class model – called *variability model* – and to relate this model to the development models. This thesis focuses on this approach and especially on the documentation of the variability and not on its relation to the development models.

There are different forms of variability models and variability modeling techniques. Section 2.1 gives a brief overview on some popular representatives of them. Feature modeling, as a very popular variability modeling technique, is separately introduced in Section 2.2 because it forms the base for this thesis. Subsequently, the concept of configuration links, which allows to relate different feature models, is presented in Section 2.3. Naturally, every concept does not only have advantages but reveals some challenges, too. Configuration links can contain (resp. produce) several contradictions, which are discussed in Section 2.4.

2.1 Variability Modeling Techniques

There are numerous variability modeling approaches. Most of them can be classified as decision modeling [Bur93] or feature modeling [KCH+90]. Besides these two families, other approaches exist that do not belong to one of them. Orthogonal variability modeling [PBL05], for example, is a mentionable approach because it is intuitive, well-founded and independent from the type of variability to be modeled. We therefore decided to present decision modeling (in general), orthogonal variability modeling and feature modeling in this thesis. Whereas the former two techniques are only briefly described here, feature modeling is introduced in detail in Section 2.2 since it forms the base for feature constraint propagation. Further popular variability modeling approaches, as e.g. the variability specification language [Bec03], Koalish [ASM03, ASM04], COVAMOF [SDNB04] and UML-based techniques [Gom04], are not discussed in this thesis. There are several reviews surveying and comparing these techniques [SD07, CBA09, CAB11].

In decision modeling, the variability is captured as decisions to be made, generally in *decision tables* or sometimes *decision trees*. Decision tables are an intuitive way to describe the variability of a product line as a set of decisions. An example for a decision table is depicted in Figure 2.1. It shows a small car product line taken from [BLP04]. Every row of the table represents a decision, each with the following attributes [SJ04, DGR11].

- **id:** the decision's identifier
- **question:** the question which has to be answered during product configuration
- **type:** the data type of the decision (defines the set of choices for it)
- **range:** a range that narrows the type (narrows the set of choices for the decision)
- **cardinality / constraint:** defines the number of choices to be made and additional constraints that have to be fulfilled
- **visible / relevant if:** specifies under which conditions the decision is relevant (i.e. has to be made)

Note that the attributes to be modeled and the concrete syntax vary from one decision modeling approach to another. A comparison of decision modeling approaches can be found in [SRG11]. The example product line comprises station wagons as well as convertibles. If the lights of a car are sensor controlled, a sensor has to be present (modeled by the additional constraint). The wiper of the car can but does not have to be equipped with a light and rain sensor. The decision `Roof_Sensor_Control`, which defines whether a car is equipped with an automatic roof raising mechanism, is only available for convertibles (modeled by the constraint in the last column). Naturally, the sensor controlled roof raising mechanism requires a sensor (modeled as constraint).

id	question	type	range	cardinality / constraint	visible / relevant if
Type	What kind of car?	Enum	StationWagon|Convertible	1..1	
Lights_Sensor_Control	Are the lights sensor controlled?	Boolean	true|false	1..1 IfSelected Wiper.Sensor=true	
Wiper	Which additional equipment is available?	Enum	None|Sensor	1..1	
Roof_Sensor_Control	Is the roof raising mechanism sensor controlled?	Boolean	true|false	1..1 IfSelected Wiper.Sensor=true	Type == Convertible

Table 2.1: Example for a decision table (car product line slightly adapted from [BLP04])

	variant to variant	variant to variation point	variation point to variation point
requires	the selection of a variant requires the selection of another variant	the selection of a variant requires the consideration of a variation point	the consideration of a variation point requires the consideration of another variation point
excludes	the selection of a variant excludes the selection of another variant	the selection of a variant excludes the consideration of a variation point	the consideration of a variation point excludes the consideration of another variation point

Table 2.2: Types of constraint dependencies in orthogonal variability models

In orthogonal variability modeling [PBL05] the variability is captured as variation points, variants and different dependencies between these elements. Variation points define "*[...] one or more locations at which the variation will occur*" [JGJ97]. The associated variants define the possible selections for the individual variation points. Consequently, each variation point has at least one associated variant and, vice versa, each variant is related to at least one variation point. We illustrate this by an example. Figure 2.1 depicts the car product line from the example described above as orthogonal variability model. Since the image was created with the VarMod-PRIME tool environment, which was developed within the VarMod-PRIME project [Var13], the concrete syntax differs from [PBL05]. Two types of variable dependencies between variation points (denoted by triangles) and variants (denoted by rectangles) are distinguished: optional (denoted by a dotted line) and mandatory (denoted by a solid line). A mandatory dependency states that a variant is required for a variation point (i.e. it is present if the variation point is part of a product). An optional dependency states that a variant is optional. For the variation point `Light`, a manual control is required and a sensor control is optional. Besides these two dependencies, there is an alternative choice allowing to group variants of a variation point and to specify that a determined number of them has to be selected. The number (resp. range) can be specified by an interval. In the example, the variants `Station Wagon` and `Convertible` of the variant point `Type` are combined to a group with the cardinality `[1..1]`. This means that every car of the product line is either a station wagon or a convertible. In addition, orthogonal feature models provide the possibility to relate variation points and variants by constraint dependencies. There are requires and excludes dependencies between variants and variation points. All available types are depicted and explained in Table 2.2. In the example, the selection of the variant `Station Wagon` requires the consideration of the variation points `Light` and `Wiper`. In contrary, if `Convertible` is selected, the variation point `Roof Control` also has to be considered. Convertibles are always equipped with a manual roof control (because the corresponding variant is mandatory) and can furthermore be equipped with a sensor controlled roof raising mechanism. However, this requires a rain sensor to be present, which is modeled by the requires dependency between the corresponding variants.

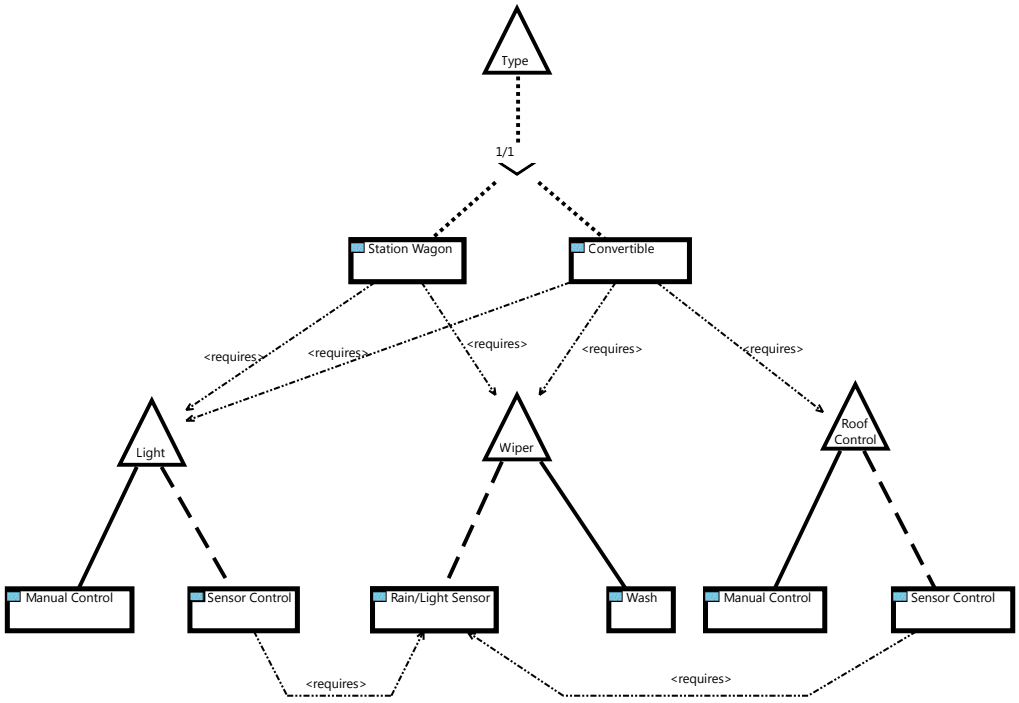


Figure 2.1: Example for an orthogonal variability model (slightly adapted from [BLP04])

2.2 Introduction to Feature Modeling

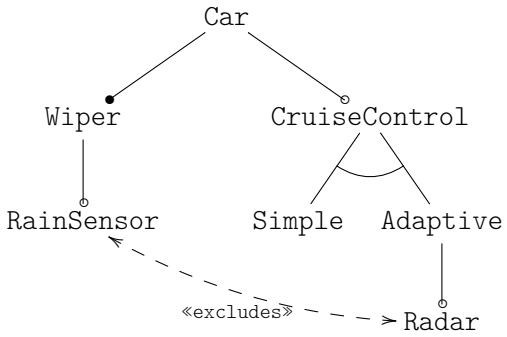
Feature modeling is a common technique for capturing a system's commonalities and variabilities in the form of *features*, which are organized hierarchically (typically as trees). There are several views on what a feature is since the term 'feature' is overloaded and very fuzzy [CHS08]. With a broad meaning, a feature can be seen as "*[...] characteristic or trait [...] that an individual product instance of a product line may or may not possess*" [Rei08, Def. 7]. The models capturing the features and their relationships are called *feature models*. Their graphical representations are typically denoted as *feature diagrams* or *feature trees* because of the hierarchical order of features. In contrast to decision modeling, hierarchy is an essential concept in feature modeling. Further differences and commonalities of feature modeling and decision modeling are discussed in [CGR+12]. A feature model represents all products of a product line and allows to derive individual products by specifying which features are present and which are not. This activity is called *product configuration* (process). It can be seen as instantiation of the feature model (similar to the instantiation of a class in object-oriented programming) or as stepwise refinement of the feature model, known as *staged configuration* [CHE04, CHE05a]. Individual steps during product configuration are often denoted as *configuration activities*.

Feature modeling was first introduced in 1990 by Kang *et al.* [KCH+90]. Since then, several extensions have been introduced. We do not want to survey feature modeling techniques here because several surveys already exist, e.g. [SHT06, Rei08, CBA09, BSRC10]. Some of these feature modeling techniques have also extended the syntax or semantics of feature models, e.g. by changing or adding notational elements. Most important extensions in the context of this thesis are the introduction of group cardinalities [RBSP02], feature cardinalities [CBUE02, CHE05a, CK05, CHE05b], feature attributes [Bed02, CBUE02] and feature inheritance [Rei08, Rei09]. In this thesis we use the term '*basic feature model*' for feature models without cardinalities, attributes and inheritance and the term '*advanced feature model*' (or just 'feature model') for feature models with all these concepts.

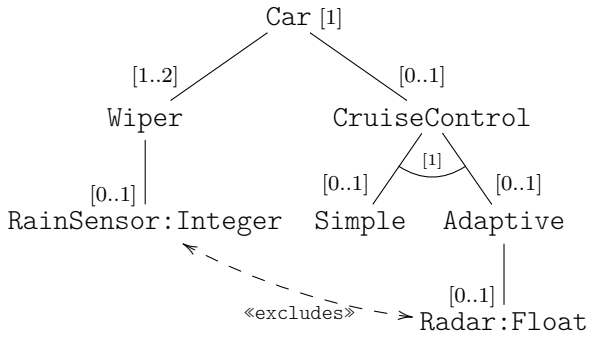
Syntax of Feature Models

The syntax of feature models is simple and intuitive. It is introduced by means of a small car product line shown in Figure 2.2.

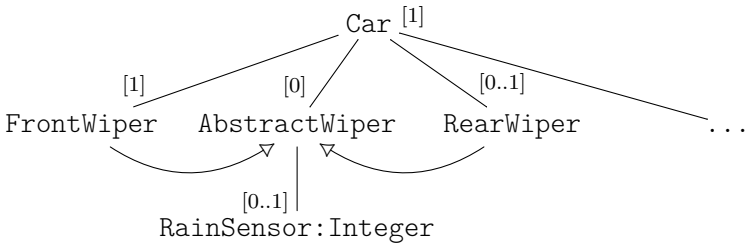
In Figure 2.2(a) the product line is modeled as basic feature model. In this product line, cars are always equipped with a wiper since the feature `Wiper` is a *mandatory feature* (denoted by the solid circle). In addition, cars can but do not have to be equipped with a cruise control – the feature `CruiseControl` is *optional* (denoted by a non-solid circle). Two types of cruise controls are allowed: a simple one and an adaptive one. The features `Simple` and `Adaptive` are in an *alternative group* (denoted by the semicircle that connects the edges from `CruiseControl` to `Simple` and from `CruiseControl` to `Adaptive`), which means that, if the feature `CruiseControl` is selected, either `Simple` or `Adaptive` must be selected. Feature models can also contain *or groups*, which are not exclusive, i.e. at least one feature has to be selected. A feature group can only contain features with the same parent.



(a) Basic feature model (slightly adapted from [Rei08])



(b) Advanced feature model (slightly adapted from [Rei08])



(c) Advanced feature model with inheritance

Figure 2.2: Examples for feature models

In the example, adaptive cruise controls can optionally be equipped with a radar and wipers with a rain sensor. In addition to parent-child relationships, cross-tree constraints can be expressed by dotted and annotated arrows between two arbitrary features. A rain sensor and a radar may not be selected both in the example.

Figure 2.2(b) shows a similar car product line modeled as advanced feature model. A cardinality (in the well-known UML notation) is assigned to every feature and every feature group. It specifies how many instances of the feature or features of the group can be included. Every basic feature model can also be expressed as equivalent advanced feature model. *Optional features* have the cardinality `[0..1]`, *mandatory features* and *alternative groups* `[1]` and *or groups* `[1..*]`. In the given example, the feature `Wiper` is a *cloned feature*, i.e. it has a cardinality with a maximum greater than `1`. A car of the product line has at least one wiper (the front wiper) but it can also have a second wiper (the rear wiper). In the case of two wipers, both *instances* can optionally be equipped with a rain sensor. Moreover, `RainSensor` as well as `Radar` are now *parameterized* (aka *attributed*) features. This means that these features cannot only be selected or deselected but data values can additionally be assigned to them. The integer value of the feature `RainSensor` configures the number of wiping modes and the float value of the feature `Radar` specifies the distance for the break assistance to be activated.

Figure 2.2(c) uses *inheritance* to model the wipers of the car. Analogous to the previous example (Figure 2.2(b)), a car can have one or two wipers and each wiper can be equipped with a rain sensor. An abstract feature, i.e. a feature with the cardinality `[0]`, `AbstractWiper` models the variability of a wiper. Inheritance is used to extend the features `FrontWiper` and `RearWiper` by all children of `AbstractWiper`. The feature `RainSensor` has thus three occurrences on instance level because the features `FrontWiper`, `RearWiper` and also `AbstractWiper`[1] have such a child. We call the occurrences of inherited features on instance level *forms* (of the feature).

Semantics of Feature Models

As already mentioned, there are a lot of feature modeling approaches with different semantics. [THSC06, SHT06, SHTB07, HST+08] introduce a comparative semantics of feature diagram languages by means of a generic construction called *free feature diagrams* to allow their direct comparison. In this thesis we focus on the concepts and semantics defined in [Rei08]. The following part briefly introduces the semantics of feature models used in this thesis informally.

The semantics of a feature model is defined as the set of its configurations (aka the set of products). A *configuration* of a feature model is, roughly spoken, given by a set of included and a set of excluded features. It is called *valid* if it fulfills the constraints defined by the feature model. This means in particular that

1. the inclusion of a feature implies the inclusion of its parent,

2. the inclusion of a feature implies the inclusion of all its mandatory children,

[1] This rain sensor of the abstract wiper can never be selected since its parent is abstract.

configuration	included features	validity
$C_{1,1}$	`Car` `Wiper`	valid

$C_{1,2}$	`Car`
	`Wiper`
	`RainSensor`
	`CruiseControl`
	`Adaptive`
		valid

$C_{1,3}$	`Car`
	`Wiper`
	`Adaptive`
		invalid (in classical feature modeling)
		
		(parent `CruiseControl` is not included)

$C_{1,4}$	`Car`	invalid (mandatory `Wiper` is not included)

$C_{1,5}$	`Car`
	`Wiper`
	`CruiseControl`
		invalid
		
		(group-constraint violated)

$C_{1,6}$	`Car`
	`Wiper`
	`RainSensor`
	`CruiseControl`
	`Adaptive`
	`Radar`
		invalid
		
		(cross-tree constraint violated)

Table 2.3: Configurations of the basic feature model in Figure 2.2(a)

3. the inclusion of a feature implies the satisfaction of all its group-constraints and

4. all cross-tree constraints are satisfied.

Otherwise it is called *invalid*. If all features are either included or excluded, a configuration is called *full*. Otherwise it is called *partial*. Note that we sometimes denote "valid configuration(s)" as "configuration(s)".

Basic Feature Models

Table 2.3 shows some examples for configurations of the basic feature model of Figure 2.2(a). Configurations $C_{1,1}$ and $C_{1,2}$ fulfill all constraints of the feature model. Configuration $C_{1,3}$ is invalid because the parent of the included feature `Adaptive` is not included (№ 1 above). A further discussion regarding this configuration can be found below. In configuration $C_{1,4}$ the mandatory child `Wiper` of the root feature is not included (№ 2 above). The feature `CruiseControl` is selected in configuration $C_{1,5}$ but no feature of the feature group is selected (№ 3 above). The cross-tree constraint that a rain sensor and a radar may not be selected both is violated in configuration $C_{1,6}$ (№ 4 above).

Advanced Feature Models

In the case of advanced feature models (see examples in Figure 2.2(b) and Figure 2.2(c)), a configuration is more complex than in the case of basic feature models. On the one hand, values can be assigned to parameterized features and, on the other hand, cloned features cannot be included or excluded themselves but they have to be instantiated and their instances can be included or excluded. The successors of cloned features are, roughly spoken, also cloned (in the example a front wiper as well as a rear wiper can be equipped with a rain sensor). This means that the set of features of the feature model and the set of features that can be addressed in configurations are not isomorphic if cloned features are used. We therefore distinguish between the *feature level* and the *instance level*. On feature level, all features of the feature model are accessible, whereas on instance level the concrete instances of cloned features and their successors are accessible in addition. This is illustrated in Table 2.4. On feature level, all features of the feature model are accessible. The individual instances of the cloned feature `Wiper` are accessible on instance level. We use the concept of *configured feature identifiers*[2] to uniquely identify a feature on instance level [Rei08] (we say a configured feature identifier *points* to a feature). For example, the configured feature identifier `<instanceName1>:Wiper` addresses the wiper with the instance name `<instanceName1>`. Although the cardinality of the feature `Wiper` is `[1..2]`, an arbitrary number of instances can be addressed. The cardinality defines only how many instances have to be present in configurations. Similar cases also apply to successors of cloned features on instance level. The feature `RainSensor` is not unique because every instance of `Wiper` has a rain sensor. This holds since cardinalities of features are evaluated relatively to their parents. In order to address a concrete rain sensor on instance level, the instance of `Wiper` has to be specified in addition (e.g. the configured feature identifier `<instanceName1>:Wiper.RainSensor` addresses the rain sensor of the wiper with the instance name `<instanceName1>`). Note that the occurrence of feature inheritance also implies that the set of features on feature level and the set of features on instance level are not isomorphic because inherited features have several forms on instance level. In the example of Figure 2.2(c), the feature `RainSensor` has three forms on instance level. Therefore, the parent of `RainSensor` has to be named in order to uniquely identify a form of `RainSensor` on instance level (e.g. the configured feature identifier `FrontWiper.RainSensor` addresses the rain sensor of the front wiper).

In this thesis we use the *Variability Specification Language* (VSL[3]) [Rei08, Rei09] to define configurations for advanced feature models. This is an easy and lightweight language that allows to address features on both levels and to define feature models as well as configurations in an intuitive manner. The syntactical structures used here are very simple. Every feature can be included by an *inclusion statement* "`[1]`", excluded by an *exclusion statement* "`[0]`" or left in *unconfigured state*. Instances

[2]Note that we sometimes use the term 'configured feature identifier' as a synonym for 'feature on instance level' in this thesis.

[3]There are two different languages called "Variability Specification Language (VSL)" [Bec03, Rei09]. In this thesis we always mean the language defined in [Rei09] unless otherwise stated.

feature level	instance level
Car	Car
Wiper	<instanceName1>:Wiper
	<instanceName2>:Wiper
	<instanceName3>:Wiper
	...
RainSensor	<instanceName1>:Wiper.RainSensor
	<instanceName2>:Wiper.RainSensor
	<instanceName3>:Wiper.RainSensor
	...
CruiseControl	CruiseControl
Simple	Simple
Adaptive	Adaptive
Radar	Radar

Table 2.4: Features on feature level and on instance level of the feature model in Figure 2.2(b)

```
Car[1];

Wiper$front,
Wiper$rear;

front:Wiper[1];
rear:Wiper[0];

front:Wiper.RainSensor[1],
front:Wiper.RainSensor=5;
```

Figure 2.3: Configuration of the advanced feature model of Figure 2.2(b) in VSL

of cloned features have to be created by *creation statements* "$" before they are configured. A summary of all syntactical structures of VSL used in this thesis can be found in Section A.1.

Figure 2.3 shows a partial configuration of the advanced feature model depicted in Figure 2.2(b) in VSL. The root feature is included by `Car[1]`. The statement `Wiper$front` creates an instance named `front` of the feature `Wiper`. Analogously, a second instance named `rear` is created. After their creation, instances can be treated like regular optional features, i.e. they can be included, excluded or left unconfigured. The front wiper is included (statement `front:Wiper[1]`) and the rear wiper is excluded (statement `rear:Wiper[0]`) in the configuration. In addition, the rain sensor of the front wiper is included (statement `front:Wiper.RainSensor[1]`) and the value 5 is assigned to it (statement `front:Wiper.RainSensor=5`).

Figure 2.4 shows a partial configuration of the example in Figure 2.2(c). The configuration describes the same product as the configuration in Figure 2.3. It is

```
Car[1];

FrontWiper[1];
RearWiper[0];

FrontWiper.RainSensor[1],
FrontWiper.RainSensor=5;
```

Figure 2.4: Configuration of the advanced feature model with inheritance of Figure 2.2(c) in VSL

mentionable that the parent of the inherited feature `RainSensor` always has to be specified.

Inclusion versus Selection Semantics

One specific of the approach of [Rei08] is the distinction between inclusion and selection semantics. *Inclusion* of a feature means that a feature appears in a configuration. For example, feature `Adaptive` is included in configurations $C_{1,2}$, $C_{1,3}$ and $C_{1,6}$ in Table 2.3. *Selection* means that a feature is included itself and all its predecessors are also included. In the example, `Adaptive` is selected in $C_{1,2}$ and $C_{1,6}$ since all its predecessors `CruiseControl` and `Car` are also included. In configuration $C_{1,3}$ the feature `Adaptive` is only included but not selected because `CruiseControl` is not included. We have stated above that the inclusion of a feature implies the inclusion of its parent. In the approach of [Rei08], this rule does not hold. This means that configuration $C_{1,3}$ of the example above is a valid configuration. The state that a feature is included but not selected can be seen as intermediate state or additional information. Configuration $C_{1,3}$ can straightforwardly be evaluated with respect to selection by neglecting all successors of not included features so that this approach is completely consistent to and compatible with classical feature modeling approaches. The additional information that `Adaptive` is included becomes important if `CruiseControl` is selected later. This leads then directly to the selection of `Adaptive`. The partial configuration $C_{1,3}$ can therefore be interpreted as "the car can only be equipped with an adaptive cruise control". The partial configuration can also be seen as specialization of the feature model (in the sense of staged configuration [CHE04, CHE05a]) because the cardinality of the feature `Adaptive` is narrowed from `[0..1]` to `[1]`.

On the contrary, *exclusion* means that a feature does not appear in a configuration (i.e. its cardinality is set to `[0]`) and *deselection* means that a feature itself or (at least) one of its successors is excluded.

The term '*inclusion semantics*' stands for the evaluation of inclusion and exclusion of features (i.e. the tree structure is neglected). Accordingly, the term '*selection semantics*' denotes the evaluation of selection and deselection of features. The tree structure of the model is taken into account. Features that are only included but not selected are omitted. The idea of this separation is introduced in [Rei08].

multi feature link	semantics as propositional logic formula
«$[F_1, F_2, ...]$ `needs` $[F_i, F_{i+1}, ...]$»	$F_1 \wedge F_2 \wedge ... \rightarrow F_i \vee F_{i+1} \vee ...$
«`excludes` $[F_1, F_2, ...]$»	$\neg(F_1 \wedge F_2 \wedge ...)$
«`alternative` $[F_1, F_2, ...]$»	$F_1 \veebar F_2 \veebar ...$

Table 2.5: Semantics of multi feature links

Feature Links and Feature Constraints

Almost every feature modeling approach provides the possibility to formulate cross-tree constraints to narrow the set of valid products. In general, two categories of *cross-tree constraints* (or short *constraints*) can be distinguished: constraints being directly integrated in the graphical representation of the feature model (here called *feature links*) and constraints being formulated in a separate constraint language (here called *feature constraints*). Note that the term 'feature constraint' is often used to denote feature links as well as feature constraints.

There are three common types of feature links: `needs`, aka `requires` (the selection of a feature requires the selection of another feature), `excludes` (two features may not be selected both) and `alternative` (exactly one of the features have to be selected). They are, in general, denoted by dotted arrows between the affected features. The feature models in Figure 2.2(a) and Figure 2.2(b) contain an `excludes` feature link each. Some techniques also allow to define "weak" versions of the mentioned feature links. These feature links do not define "hard" constraints but can be seen as proposals for modelers, which are shown in the corresponding tool during product configuration. They are neglected in this thesis. In some feature modeling approaches feature links may not (or only limited) be used with cloned features or their children. For example, Czarnecki *et al.* [CK05] claim the additional specification of a *context* for constraints in combination with cloned features. In other feature modeling approaches, feature links with more than one source and target feature are allowed. We call these feature links *multi feature links*. They are denoted as arrows with more than one source and more than one target feature. Their semantics are shown in Table 2.5. Note that $\veebar$ denotes an exclusive disjunction.

Although feature links are easy to understand and intuitive, they have a limited expressiveness. This leads to the more general concept of feature constraints. In this thesis we allow only feature constraints that can be expressed by propositional logic, according to the approach of [Rei08]. We do not consider constraints over the data types of features or the number of instances of cloned features as Czarnecki *et al.* [CK05] do.

A noteworthiness of the approach of [Rei08] is that feature links are evaluated on feature level and feature constraints are formulated and evaluated on instance level. This is important in the case of advanced feature models if the set of features on feature level is not isomorphic to the set of features on instance level (if cloned features or inheritance are used). Then, not every feature constraint can be translated into an equivalent feature link and vice versa. We illustrate the semantics of feature links and feature constraints and their correlation by examples.

Consider configurations $C_{1,1}$ and $C_{1,2}$ in Table 2.3. If these configurations are considered as full configurations (all not specified features are excluded), it is obvious that they fulfill the feature link of the feature model. If they are considered as partial (all not specified features are unconfigured), there are two viewpoints: (a) it is undefined whether the configurations are valid because the feature link could be violated by further configuration steps (e.g. the inclusion of `Radar` in configuration $C_{1,2}$) or (b) the partial configurations are valid because no constraint is violated at this time. In the former viewpoint (a), a logic that can be evaluated to three states (`true`, `false` and `undefined`) is required. According to the approach of [Rei08], the latter perspective (b) is used and every constraint can always be evaluated to `true` or `false`. Consequently, $C_{1,1}$ and $C_{1,2}$ are defined to be valid – even if they are considered as partial configurations. Since feature links are evaluated with respect to selection (and not inclusion), a partial configuration in which `RainSensor` as well as `Radar` are included but not selected is valid. This is consistent to (b) because the configuration does not violate any constraint at this time – `RainSensor` and `Radar` are not both selected.

In the case of basic feature models, every feature link can be translated into an equivalent feature constraint but not vice versa. The given feature link can be expressed by the feature constraint $\neg$(`RainSensor[+]`$\wedge$`Radar[+]`). The notation `[+]` is called *selection statement* and expresses the selection of a feature. It is important to consider that inclusion statements can also be addressed in feature constraints. For example, the feature constraint $\neg$(`RainSensor[1]` $\wedge$ `Radar[1]`) expresses that the rain sensor and the radar may not be included both. This constraint cannot be translated into an equivalent feature link because inclusion is addressed.

Now consider the advanced feature model depicted in Figure 2.2(b). The existing feature link states, similar to the previous example, that `RainSensor` and `Radar` may not be selected both. However, the parent `Wiper` of `RainSensor` is cloned and every instance of `Wiper` can separately be equipped with a rain sensor. The feature link states that no wiper may be equipped with a rain sensor if the radar is present. Consequently, the configuration in Figure 2.3 becomes invalid if the radar is added. In this case, an equivalent feature constraint cannot be formulated because feature constraints address the instance level, i.e. only individual instances of the wiper can be accessed. For example, the feature constraint $\neg$(`front:Wiper.RainSensor[+]` $\wedge$ `Radar[+]`) expresses that only the rain sensor of the front wiper and the radar are exclusive. Naturally, a wild card "`*`" in feature constraints to address an arbitrary instance of a cloned feature would be conceivable (e.g. `*:Wiper.RainSensor`). However, this would also result in two different types of constraints: constraints on feature level and constraints on instance level. In addition, mixed forms would be possible. In our approach the separation is clear: feature constraints address the instance level and feature links address the feature level.

If inheritance is used as in Figure 2.2(c), the rain sensor also has multiple occurrences on instance level. An `excludes` feature link between the features `RainSensor` and `Radar` has the same semantics as in the case with cloned features: no wiper may be equipped with a rain sensor if a radar is present. Analogously to the previous ex-

ample, feature constraints can only address individual occurrences of the rain sensor, e.g. $\neg$(`FrontWiper.RainSensor[+]` $\wedge$ `Radar[+]`).

Summary of the Characteristics of Reiser's Approach

Because this thesis is tailored to the approach of [Rei08, Rei09], most important features of this approach are summarized here.

- cardinalities, parameterized features, inheritance and multiple roots are supported
- instances of cloned features have to be created before they can be selected / deselected
- a distinction is made between inclusion and selection semantics
- feature links are evaluated with respect to selection on feature level
- feature constraints are propositional logic formulae formulated and evaluated on instance level and can address inclusion as well as selection
- constraints (i.e. feature links as well as feature constraints) can always be evaluated to `true` or `false` (even in the case of partial configurations)

A formalization of feature models is presented later in Section 6.1 in order to avoid ambiguities and misunderstandings.

2.3 The Concept of Configuration Links

Configuration links are directed relations between feature models allowing the automatic derivation of a target configuration from a given source configuration. The resulting target configuration does not necessarily have to be full but it can also be a partial configuration, which needs manual intervention. It is denoted as *configuration predefinition* or simply *preconfiguration*. The basic idea of configuration links is not to define mappings of features (of one feature model to another), as the example in Figure 1.2 may suggest, but rather to map a selection of products – so-called *product sets* – to features of another feature model. The idea of product sets and their mappings is presented in an early work of Reiser and Weber [RW05]. A product set can be related by "includes" and "excludes" links (not to be confused with feature links) to features of a feature model, which means that all products of a product set have or do not have a designated property (i.e. a feature). In the example (Figure 1.2), formula 1.1 states that all U.S. cars (this describes a product set) have a radar (this is the property that all cars of this product set have). Based on this work, configuration links were developed and introduced in [Rei08]. They are compatible with all feature modeling concepts and techniques presented in Section 2.2. This section only introduces the concept of configuration links and not their use cases as [RKW07, Rei08, RKW09] do.

A configuration link consists of a set of *configuration decisions*, which in turn consist of a *criterion* and an *effect* each. In the example in Figure 1.2, formulae 1.1

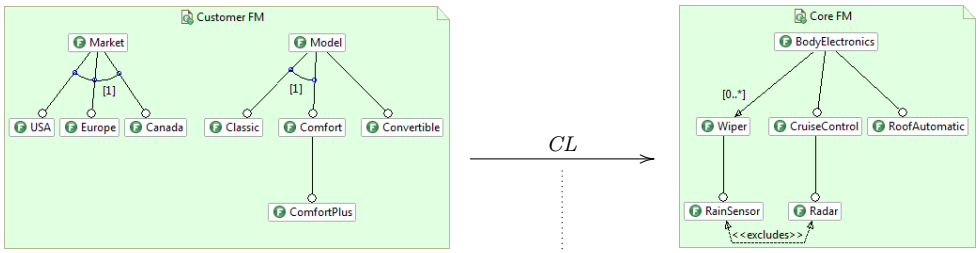
and 1.2 are configuration decisions, each denoted in the form $\langle criterion \rangle \mapsto \langle effect \rangle$. A criterion is an arbitrary feature constraint with respect to the source feature model, stating under which conditions the corresponding effect is applied. The effect is a set of so-called configuration steps with respect to the target feature model. A *configuration step* is an atomic *configuration activity* and can be

1. the restriction of a feature's cardinality,
2. the creation of a cloned feature's instance or
3. the restriction of a parameterized feature's type.

Note that configuration steps address features on instance level and not on feature level. In the example, the effect in configuration decision 1.1 selects the feature `Radar`. This means that the cardinalities of `Radar` itself and all its predecessors are narrowed to `[1]`. Exclusion statements as well as deselection statements can be used in configuration decisions to exclude a feature. A *deselection statement* `[-]` naturally states that a feature is deselected. However, if deselection statements are used in effects of configuration decisions, they are equivalent to exclusion statements. They exclude the addressed features themselves and do not affect their predecessors. More applications of deselection statements are presented later in this thesis when advanced feature constraints are discussed (cf. Section 5.2.5).

Configuration links are consciously designed to allow redundancies and contradictory configuration decisions. A configuration decision is called *redundant* if it is fully covered by another configuration decision. The motivation for allowing redundancies is that the redundant configuration decisions could have different rationales, which makes them to distinct configuration decisions. If one of these rationales drops, the corresponding configuration decision can easily be removed. The other configuration decisions arising from different rationales remain in the configuration link. More details on the benefits of this approach can be found in [RW05, Rei09]. *Contradictory configuration decisions* are two (or more) configuration decisions with contradicting effects, i.e. both effects cannot be applied at the same time. Configuration links are designed to resolve these contradictions automatically in a predefined way [Rei08, Rei09]. Thus, adding a new configuration decision can define an exceptional case but it cannot corrupt a configuration link (in the way that it cannot be applied anymore). The motivation for this approach is to keep configuration decisions independent from one another. This is a fundamental difference between configuration links and constraints. Adding a new constraint to a set of constraints can lead to a contradiction.

Figure 2.5 shows a more complicated configuration link as an illustration. This example is a more complex version of the car product line depicted in Figure 1.2, in which the customer-oriented feature model provides the additional possibility to choose a comfort plus package. Moreover, every car model can be delivered as convertible. The technical model contains the additional feature `RoofAutomatic` expressing that a car is equipped with an automatic roof raising mechanism. Note that the feature `Wiper` is cloned because a car can be equipped with more than one wiper. The additional feature constraint of the technical feature model states that cars with a roof automatic cannot be equipped with a rear wiper. The configuration



$$\texttt{true} \mapsto \texttt{Wiper\$front, front:Wiper[+]} \quad (2.1)$$

$$\texttt{USA[+]} \mapsto \texttt{CruiseControl[1], Radar[1]} \quad (2.2)$$

$$\texttt{USA[+]} \vee \texttt{Canada[+]} \mapsto \texttt{CruiseControl[1]} \quad (2.3)$$

$$\texttt{Comfort[+]} \mapsto \texttt{front:Wiper.RainSensor[1]} \quad (2.4)$$

$$\texttt{ComfortPlus[+]} \mapsto \texttt{Wiper\$rear, rear:Wiper[1], rear:Wiper.RainSensor[1]} \quad (2.5)$$

$$\texttt{Convertible[+]} \wedge \texttt{Comfort[+]} \mapsto \texttt{RoofAutomatic[1]} \quad (2.6)$$

$$\neg\texttt{Convertible[+]} \mapsto \texttt{RoofAutomatic[0]} \quad (2.7)$$

Figure 2.5: A configuration link from a customer-oriented (left) to a technical feature model (right) in a car product line (extended version of Figure 1.2)

link between the feature models consists of seven configuration decisions. Configuration decision 2.1 states that every car is equipped with a front wiper (the instance `front` of the feature `Wiper` is created and then selected). Configuration decision 2.2 defines that all cars in the U.S. are equipped with a cruise control that uses a radar allowing the vehicle to slow down when approaching an obstacle. The fact that all north American cars come with cruise controls is determined by configuration decision 2.3. This configuration decision is partly redundant to configuration decision 2.2 but it could originate from a different rationale, e.g. because of legislation. The front wipers of comfort cars are equipped with rain sensors (configuration decision 2.4) and comfort plus cars additionally come with rain sensor equipped rear wipers (configuration decision 2.5). Configuration decision 2.6 states that comfort convertibles have an automated roof raising mechanism. Naturally, closed cars cannot be equipped with a roof automatic, as configuration decision 2.7 constitutes. Configuration decisions 2.6 and 2.7 are contradictory because their effects are contradicting. However, these configuration decisions can never be applied together because their effects are disjoint (i.e. a configuration cannot fulfill both criteria at the same time).

Semantics of Configuration Links

A configuration link can be applied to an arbitrary source configuration and delivers a target configuration. In the first step of the application, the criteria of all configuration decisions are evaluated with respect to the given source configu-

ration. Then, the effects of all configuration decisions whose criteria are fulfilled by the configuration are combined and, finally, applied. For example, the selection of `Comfort` and `Convertible` enables configuration decisions 2.1, 2.4 and 2.6. The corresponding effects are combined and applied, which leads to the target configuration `BodyElectronics[1]`, `front:Wiper[1]` (by configuration decision 2.1), `front:Wiper.RainSensor[1]` (by configuration decision 2.4), and `RoofAutomatic [1]` (by configuration decision 2.6), i.e. a preconfiguration of a car equipped with a front wiper and a roof automatic. An important case arises if two contradictory configuration decisions are applied at the same time. Imagine an additional configuration decision `Convertible[+]` $\mapsto$ `rear:Wiper[0]`, which is contradictory to configuration decision 2.5, and a source configuration with the features `ComfortPlus` and `Convertible` being selected. This enables the two mentioned contradictory configuration decisions and the contradicting effects `rear:Wiper[1]` and `rear:Wiper[0]` have to be combined and applied. The resolution of this contradiction takes place within the combination step. The rule is that an exclude is prioritized over an include following the idea to be as conservative as possible. Therefore, `rear:Wiper[1]` and `rear:Wiper[0]` are combined to the configuration step `rear:Wiper[0]`, which can then be applied. In general, conflicting cardinalities are resolved by their intersection or, if it is empty, to `[0]`. The same resolution is applied for types, however, if the intersection is empty, the type does not get narrowed. This resolution of contradictions is a fundamental principle of configuration links and constitutes a drastic difference to classical constraints. Because of this difference, a configuration link cannot simply be expressed as constraint by formulating the individual configuration decisions as inclusions and combining them by conjunctions.

Formal semantics of configuration links are presented later in Section 6.2 in order to avoid ambiguities.

2.4 Contradictions in Configuration Links

The creation of configuration links is simple and intuitive: individual configuration decisions are formulated according to the requirements and then combined to a configuration link. However, their combination can lead to some hard to locate contradictions within the variability specification. In [Rei08] three types of contradictions that can arise are identified.

1. contradictions between configuration decisions for a single feature
   - narrowing the cardinality to conflicting values
   - referring to a non-created instance of a cloned feature
   - narrowing the type to conflicting values
2. contradictions resulting from parent-child relations in the target feature model
   - including a feature while excluding its parent
   - creating an instance of a cloned feature while excluding its parent

3. contradictions resulting from cardinalities or constraints
    - selecting a number of instances of a cloned feature not allowed by the cardinality
    - selecting a number of grouped features not allowed by the cardinality
    - violating a feature link
    - violating a feature constraint

Category 1 contradictions do not represent errors and are resolved by the semantics of configuration links and feature models. Contradictions of category 2 also do not contribute errors but should produce warnings by the modeling tool. The contradictions of category 3 comprise the selection of an invalid number of instances or features and the violation of constraints.

As example for such a contradiction, consider the example in Figure 2.5 and the source configuration with the features `USA`, `ComfortPlus` and `Convertible` being selected. This results in a target configuration with a front and a rear wiper, both equipped with rain sensors, a cruise control with a radar and a roof automatic. This configuration is invalid since the `excludes` feature link as well as the additional feature constraint are violated.

According to [Rei08], the tool should aid the engineer in spotting and resolving such issues. However, it is not trivial to spot these contradictions since not only the target feature model and the effects of the individual effects of the configuration decisions have to be considered but also their criteria and the source feature model. The appearance of effects which would violate a constraint or a cardinality if they are applied together is only an indication that invalid configurations can be produced. Perhaps their criteria are disjoint or the combination of their criteria describes an invalid source configuration. In these cases, the contradiction can be neglected. The technique of feature constraint propagation tackles contradictions of category 3 that lead to a violation of constraints by evaluating the variability specification and visualizing the effects of constraints of lower-level models in higher-level models.

Chapter 3

Related Work

Feature constraint propagation complements the concept of configuration links. Consequently, one part of the related work belongs to configuration links, i.e. approaches for managing complex variability by hierarchical organization or modeling the variability in different models or views and relating them. The other part of the related work contains literature that is directly related to the technique of feature constraint propagation itself. Besides a pure introduction of related approaches and techniques, this section also contains a comparison of them to this thesis.

The idea of feature link propagation along configuration links has already been presented before in [Fab08]. However, the approach is a different one. It does not base on the transformation of logical formulae but it defines rules stating under which conditions a special feature link results in a propagated feature link. Consequently, only a determined set of feature links (`needs` and `excludes`) can be propagated. General constraints or different kinds of feature links are not taken into account. The result of the propagation is always a `needs` or `excludes` feature link and not a general constraint. A drastic shortcoming of the approach is that it does not satisfy the *correctness* property (presented in Section 1.2, Figure 1.3), i.e. a valid source configuration can lead, through the application of the configuration link, to an invalid target configuration. This is because feature links cannot always be propagated but only under certain conditions and because this approach does not base on logic transformations. Further shortcomings of the approach are that advanced feature modeling concepts (cloned features, feature inheritance and parameterized features) and complex formulae in criteria of configuration decisions are not supported and the separation of selection and inclusion semantics is not completely outlined. The approach allows the propagation of feature links bottom-up (from lower-level models to higher-level models) as well as top-down (from higher-level models to lower-level models).

The idea of transforming logical formulae to accomplish feature link propagation along configuration links was first introduced in our earlier research [KRR10]. However, the concept presented in this paper is only compatible with normalized configuration links (i.e. configuration links mapping single source-side features to single target-side features, cf. Section 4.1) and does not cover inclusion but only selection semantics. In addition, only feature models containing solely optional features are supported. Advanced feature modeling concepts (cloned features, feature

inheritance and parameterized features) are not supported. This work can be seen as basis for the technique of feature constraint propagation presented in this thesis.

To the best of our knowledge, there is no further literature relating to constraints in relationship with configuration links.

This chapter contains four sections. The first two sections describe the closest related work: approaches mapping features of a feature model to different target models and dealing with implicit constraints (Section 3.1) and approaches for managing complex variability (Section 3.2), some of them with concepts and techniques similar to configuration links and feature constraint propagation. Subsequently, approaches using different "constraint propagation" techniques (cf. Section 1.4) are briefly summarized in Section 3.3. Finally, Section 3.4 shortly introduces some approaches addressing automated feature model analyses. The related work is presented in chronological order within all sections.

3.1 Feature Mapping Approaches

There are several approaches investigating the mapping of feature models to other development artifacts. In this setting, constraints of the lower-level artifacts also affect the feature model, analogous to the scenario with two feature models and a configuration link between them. This section presents approaches considering these implicit effects of lower-level artifacts.

2006

Czarnecki and Pietroszek [CP06] introduce an approach for verifying feature-based model templates. A feature-based model template consists of a feature model and an arbitrary annotated model, which is defined by a meta-model basing on the Meta-Object Facility (MOF) formalism [MOF13]. The annotations of the model refer to features of the feature model and describe under which conditions an element of the model is present. These *presence conditions* are propositional logic formulae over the features. Instantiation of a model template for a concrete feature model configuration means that a concrete model is derived from the given configuration by evaluating all presence conditions. A model template is called correct if and only if every valid configuration of the feature model leads to a correct template instance. This is similar to our definition of correctness and minimality (cf. Section 1.2). As already known from this thesis, constraints of the lower-level model can have impacts on the higher-level feature model.

For example, consider a concrete class diagram with two classes `A` and `B` and an association between them. The presence condition of the class `A` could state that this class is only present if a feature `F` is selected. An instantiation of the model template with `F` being not selected leads to a derived model without class `A`. Consequently, the association is dangling, i.e. the derived model (template instance) is not well-formed.

The paper presents a verification approach for well-formedness constraints formulated in the Object-Constraint Language (OCL) [OCL13]. OCL uses a three-valued logic and allows constraints with quantifiers and over data-types. In the first step of the approach, the *accumulated presence conditions*, which comprise hierarchical

structures within the model, are calculated. Then, the feature model is formalized as propositional logic formula and connected via the accumulated presence conditions with the well-formedness constraints to be checked. Finally, a constraint solver is used to apply the verification of the template. If an error was found, the solver gives sample valuations, for which the resulting template instance will not be well-formed. If the annotations and the annotated model are correct but the verification reveals an error, a constraint could be missing in the feature model. In this case, the authors propose to apply a formula simplifier to the formula describing the identified sample valuation and to add the resulting constraint to the feature model. This is called constraint propagation from the solution space to the problem space.

The proposed approach is very close to our idea of feature constraint propagation: constraints of a lower-level model can be propagated to a higher-level feature model. In particular, the presence conditions correspond to the reverse inclusion mapping and the accumulated presence conditions to the reverse selection mapping of feature constraint propagation, which are introduced later in this thesis (Section 5.1.2). In addition, our approach of feature constraint propagation also comprises a simplification of the resulting formulae (cf. Section 5.1.4). However, there are several differences between this approach and our approach. The target model in this approach is an arbitrary model and the mapping describes conditions for elements of the model to be present. Configuration links, in contrast, only address feature models and do not affect the structure of the target model itself but describe how to configure the target model. Moreover, [CP06] addresses only basic feature models, whereas our approach fully covers advanced concepts like cloning, parameterized features and inheritance. Configuration links can narrow the cardinalities, narrow the types or create instances of cloned features, which is more complex than just stating under which conditions something is present. However, well-formedness constraints that can be checked by [CP06] are more expressive than propositional logic formulae that can be propagated by our approach. For this purpose and because the main goal of the approach of Czarnecki and Pietroszek is to verify the model template and not to visualize the resulting constraints, they use a solver to apply the analysis. Our propagation approach is based on the transformation of propositional logic formulae and does not require a solver. It is equipped with several adapted simplification steps (restructuring steps), which are presented later in this thesis (Section 5.1.5 and Section 5.1.6), since it is tailored to the approach of [Rei08]. Thus, inclusion as well as selection semantics and constraints on feature level as well as constraints on instance level are supported. In addition, algorithms for converting constraints between these levels are contained.

Besides [CP06], there are several similar approaches investigating the same issue for different target models. We discuss them briefly in the following.

2007

Thaker *et al.* [TBKC07] consider the mapping of basic feature models to feature modules. A feature module is, roughly spoken, the implementation of a concrete feature. Feature modules can reference elements implemented in other feature modules. The effects of these implementation constraints on the feature model can be iden-

tified with the approach of [CP06]. Adding them to the feature model guarantees *safe composition*, i.e. all programs of the product line are type safe.

2008

Heidenreich [HKW08] presents a mapping approach for cardinality-based feature models to models basing on Ecore meta-models [SBPM09] and implemented this approach in the tool *FeatureMapper* [Fea13]. He discusses possible extensions to *FeatureMapper* for checking the correctness of whole software product line in [Hei09], i.e. every feature configuration shall lead to a well-formed target model. The goal is to create a generic framework that can be parameterized with language specific well-formedness rules.

2010

Parra *et al.* [PCBD10] propose a mapping approach with aspect models (basing on a given aspect meta-model) as target models. This approach explicitly addresses the identification of implicit dependencies. It contains a *left to right* and a *right to left* analysis according to a given configuration. The former analysis allows to propagate constraints from the feature model to the aspect model and the last-mentioned allows to propagate constraints in the opposite direction. They address only `requires` and `excludes` dependencies and allow only basic feature models. The overall goal is the derivation of a conflict-free composition strategy of artifacts.

2011

Alférez *et al.* [ALHM+11] introduce a mapping approach with use scenarios (i.e. use case and activity diagrams) as target models. They allow basic feature models with `requires` and `excludes` dependencies. The analysis is accomplished with a SAT solver. Based on the result, the developers may modify the feature model manually by adding additional dependencies.

2013

Gröner *et al.* [GBPG13] address process model templates as target models. They allow basic feature models as source models and only two kinds of cross-tree constraints: `includes` (aka `needs`) and `excludes` dependencies. Both models as well as the mapping between them are formalized in description logic [BCM+03] as knowledge bases and a solver is used to apply the analysis. This approach aims to identify errors, not to propagate constraints.

3.2 Approaches for Managing Complex Variability

Besides the approach of modeling a large-scale product line in a hierarchical manner used in this thesis [Rei08], there are several other approaches for dealing with complex variability. Some of them are also based on the idea of hierarchical organization, others define different views to a variability specification and still others address the configuration of products. A systematic survey of various concerns of feature diagrams and techniques to separate them can be found in [HTH13]. In this thesis we focus on approaches proposing hierarchical organization or the organization of the

variability in multiple models or views. However, we do not revisit the complete related work of configuration links, as described in [Rei08]. We are particularly interested in the possibility of defining relationships between models or views and the ability to deal with constraints, especially constraints between different models. The relationships of the presented approaches to feature constraint propagation are discussed during their description.

2007

Collaborative Product Configuration by Mendonça *et al.*

Mendonça *et al.* [MCO07, MCMdO08] present an approach for collaborative product configuration, which allows different stakeholders, each with a particular perspective on the product, to interact and negotiate configuration decisions[1]. Conflicts between configuration decisions of different stakeholders can be automatically identified and solved according to priority schemes defined by decision makers. Priority schemes can be role-based (i.e. the decision of the highest ranked decision role is prioritized) or decision-set-based (i.e. priorities are assigned to decision sets according to the importance of contained features). The major goal of this approach is not to solve conflicts automatically by constraint solving but to involve and support humans during this activity.

Separation of Concerns in Feature Modeling by Metzger *et al.*

The basic idea of Metzger *et al.* [MHP+07] is the separation of product line variability and software variability and the definition of a relation between them to disambiguate the documentation of variability and to separate concerns. In this approach the product line variability is not modeled as feature model but as orthogonal variability model (cf. Section 2.1). This model is related to a feature model, which describes the software variability, by *cross-model links* (X-links). A variant-specific X-link connects variants of the orthogonal variability model to features in the feature model. It specifies that the selection of a variant requires to the selection of all connected features and, vice versa, that the selection of a feature requires the selection of at least one connected variant. In addition to variant specific X-links, there are global X-links, stating that some features are always selected. Furthermore, a X-link can be an arbitrary Boolean formula over the variants and variation points of the orthogonal variability model and the features of the feature model.

X-links are similar to configuration links but they relate orthogonal variability models with feature models and not feature models with one another. In addition, the intention and the semantics of configuration links differ from classical constraints (cf. Section 2.3). Metzger *et al.* also present a formalization of feature diagrams, orthogonal variability models and X-links. It uses the existing generic construction of *varied feature diagrams*, which is based on free feature diagrams (cf. Section 2.2) as pivot language. Feature models, orthogonal variability models and X-links are

[1]The term 'configuration decision' means here (and in most other related approaches) the manual selection or deselection of a feature. This is not to be confused with configuration decisions in configuration links.

translated into this formal language and obtain a formal semantics as consequence. The translation of feature models has already been described in [SHTB07]. Since the abstract syntax of orthogonal feature models was only given as meta-model [PBL05] at that time, the translations of orthogonal variability models and X-links to varied feature diagrams are precisely defined.

The resulting precise semantics forms the base for the implementation of a tool allowing automated reasoning on variability. This tool can, similar to feature constraint propagation, e.g. check the realizability (i.e. check if all products of the product line are realizable) or identify all non-realizable products. However, the idea of feature constraint propagation is to make the impacts of constraints of lower-level models accessible on higher levels and not to enumerate all non-realizable products. The output of the automated reasoning technique is a set of non-realizable configurations of the higher-level model, whereas the output of feature constraint propagation are source-side constraints. Reasoning techniques solve constraints, feature constraint propagation transforms them.

2008

Multiple Product Lines for Software Supply Chains by Hartmann and Trew

Hartmann and Trew [HT08] introduce the concepts of *multiple product line feature models* and *context variability models*. A context variability model describes the commonality and the variability of the context in which a product is used (e.g. geographic regions). This model can (but does not have to) be a feature model. It is combined with a conventional feature model by a set of dependencies (n:m "requires", "excludes" and "set group cardinality" relationships) between the models. The result is then called multiple product line feature model. This model is similar to two feature models combined with a configuration link. A configuration of the context model leads to a feature model for a single context. This means that, analogous to the application of configuration links, the dependencies between the models are evaluated with respect to a configuration of the context variability model and lead to a specialized feature model. However, configuration links differ in their exact semantics (e.g. the resolution of contradictory configuration decisions). Multiple product line models can also be used within software supply chains. The idea is to generate specialized feature models from the multiple product line model. The used concepts of specialization and configuration are introduced by Czarnecki *et al.* in [CHE04, CHE05a]. Finally, different merge operations of feature models and multiple product line models are described, e.g. to combine models from different suppliers. Since merge operations are very complex, the authors presented *supplier independent feature models* [HTM09], which contain the super-set of features of all suppliers. This model is then composed (not merged) with the feature models of all suppliers (*supplier specific feature models*) to a so-called *composite supplier feature model*.

The relationship between the individual supplier specific feature models and the supplier independent feature models are defined through the already mentioned dependency relations. In this setting, a configuration of the higher-level feature model

(the supplier independent feature model) can be used to derive a fully configured product, i.e. all lower-level feature models (the supplier specific feature models) are fully configured. Configurations of the supplier specific models do not have to be created manually. This is similar to a use case of configuration links: the definition of artifact lines which are related to a core feature model (cf. Section 1.1). In addition, both approaches allow the derivation of configurations of lower-level feature models through the configuration of a higher-level feature model. However, the exact semantics of configuration links differ from this approach.

2009

Multi-view Feature-based Configuration by Hubaux *et al.*

Hubaux *et al.* [HCH09, HHSD10, HHS+11, Hub12] do not define multiple variability models but follow a different idea to manage complex variability. They introduce the *feature-based configuration workflow*, which supports collaborative product configuration. Responsibilities and rights can be defined with different views on a feature model. These views are insulated spaces in which users can safely configure the part of the feature model that is assigned to them (*concern-specific configuration views*). The concept of *multi-view feature models* is formally defined as free feature diagram (in the sense of [SHTB07]) with a multiset (i.e. a set in which elements can appear more than once) of views. Conflicts between user decisions are handled by a so-called *range fix* algorithm. The technique of *decision propagation* (aka constraint propagation as introduced in Section 1.4) is used to ensure that neither a manual configuration decision nor the application of the range fix algorithm can lead to invalid configurations. All concepts and their properties are integrated into a sound mathematical framework.

Viewpoint-oriented Variability Modeling by Mannion *et al.*

Mannion *et al.* [MSA09] propose the use of *viewpoints* on feature models for managing complex variability and capturing the needs of different stakeholders. Viewpoints are configured separately from one another and are then combined to get an overview on the product line or to create products. During the combination of different viewpoints, features with the same name are identified and conflicts, if occurring, are resolved according to *conflict resolution rules*, which are defined by the authors.

These rules have some similarities to the application of configuration links: the combination of an optional and a mandatory feature results in a mandatory feature and the combination of an optional and an abstract feature results in an abstract feature. However, this approach does not define mappings between different feature models.

A Feature Interaction Approach by Sanen *et al.*

Sanen *et al.* [STJ09a] tackle the issue of separating product line variability and software variability, according to [MHP+07]. Two distinct feature models are used to model the two kinds of variability. The presented case study is an adaption support system DyReS [TJSJ08], which allows the safe implementation of distributed

adaptions of a client-server based application. In this setting, the product line variability is defined by requirements and characteristics that are relevant for a particular end product (so-called *problem-space features*) and the software variability is given by customization strategies that can be applied to instantiate the resulting software (so-called *solution-space features*). Then, the concept of problem-solution feature interactions [STJ09b], which allows to define dependencies between problem-space and solution-space features, is generalized. In this paper, problem-solution feature interactions are defined as arbitrary *default logic* formulae over problem-space and solution-space features.

Default logic [Rei80] is a non-monotonic logic. This means that the consequence relation is not monotonic. Monotonicity means, in this context, that the adding of a formula to a theory cannot produce a reduction of its set of consequences or, roughly spoken, "learning a new fact cannot reduce the set of already known facts". In contrast to most studied logics, default logic allows *reasoning by default*, i.e. reasoning with default assumptions. These assumptions can be changed if additional facts are added. For example, if we assume that all birds with the exception of penguins can fly, we can conclude that an individual bird can fly if it is not explicitly stated to be a penguin. However, if the fact that the bird is a penguin is added, it cannot be concluded anymore that the bird can fly [Rei80]. The motivation of using default logic for the approach of [STJ09a] is to provide the possibility to work with partial information (e.g. incomplete requirements specifications) and to support the definition of defaults.

After both feature models are created and related, the problem-space feature model is configured and a constraint solver is used to generate a configuration of the solution-space feature model with an optimal combination of solution-space features. This work is focused on automatic derivation of optimal software configurations from requirement specifications, i.e. a configuration of a lower-level feature model is derived from a configuration of a higher-level feature model. This is analogous to the intention of configuration links. However, configuration links differ in their exact semantics (especially the resolution of contradictory configuration decisions). In contrast to problem-solution feature interaction, configuration links do not use default logic. Nevertheless, default values in criteria of configuration links can be simulated: the justification that a feature is selected by default can be simulated by the criterion that it is not deselected[2] (because this evaluates to `true` if the feature is either selected or unconfigured) and, vice versa, the justification that a feature is deselected by default can be simulated by the criterion that it is not selected.

Relating Requirements and Feature Configurations by Than Tun *et al.*

An approach allowing to limit configuration activities to product line level and to derive the configuration of the software level automatically is presented by Than Tun *et al.* [TBC+09]. This approach is non-prescriptive about the used variability model (i.e. feature models can but do not have to be used). The sole requirement is that the relations between the features can be expressed by a propositional logic formula. The

[2]Deselection cannot be addressed in feature constraints, as introduced in Section 2.2. This is only possible with advanced feature constraints, which are introduced later in Section 5.2.5.

approach proposes the separation of three variability models (exemplarily, feature models are used). One related to requirements (the view of requirements engineers), one related to the *problem world context* (the view of system engineers who design the system hardware) and one to specifications (the view of software engineers). These models are then connected by two X-links (see above and [MHP+07]): the requirements model is related to the problem world context model and this, in turn, is related to the specification model. Constraints over the states of the features and their attributes can be formulated for every feature model. Then the developer can configure the requirements model (and sometimes parts of the problem world context model) and one or more optimal configurations of software features are calculated. "Optimal" means in this context that the configuration is consistent to the given configurations of the higher-level models and all constraints are fulfilled. In addition, optimization goals (functions over feature attributes which do not belong to an individual feature model) can be formulated (e.g. the maximization or minimization of a special value). The idea of calculating configurations of lower-level models is analogous to the one of configuration links.

With this approach, not only configurations can be calculated but it is also possible to identify configurations of the higher-level model that do not satisfy the constraints defined by the lower-level model. Vice versa, configurations of the lower-level which are never required by the higher-level model can also be identified. This is similar to feature constraint propagation and forward feature constraint propagation (cf. Section 5.2.6). However, as already mentioned, the intention of feature constraint propagation is rather to make the resulting constraints accessible on higher levels. The tool used in this work is pure::variants [Pur12]. This tool allows to define relationships between features of different models and has an integrated auto resolver to correct configurations on the fly during product configuration.

2011

An Algebra of Product Families by Höfner *et al.*

Höfner *et al.* [HKM11] use idempotent semirings to define an algebra of product families. An idempotent semiring is a set with two binary operations (addition and multiplication) and some specific conditions. Feature models can easily be transformed into algebraic terms. This algebra allows the use of theorem provers to verify certain properties. Algebraic integration constraints (requires and excludes relationships) are used to define dependencies between features. This approach also allows to define such constraints between features of different views (resp. models). The authors present a Haskell [Has13] implementation of their approach, which is able to determine all valid products (of the whole product line) for a given set of integration constraints and to extract a view out of a product line.

Multi-Dimensional Variability Modeling by Rosenmüller *et al.*

In [RSTS11], Rosenmüller *et al.* present another approach for avoiding highly complex models. The idea of this approach is the separation of different variability dimensions in form of distinct feature models and their combination on demand. However, this approach does not explicitly address product line variability and software

variability but is generic in the definition of variability dimensions. After combining different feature models, constraints between them can be defined. The configuration process in this approach is described as stepwise specialization [TBK09] of the feature model. This process is known as staged configuration and introduced in [CHE04, CHE05a]. In addition, a textual language for multi-dimensional variability modeling called *Velvet* is introduced. This language is based on the language TVL [BCFH10] and allows to define several feature models with constraints in a textual form. Velvet is similar to VSL (cf. Section A.1).

The approach provides three mechanisms to compose variability models: inheritance, superimposition and aggregation. A feature model can inherit features and constraints from one or several other feature models. This means that all these features and constraints are merged. In the resulting feature model, additional constraints relating the inherited features can be formulated. In contrast, superimposition allows to decompose a feature model into multiple models. The aggregation composition mechanism allows to compose different instances of a product line and also instances of different product lines.

In [STSS13], the presented approach of Rosenmüller *et al.* is complemented by automated analyses.

2012

Separation of Concerns in Feature Modeling by Acher *et al.*

Another approach to separate the concerns in feature modeling and to manage highly complex variability was presented by Acher *et al.* [ACLF12]. This work introduces three complementary operators for composition and decomposition of feature models to ensure the *separation of concerns*: aggregate, merge and slice. The combination of these operators can be used to simplify a feature model or accomplish tedious and error-prone tasks as supporting multiple perspectives on a large feature model, reasoning about local properties with existing analysis techniques or modifying parts of a large feature model. The aggregate operator allows to define constraints between two feature models by creating a new feature model with a synthetic root and adding the roots of both feature models as mandatory children to this root. All constraints of the individual models are transferred into the new model. The merge operator serves to combine two feature models with similar features. Overlapping parts of the feature models are merged together, i.e. identical features (with the same name) are identified. There are several modes with different semantics available for this operator. The slice operator (primarily presented in [ACLF11]) extracts a feature model out of an existing model according to a slice criterion (a set of features to be included in the new model). It is combined with corrective explanations, an automated analysis operation on feature models (cf. Section 5.2.1), such that the sliced feature model does not contain any anomalies. The exact semantics is defined for all three operators in terms of configuration sets and feature hierarchy. This semantics describes the relation between the sets of configurations of the input and the output models and the relation between the feature hierarchies of the input and output models.

One example for the separation of concerns paradigm is the separation of product line variability and software variability (cf. [MHP+07]). Acher *et al.* show how their composition and decomposition operators can be combined to support this separation of concerns. In addition, they developed a dedicated technique that allows reasoning about the two kinds of variability in this setting. The aggregation operator is used to combine both feature models. Subsequently, all constraints of the mapping between the original models are added to the resulting feature model. The realizability property ("all products of the product line are realizable") is then formally defined in terms of feature models and the three defined operators (aggregate, merge and slice). If realizability does not hold for a model, the products that cannot be realized are identified by applying and combining the three defined operators.

Similar to feature constraint propagation, this approach also allows to identify configurations of the higher-level model which are not realizable. The differences between feature constraint propagation and such analysis techniques have already been mentioned.

Tracing Software Product Lines by Mohalik *et al.*

The approach of Mohalik *et al.* [MRM+12] is closely related to [MHP+07]. They present a precise and formal definition of a traceability relation between product line specifications (feature sets) and and their implementations (component sets). This relation expresses the implementability of a software product line. In comparison to [MHP+07], the product line specifications correspond to the product line variability, the product line implementations correspond to the software variability and traceability relations correspond to X-links. This work does not address a special kind of variability model but is based on pure set-theoretic semantics of a software product line. It is therefore applicable to several types of variability models (e.g. feature models or orthogonal variability models). Realizability and some other properties (similar to those in [MHP+07]) are then defined with respect to the traceability relation and can be checked automatically by a constraint solver.

Configuration links are similar to the traceability relations (resp. X-links), as already discussed.

The SPREBA Approach by Stoiber

Stoiber [Sto12] presents a fully integrated, compositional product line requirements modeling approach named SPREBA, which neither requires variability mapping nor some kind of orthogonal variability model. This approach is based on the ADORA requirements and architecture modeling language [GBR+00, Joo00] but can be applied to any modeling language with a concrete syntax. The SPREBA approach provides several views on the variability specification and the requirements model on different layers of abstraction. The impacts of configuration decisions (*variability binding decisions*) on product functionality and variability decisions are automatically derived and visualized. Several SAT-based automated analysis operations can be applied to the model, e.g. the satisfiability of a model can be verified. The approach uses a three-valued logic. Constraints can be evaluated to `true`, `false`

or `undecided`. The technique of constraint propagation, as introduced in [Bat05] (cf. Section 1.4), is used to calculate the impacts of configuration decisions. The application of this technique ensures that every model during incremental product configuration is consistent (i.e. it fulfills all variability constraints). Incorrect models are "repaired" immediately. This means that all constructed models are *correct by construction*. Constraints which are fulfilled by a configuration during the product configuration process become hidden, so that only constraints which are undecided (i.e. a new configuration decision could violate them) are visualized. If a configuration decision influences only a part of a constraint, the constraint is restructured such that only the undecided parts (i.e. the parts that could be violated by a new configuration decision) are visualized. In detail, bound variables are replaced by truth values (`true` and `false`) and the Quine-McCluskey algorithm [McC56] is used to minimize the constraint. Then the constraint is visualized in a graphical representation, which is similar to multi feature links (cf. Section 2.2).

This processing is related to some parts of feature constraint propagation: the minimization step (cf. Section 5.1.4) and parts of the extraction of feature links (cf. Section 5.1.6). However, the technique of feature constraint propagation presented in this thesis differs from the technique of constraint propagation presented in [Sto12]. Feature constraint propagation does not calculate the impacts of configuration decisions (variability binding decisions) but propagates constraints from one feature model into a related feature model. It is tailored to configuration links and the feature modeling concepts presented in [Rei08].

Further approaches addressing variability modeling and software reuse in software architectures, as e.g. [vO04, ASM04, HRR+11], are also related to configuration links because of the idea of "compositional variability management" (cf. Section 1.1). These approaches are not described in detail here since they are not very close to this work. The relation of [vO04] to configuration links is discussed in [Rei08].

Summary

All presented approaches for managing complex variability are summarized and categorized in the following.

- approaches with several variability models
  - [MHP+07]: individual models for product line variability and software variability
  - [HT08, HTM09]: different models (e.g. of different suppliers) are related and merged into multiple product line feature models
  - [STJ09a]: individual feature models for product line variability and software variability
  - [TBC+09]: individual models for product line variability and software variability, deriving configurations of the software automatically
  - [HKM11]: algebraic description of product lines that allows to define dependencies between different models

  - [RSTS11]: different feature models with composition and decomposition operators
  - [ACLF12]: composition and decomposition operators for the separation of concerns in feature modeling
  - [MRM+12]: traceability relation between distinct variability representations

- approaches for collaborative product configuration
  - [MCO07, MCMdO08]: collaborative product configuration and resolution of conflicting configuration decisions
  - [HCH09, HHSD10, HHS+11, Hub12]: different views on feature models and workflow for collaborative product configuration
  - [MSA09]: viewpoints on feature models and their combination for the configuration process

- integrated approaches
  - [Sto12]: fully integrated compositional product line requirements modeling approach

The closest approaches to configuration links are those providing the possibility to model the variability in several related models. From a technical perspective, the basic idea of configuration links is to allow the derivation of configurations and not to define constraints between the two models for consistency checks as in [MHP+07, TBC+09, MRM+12]. From a more conceptional point of view, a proposed use case of configuration links is to split the product line into several smaller product lines (artifact lines) and to model each artifact line with its own feature model (cf. Section 1.1). Moreover, the variability of hierarchical organized artifacts, such as component diagrams, should be organized hierarchical as well (compositional variability). Unlike this idea, most other approaches distinguish only between two or three levels of variability. Analogous to the intention of configuration links, [HT08, HTM09, STJ09a] explicitly address the automated derivation of configurations of the lower-level model as well. However, the exact semantics of all defined relations between two variability representations in these approaches differ from the one of configuration links (cf. Section 2.3). Especially the resolution of contradictory configuration decisions and the inclusion semantics are different.

The technique of feature constraint propagation is tailored to configuration links. None of the approaches described above uses configuration links. Admittedly, the approaches [MHP+07, TBC+09, ACLF12] present techniques to enumerate configurations of a higher-level model which are not realizable in lower-level models. This is similar to the goal of feature constraint propagation and, therefore, these techniques are closely related to this thesis. However, all these techniques base on constraint solving and not on the transformation of constraints from lower-level variability representations to higher-level variability representations as feature constraint propagation does. A more detailed comparison of constraint solving approaches and the

fundamentally different approach of feature constraint propagation is presented in Section 5.3.1.

In addition, parts of the integrated approach [Sto12] are very close to parts of feature constraint propagation, even though this approach and the presented technique of constraint propagation are radically different from the use of configuration links and feature constraint propagation as presented in this thesis. This concerns, in particular, the idea of restructuring and visualizing constraints.

3.3 Constraint Propagation Approaches

There are several works addressing the term 'constraint propagation' in a different manner than we do in this thesis, as already discussed in Section 1.4. Since the technique of feature constraint propagation presented in this thesis is fundamentally different from these constraint propagation techniques, we only present a brief overview on literature referring to these techniques and do not accomplish a complete literature research.

Constraint Propagation in Constraint Satisfaction Problems

Constraint propagation is a very general concept in constraint satisfaction problems [Tsa95], which appears under different names as domain filtering, pruning, consistency techniques, constraint relaxation, filtering algorithms, narrowing algorithms, constraint inference, simplification algorithms, label inference, local consistency enforcing, rules iteration and chaotic iteration [Bar01, Bes06].

1992

The use of constraint propagation for solving constraint satisfaction problems is not new. In 1992, Kumar [Kum92] surveys different algorithms for constraint satisfaction problems. This survey contains, besides constraint propagation algorithms, also backtracking algorithms and compares them with one another.

2001

Barták [Bar01] presents a good introduction to constraint propagation and a more actual survey of several constraint propagation techniques including their main ideas, their advantages, their limitations and a comparison of their pruning power. This is discussed in detail for binary constraints. The extensions of the techniques to non-binary constraints are briefly explained.

2006

Bessière [Bes06] provides a unifying formal framework for constraint propagation approaches. A lot of constraint propagation approaches are introduced and then embedded into this framework to relate the different notions. In addition, the main existing types of constraint propagation are introduced in detail.

2009

Tack [Tac09] shows how a propagation-based constraint solver can be designed. This work provides a deeper insight into the use of constraint propagation techniques in constraint solvers.

Constraint Propagation in Feature Modeling

Constraint propagation in feature modeling is also known as decision propagation or choice propagation. It is the application of constraint propagation techniques known from constraint solving (as described above) to the product configuration process.

2005

Batory [Bat05] connects feature models, grammars and propositional formulae and thus allows to define arbitrary propositional constraints over the features of a model. Moreover, he first proposes the use of efficient *logic truth maintenance systems* (a method for knowledge representation and manipulation, [FK93]) together with SAT solvers to debug feature models. In order to simplify the product configuration process, Batory proposes the use of Boolean constraint propagation to calculate the implications of a configuration decision, so that the construction of invalid products is not possible.

Czarnecki and Kim [CK05] use the Object-Constraint Language (OCL) [OCL13] to formulate constraints on cardinality-based feature models [CHE05b]. These constraints can also address values of data-types of the features and their cardinalities. The technique of constraint propagation is used to calculate the effects of a configuration choice and to infer the selection or deselection of the corresponding features automatically. This technique is integrated in the tool prototype and accomplished by a BDD solver.

2007

Benavides [Ben07] presents the FAMA framework [BSTRC07], which allows to automate the analysis of feature models. It is based on a formal semantics defined in Z, a standard specification language [ZIS, Spi89]. This semantics comprises software product lines as well as the analysis operations. It is compatible with parameterized features and supports multiple constraint solvers to accomplish feature model analyses. In addition, the tool automatically propagates decisions avoiding incorrect product configurations. This constraint propagation also includes data types of features. For example, if only products with a price lower than a determined value are considered, the propagation could automatically select some features and deselect some other features.

2008

Hemakumar [Hem08] investigates a method to identify contradictions in feature models. Contradiction freedom is a much stronger property than satisfiability. It means that every possible submodel of a feature model is satisfiable (i.e. has at least one product). A submodel is a specialization of the original model that arises when

a feature is selected or deselected in the original model and the consequences are propagated (cf. [CHE04]). Different kinds of constraint propagation algorithms are used to identify contradictions during runtime (i.e. during product configuration): BDD constraint propagation and Boolean constraint propagation. In contrast to BDD constraint propagation, Boolean constraint propagation is incomplete (i.e. it cannot infer all facts). Hence, if both constraint propagation algorithms produce different outputs, a contradiction is found. However, the focus of the paper is to investigate the identification of contradictions by static analysis.

Elfaki *et al.* [EPAH08] present a knowledge based method to validate feature models. A knowledge base can be seen as information repository containing a set of data in the form of rules that describe the knowledge. It allows the application of automated reasoning techniques. In the presented approach, the variability with all constraints is represented as rules in a knowledge base. Several analysis operations can be performed. Among others, the propagation of configuration decisions is supported.

2012

As already discussed in Section 3.2, the approaches [Hub12] and [Sto12] also use constraint propagation to calculate the effects of configuration decisions.

3.4 Automated Feature Model Analysis Approaches

Automated feature model analysis techniques can be defined as *computer-aided extraction of information from feature models* [BSRC10], e.g. checking satisfiability, enumerating all products, detecting anomalies. Most analysis approaches use off-the-shelf constraint solvers to automatically analyze the feature model and calculate the output of the operation. This means that analysis operations are based on constraint solving and not, as feature constraint propagation, on the transformation and restructuring of constraints. Nevertheless, feature model analysis approaches are related to feature constraint propagation because of two reasons: (1) some of them can be used as an alternative to feature constraint propagation to identify configurations of the higher-level model which lead to valid configurations of the lower-level model (cf. Section 5.3.1) and (2) some of them can be combined with feature constraint propagation to complement the technique (cf. Section 5.2.1). We only present a brief overview on automated feature model analyses here.

2010

Benavides *et al.* [BSRC10] present an extensive and detailed review on the automated analysis of feature models. This review contains articles that were published between 1990 and 2009 and that propose any analysis operation on standard feature models, the automation of any feature model analysis or performance studies of analysis operations. The first part of the survey summarizes proposed analysis operations on feature models. The next part addresses the automated support. The approaches are classified into four different groups: (1) propositional logic based approaches, (2) constraint programming based approaches, (3) description logic based approaches

and (4) other approaches (approaches that are not classified in the former groups). In propositional logic based approaches the feature model with all constraints is translated into a propositional logic formula and a SAT solver or a BDD solver is used to accomplish the analysis operation. Similarly, in constraint programming based approaches [BTRC05b, BTRC05a, JK07, Ben07] the feature model with all its constraints is translated into a constraint satisfaction problem (CSP) and a CSP solver is used for the analysis. Unlike propositional logic solvers, CSP solvers can also deal with numerical values such as integers and intervals and not only with binary values. As one might expect, description logic based approaches use description logic reasoners to perform the analysis. Description logics are a family of knowledge representation languages [BCM+03]. They are more expressive than propositional logic and allow efficient reasoning within knowledge domains. Finally, the authors discuss challenges to be addressed in the future, e.g. the formal description of the analysis operations, advanced support for parameterized features, further studies about computational complexity of the analysis operations and the development of standard benchmarks for analysis operations. Some literature that is presented in the following addresses some of these challenges. A more detailed version of the literature review can be found in the technical report [BSRC09].

2011

Pohl *et al.* [PLP11] describe a benchmark test regarding nine up to date high-performance constraint solvers for the same automated analysis operations. They investigated three SAT, three BDD and three CSP solvers and the analysis operations *valid feature model* (checks if the feature model represents at least one product), *number of products* (calculates the number of represented products), *variability* (calculates the variability factor [BTRC05a]) and *all products* (enumerates all products of a feature model). The study comprises the analysis of 90 feature models of different sizes. The conclusions are that BDD solvers achieve the best results in most cases. CSP and SAT solvers produced comparative low standard deviations and are identified as a good choice for smaller models.

2012

Michel *et al.* [MHGH12] present the first approach that proposes the use of SMT[3] solvers for configuration problems in feature modeling. Examples for configuration problems are checking if a given configuration fulfills some constraints, the creation of a configuration that fulfills a given constraint or the enumeration of all valid configurations. There are also multi-step configuration problems, which take a sequence of intermediate configurations into account [WDSB09]. Michel *et al.* argue that other solvers have different shortcomings in comparison to SMT solvers: the input languages of SAT solvers are not expressive enough for many kinds of configuration problems and CSP solvers lack of efficiency.

The approach of Mohalik *et al.* [MRM+12] has already been described in Section 3.2 because their traceability relation is closer related to this thesis. At this point,

[3] A satisfiable modulo theories (SMT) problem is special form of a constraint satisfaction problem. It is based on classical first-order predicate logic with equality.

we focus on the constraint solver of their choice. The analysis problems are encoded as quantified Boolean formulae (propositional logic formulae with existential and universal quantifiers) and then solved with a QSAT solver. The motivation for a QSAT solver is that it is quite efficient (especially in the accomplished case studies). A tool prototype named SPLANE was implemented and different case studies were accomplished with their QSAT solver and, for comparison, also with a SAT solver. The QSAT solver was much faster than the SAT solver.

Segura *et al.* [SGB+12] present a framework named BeTTy for benchmarking and testing of feature model analysis tools. BeTTy is able to detect faults in feature model analysis tools automatically. It is equipped with a test data generator (for feature models with or without parameterized features and their sets of products) [SHBRC11], which is configurable by the user so that various models, e.g. models that are computationally hard to analyze, can be generated. In addition to the input data for the analysis tool, the framework also generates the expected outputs for a number of analysis operations making the detection of faults in analysis tools straightforward. Besides functional testing of feature model analysis tools, BeTTy also provides the possibility to compare their performance.

2013

Schröter *et al.* [STSS13] investigate the analysis of *dependent feature models*. A dependent feature model is a feature model that may contain dependencies to other (dependent) feature models. The authors define how a dependent feature model can be converted into a propositional logic formula in order to apply existing analysis operations. In addition, they discuss whether the analysis operations produce correct results in the context of dependent feature models. The approach was implemented in the prototype VeAnalyzer [VeA13].

Chapter 4

Feature Constraint Propagation: Problem Description and Basic Approach

Before introducing the technique of feature constraint propagation in detail (Chapter 5), we present its basic idea and difficulties by several examples. Feature constraint propagation has to be designed such that it is compatible with all concepts of advanced feature models and arbitrary configuration links according to [Rei08] and that it is possible to propagate arbitrary constraints. With the term 'constraints' we mean feature links as well as feature constraints as defined in Section 2.2.

This chapter starts with very small and easy examples, which do not contain advanced concepts. The examples are then extended in a gradual manner until all advanced concepts are covered. The intention of this strategy is to illustrate how different feature modeling concepts complicate feature constraint propagation and to present first ideas how to deal with these challenges. Moreover, this chapter reflects how the technique of feature constraint propagation was developed. We started with very simple examples, defined a preliminary propagation algorithm, extended the examples and realized that the defined propagation became inappropriate. Consequently, we adapted the technique and continued with extending our examples. The development of feature constraint propagation required multiple iterations. Based on the examples in this chapter, properties referring to technical details of feature constraint propagation are derived. We call them "requirements" for the technique since feature constraint propagation has to meet them in order to be compatible with all feature modeling concepts of [Rei08]. All requirements are revisited in Chapter 5 when the technique is introduced in detail.

The main contributions of this chapter are (1) to communicate the arising challenges of defining feature constraint propagation to the reader of this thesis and (2) to identify necessary requirements for feature constraint propagation.

Section 4.1 contains the problem description of feature constraint propagation with basic feature models. Cloned features are added in Section 4.2 and it is shown how they complicate feature constraint propagation. Section 4.3 describes how fea-

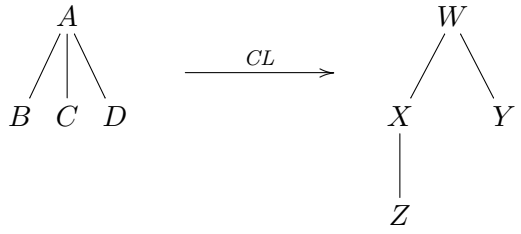


Figure 4.1: A configuration link between two basic feature models

ture inheritance and Section 4.4 how parameterized features affect the technique. Finally, a conclusion is given in Section 4.5.

4.1 Propagation with Basic Feature Models

As a first step towards feature constraint propagation, consider the example in Figure 1.2 again. Both basic feature models contain only optional features. The feature link of the technical model can be expressed by the logical formula $\neg(\texttt{RainSensor[+]} \wedge \texttt{Radar[+]})$. Note that the example's configuration link is a 1:1 mapping of features (this is called *normalized configuration link*) without redundancies or contradictions and merely respects selection (i.e. it contains only selection and no inclusion statements). We can directly see that `Radar` gets selected by the configuration link if `USA` is selected and, likewise, `RainSensor` gets selected if `Comfort` is selected. In order to propagate the given feature link, we simply have to replace variables (`Radar` by `USA` and `RainSensor` by `Comfort`) in the logical formula $\neg(\texttt{RainSensor[+]} \wedge \texttt{Radar[+]})$ and obtain the transformed formula $\neg(\texttt{Comfort[+]} \wedge \texttt{USA[+]})$. This formula can obviously be expressed by the source-side feature link «`Comfort excludes USA`». By this small example, we can see the basic idea of feature constraint propagation: we map target-side features to source-side features and replace them in formulae describing target-side constraints. The mapping of a target-side feature states under which conditions this feature gets selected in terms of source-side features.

In order to show which special cases can occur when dealing with arbitrary basic feature models, constraints and configuration links and how they affect feature constraint propagation, consider the example in Figure 4.1. Assume all features (including the roots) to be optional and the existence of a feature link «`X needs Y`».

Normalized Configuration Links

$CL_{1.1}$
$\texttt{B[+]} \mapsto \texttt{X[+]}$
$\texttt{C[+]} \mapsto \texttt{X[+]}$
$\texttt{D[+]} \mapsto \texttt{Y[+]}$

Even when dealing with normalized configuration links, more complex cases than in the example of Figure 1.2 can appear. In the first step, we take distinct configuration decisions with the same effect, as occurring in $CL_{1.1}$ into account. In order to propagate the feature link «X needs Y», we have to pay special attention to feature X because there are two configuration decisions selecting X. If one of these configuration decisions is active, X gets selected by the configuration link. Y obviously gets selected if D is selected. Because of the feature link, we know that the selection of X implies the selection of Y. This means that the selection of B or C also implies the selection of D, which can be expressed by feature links «B needs D» and «C needs D». We can see that the propagation of a target-side feature link can lead to multiple source-side feature links. Therefore, feature constraint propagation is not just a 1:1 mapping of feature links.

$CL_{1.2}$
B[+] $\mapsto$ X[+]
C[+] $\mapsto$ X[+]
D[+] $\mapsto$ Y[+]
C[+] $\mapsto$ Y[−]

Now we face contradictory configuration decisions within a configuration link by adding an additional configuration decision to the configuration link from above leading to $CL_{1.2}$. The third and the fourth configuration decision are *contradictory* because their effects cannot be applied at the same time (cf. Section 2.3): Y cannot be selected and deselected at once. We have already mentioned that an exclude of a configuration activity overrides an include. This means that the selection of C leads to the deselection of Y, regardless whether D is selected or not. In reverse, this means that Y gets selected if D is selected and C is not selected. By replacing variables of the formula X[+] $\rightarrow$ Y[+] (expressing our feature link), we obtain (B[+] $\vee$ C[+]) $\rightarrow$ (D[+] $\wedge \neg$C[+]). At this point, we could stop the propagation and present this formula as resulting feature constraint. However, this formula is hard to understand and should be processed further. We can convert this formula to the logical equivalent formula (B[+] $\rightarrow$ D[+]) $\wedge \neg$C[+], which can be expressed by the source-side feature link «B needs D» and the additional feature constraint $\neg$C[+]. This representation is easy to understand since we can visualize the feature link graphically in the source feature model and the additional feature constraint is very short. We think that feature links are in general easier to understand than feature constraints because they are visualized graphically. Therefore, our attempt is to express as many parts as possible of the propagated constraints as feature links and only the remaining parts as feature constraints. Consequently, feature constraint propagation needs the functionality to extract feature links out of constraints. We capture this in the following requirement.

Requirement Req-1: *Feature constraint propagation has to provide techniques for extracting feature links out of constraints.*

$CL_{1.3}$
B[+] ↦ Z[+]
C[+] ↦ Y[+]

Let us proceed to configuration link $CL_{1.3}$. At first sight, this configuration link looks quite trivial. It does not contain redundancies or contradictions in the effects and is a 1:1 mapping of source-side to target-side features. However, when we try to propagate our feature link «X needs Y», a new aspect arises. To answer the question whether a feature is selected, we have so far looked at all configuration decisions which directly select the corresponding feature. Configuration link $CL_{1.3}$ does not contain such configuration decisions. Nevertheless, the first configuration decision B[+] ↦ Z[+] affects feature X since the selection of Z implies the selection of all predecessors (including X) of Z. Hence, X gets selected if B is selected. In addition, we can directly see that Y gets selected if C is selected. The propagation leads to the feature link «B needs C». Moreover, we figured out that features can get selected through the application of the configuration link even if they are not explicitly contained in the effects of any configuration decision. This is due to the fact that selection statements in effects of configuration decisions have implicit effects. Feature constraint propagation has to respect all these effects. We formulate this as further requirement for the technique.

Requirement Req-2: *Feature constraint propagation has to respect implicit effects arising from selection statements in effects of configuration decisions.*

$CL_{1.4}$
B[+] ↦ Z[+]
C[+] ↦ Y[+]
C[+] ↦ X[-]

Configuration link $CL_{1.4}$ is equal to configuration link $CL_{1.3}$ with an additional configuration decision that deselects X. This configuration link contains two contradictory configuration decisions with respect to X. However, this is not obvious since only one of these configuration decisions explicitly affects X. A closer look reveals that X gets selected if B is selected and C is not selected. The selection of Y follows, analogously to the previous example, from the selection of C. This means that the exclusive selection of B leads to the selection of X and the target-side feature link is violated. When selecting C in addition, Y gets selected and, furthermore, the last configuration decision implies the deselection of X. Both features X and Y get never selected through the application of the configuration link at the same time. Nevertheless, we derive the propagated feature link «B needs C» since the selection of C leads to the deselection of X. This example shows that feature constraint propagation also has to deal with implicit contradictions in configuration links. We do not formulate a new requirement for this case since it is already covered by the previous requirement.

Non-Normalized Configuration Links (without Inclusion Statements)

$CL_{1.5}$
$\neg$B[+] $\wedge$ C[+] $\mapsto$ Y[+]
D[+] $\vee$ C[+] $\mapsto$ X[+]
C[+] $\vee \neg$D[+] $\mapsto$ Y[-]

Now we continue with the non-normalized configuration link $CL_{1.5}$. This configuration link contains three non-atomic criteria. We can directly see that the fulfillment of the criterion D[+] $\vee$ C[+] of the second configuration decision implies the selection of X. In this case, we know, by feature link «X needs Y», that Y also has to be selected. This is achieved if the criterion $\neg$B[+] $\wedge$ C[+] of the first configuration decision is fulfilled and the criterion C[+] $\vee \neg$D[+] of the third configuration decision is not fulfilled. We obtain the source-side constraint (D[+] $\vee$ C[+]) $\rightarrow$ (($\neg$B[+] $\wedge$ C[+]) $\wedge \neg$(C[+] $\vee \neg$D[+])). This is logically equivalent to $\neg$C[+] $\wedge \neg$D[+] and can be expressed by two separate feature constraints $\neg$C[+] and $\neg$D[+]. Despite the simplicity of our example, the formula resulting of the replacement is quite complex and hard to understand. Only after simplification, this formula is appropriate for presentation and can be expressed by two very short feature constraints. The complexity and length of formulae grow even more when dealing with larger examples. We follow that feature constraint propagation has to accomplish a minimization of resulting formulae, as captured in the next requirement.

Requirement Req-3: *Feature constraint propagation has to minimize resulting formulae and convert them into an appropriate form.*

In our examples, we only consider configuration decisions with atomic effects, although, they can contain combined effects (cf. Section 2.3). However, a configuration decision with a combined effect can easily be expressed by multiple configuration decisions with atomic effects having the same criteria. Therefore, we neglect this case in this section.

Configuration Links with Inclusion Statements

$CL_{1.6}$
B[+] $\mapsto$ X[+]
D[1] $\mapsto$ Y[+]

Up to now, we have only considered configuration links with only selection statements (in criteria as well as in effects). Now we take inclusion statements into account. Consider configuration link $CL_{1.6}$, which has an inclusion statement in a criterion. We can directly see that the selection of B leads to the selection of X and that the inclusion of D leads to the selection of Y. Therefore, the target-side feature link «X needs Y» entails the propagated constraint B[+] $\rightarrow$ D[1] (the selection of B implies the inclusion of D). Note that this constraint cannot be expressed directly as feature link since it contains an inclusion statement and feature links are always evaluated with respect to selection. Let us have a closer look on this constraint

and replace the selection statement by equivalent inclusion statements. We can see in the source feature model that B is selected if and only if A and B are included. This leads to the logical formula $(\texttt{A[1]} \wedge \texttt{B[1]}) \rightarrow \texttt{D[1]}$. If the premise $\texttt{A[1]} \wedge \texttt{B[1]}$ is fulfilled, then the inclusion of D is equivalent to the selection of D since A, the parent of D, is selected by premise. Therefore, we can convert this formula into the equivalent constraint $\texttt{B[+]} \rightarrow \texttt{D[+]}$, which can be expressed by the feature link «B needs D». This shows that feature constraint propagation has to replace inclusion statements by selection statements in order to extract feature links. We claim this fact in the next requirement.

Requirement Req-4: *Feature constraint propagation has to convert propagated inclusion statements into selection statements respecting the feature model's tree structure.*

$CL_{1.7}$
$\texttt{B[+]} \mapsto \texttt{X[+]}$
$\texttt{D[+]} \mapsto \texttt{Y[1]}$

In the next example (configuration link $CL_{1.7}$), we consider inclusion statements in effects of configuration decisions. In order to propagate the target-side feature link «X needs Y», we have to figure out under which conditions X and Y are selected. For this purpose, we have so far looked for selection statements for the corresponding features. Obviously, X gets selected if B is selected. However, there is no effect containing a selection statement for Y – neither explicit nor implicit. At first sight, it looks like Y cannot get selected through the application of the configuration link. This would imply that X may not be selected by the configuration link and, therefore, the constraint that source-side feature B is dead could be propagated. However, a closer look to the example shows that this is not the correct result. In fact, selecting X and Y leads to a target configuration with both features X and Y being selected. This is caused by the fact that the selection of X implicitly implies the selection of W, the parent of Y. Then, the inclusion of Y by the second configuration decision leads to its selection. Therefore, the result of the propagation is again the feature link «B needs D». As we can see, effects of configuration decisions affect each other. Only if both effects occur in our example, Y gets selected. Consequently, feature constraint propagation always has to take inclusion as well as selection statements for every feature into account. It is not sufficient to regard particular selection statements for the propagation of feature links or selection statements in feature constraints. This is captured in the following requirement, which can be seen as specialization of or a different view on Requirement Req-2.

Requirement Req-5: *Feature constraint propagation always has to consider inclusion as well as selection statements in effects of configuration decisions – even when propagating feature links or selection statements in feature constraints.*

$CL_{1.8}$
B[+] ↦ X[1]
D[+] ↦ Y[1]

Now consider configuration link $CL_{1.8}$. We can directly see that the application of this configuration link can only include – but not select – X and Y. In every derived target configuration the feature W, the parent of X and Y, is unconfigured (i.e. neither included nor excluded). Because X never gets selected through the application of the configuration link, the target-side feature link «X needs Y» is always fulfilled. Consequently, its propagation delivers true (i.e. every source configuration leads to a valid target configuration). This might seem counterintuitive because the manual selection of the unconfigured feature W would directly lead to a conflict (i.e. the feature link would be violated and the configuration would become invalid). However, the alternative to propagate the feature link «B needs D» in this example would violate the minimality property. This would mean that there are invalid source configurations (feature B selected and feature D deselected) leading to valid target configurations as already shown above.

In order to achieve the minimality of feature constraint propagation, we decided to completely omit manual configuration, i.e. the fact that a derived target configuration can become invalid by manual configuration does not matter. Every configuration of a feature model with constraints (differing from true) can be made invalid by manual configuration and, vice versa, every configuration of a feature model with constraints (differing from false) can be made valid by manual configuration. From this point of view, arbitrary manual configuration cannot be involved into feature constraint propagation. However, we could involve manual configuration of unconfigured features. This seems to be a conceivable alternative to our approach on the first sight. The propagation would then propagate stronger constraints, such that the application of the configuration link to source configurations fulfilling the propagated constraints always leads to valid target configurations that cannot become invalid by configuring unconfigured features. However, this would mean that the propagation would not be very meaningful anymore. From a conceptual point of view, the fulfillment of individual constraints has to be ensured at a certain phase of engineering. For example, some technical dependencies have to be ensured when the technical feature model is configured and do not play a role on other levels. These are the dependencies of features whose states are not derivable by a source configuration (i.e. all features which are not contained in an effect of any configuration decision). Why should feature constraint propagation visualize these constraints on other levels of abstraction if they do not affect them? Imagine an arbitrary configuration link in the example from above that do not affect the target features X or Y. Every derived target configuration could be made invalid by selecting the unconfigured feature X and deselecting the unconfigured feature Y. Hence, the propagation would deliver false, which is not intuitive. The fulfillment of the target-side feature link should be ensured when the target configuration is completed. It does not affect the source feature model. For this reason, we decided to make a closed world assumption and to completely neglect manual configuration activities in feature constraint propagation. Manual configuration should rather be

improved by good tool support and well-known automated analysis techniques than being involved in feature constraint propagation.

Inclusion Statements in Constraints

As already mentioned, feature constraints can contain inclusion as well as selection statements. Up to now, we have only considered the case without inclusion statements. Let us now face a target-side feature constraint `X[1]` $\rightarrow$ `Y[1]`, which contains inclusion statements. If we propagate this constraint along configuration link $CL_{1.8}$, we derive the source-side feature link «`B needs D`». We can see that even feature constraints (with only inclusion statements) can be propagated to feature links. This example does not establish new requirements for the technique of feature constraint propagation but emphasizes the need for considering inclusion statements in effects (see Requirement Req-5). Otherwise, constraints with inclusion statements could not be propagated.

We have analyzed different kinds of configuration links and constraints so far. Now, in the remainder of this and the coming sections, we continue with analyzing different kinds of feature models. We start by looking at basic feature models with abstract and mandatory features and then proceed to advanced feature models with cloned features, feature inheritance and parameterized features.

Abstract and Mandatory Features

$CL_{1.9}$
`B[+]` $\mapsto$ `Z[1]`
`D[+]` $\mapsto$ `Y[+]`

Consider again the example in Figure 4.1 but assume feature `X` to be a mandatory feature. In order to propagate a feature link «`Y needs Z`» along configuration link $CL_{1.9}$, we have to pay special attention to mandatory feature `X`. We can directly see that `Y` gets selected if `D` is selected and that `Z` gets included if `B` is selected. However, we are interested in the selection of `Z` and not in its inclusion. As already mentioned, we also have to consider implicit effects of configuration decisions. So we look at the parent `X` of `Z`. This feature is not affected by the configuration link. However, `X` is a mandatory feature and, therefore, always included. This effect is not evoked by the configuration link but by the feature's cardinality. Thus, we know that the selection of `B` implies the inclusion of `Z` and `X`. Furthermore, we know that the selection of `W`, the parent of `X`, is an implicit effect of the second configuration decision if `D` is selected in the source-side model. Consequently, `Z` gets selected if both `B` and `D` are selected. This means that the propagation results in the feature link «`D needs B`». As we can see, feature constraint propagation has to treat mandatory features in a special way.

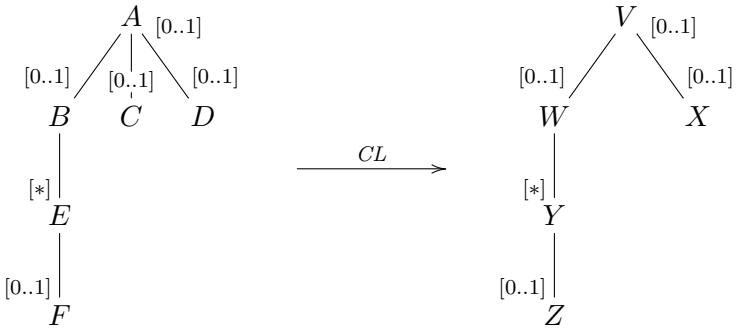


Figure 4.2: A configuration link between two feature models with cloned features

When dealing with abstract features, analogous challenges arise. Abstract features are always excluded, which also has to be respected by feature constraint propagation. Consequently, we formulate the following requirement.

Requirement Req-6: *Feature constraint propagation has to deal with the facts that mandatory features are always included and abstract features are always excluded.*

4.2 Propagation with Cloned Features

In order to illustrate the challenges arising when dealing with cloned features, we proceed with a new example. It is depicted in Figure 4.2. Both feature models contain one cloned feature each. We only consider the case with unbounded cardinalities (denoted by `[*]`) here. All cloned features with bounded cardinalities (e.g. `[0..2]`, `[1..2]`, `[42]`) can be handled analogously because they imply exactly the same special cases regarding feature constraint propagation. This holds because cardinalities only limit the number of instances but not their names. In other words, a cardinality of `[2]` means that every configuration has to contain exactly two instances of the corresponding feature, however, there are – theoretically – unlimited different configurations because the instance names are arbitrary (instances [`i1` and `i2`], [`i3` and `i4`], etc.). Feature links can only address the existence of any instance and feature constraints can only address individual instances by name. Note that feature constraint propagation does not ensure that a valid count of instances exists (neither on source nor on target size), as already mentioned. It only propagates constraints formulated as feature constraints and feature links.

First of all, we take cloned features in effects of configuration decisions into account. Subsequently, we address cloned features in criteria. Consider the target-side feature constraint $\neg$(`y1:Y[+]` $\wedge$ `X[+]`), which states that the instance with the name `y1` of feature `Y` and the optional feature `X` may not be selected both.

$CL_{2.1}$
B[+] ↦ y1:Y[+]
D[+] ↦ X[+]

Let us start with configuration link $CL_{2.1}$. It seems obvious that the propagation of the target-side constraint ¬(y1:Y[+] ∧ X[+]) results in feature link «B excludes D» since the selection of B evokes the effect y1:Y[+] and the selection of D leads to the selection of X. However, this is not the correct result. When dealing with cloned features, we also have to consider instance creation. Operator [1] in effects means that the cardinality of a feature is narrowed to [1]. This is a necessary and sufficient condition for optional features to be included, however, for cloned features, this condition is only necessary but not sufficient. Instances of cloned features are included if their cardinality is set to [1] and the corresponding instance exists (cf. Section 2.2). Naturally, the operator [+] means that the operator [1] is applied to a feature and all its predecessors. Since configuration link $CL_{2.1}$ does not create the instance y1 of Y, y1 never gets included (resp. selected) by this link. Therefore, the result of the propagation is true. Analogously to the example above ($CL_{1.8}$), manual refinement of the target configuration (the creation of instance y1 of feature Y) could make it invalid (y1:Y would get selected and, consequently, the constraint ¬(y1:Y[+] ∧ X[+]) would be violated). As already discussed above, manual configuration is neglected by feature constraint propagation.

$CL_{2.2}$
B[+] ↦ Y$y1, y1:Y[+]
D[+] ↦ X[+]

In practice, instance creation statements occur in most cases combined with inclusion (resp. selection) statements, as in $CL_{2.2}$. Then the propagation result is, as supposed above, the feature link «B excludes D».

Note that the semantics of the operators [1] and [+] differ, depending on whether they occur in feature constraints (resp. criteria of configuration decisions) or in effects of configuration decisions. In feature constraints (resp. criteria of configuration links) these operators encompass instance creation. For example, the statement y1:Y[+] in constraints means "*the instance y1 of Y is selected (i.e. this instance is created as well)*" and in effects, as already mentioned, it means "*the cardinalities of y1:Y and all its predecessors are set to [1]*". The idea of this diversity is that we do not need the possibility to express constraints like "*the cardinality of the instance y1 of Y is set to [1] but its instance is not created*". We could express this by the constraint "*y1:Y is excluded*". However, we want to provide the possibility to separate instance creation statements and inclusion (resp. selection) statements in effects of configuration decisions as illustrated in the next example.

$CL_{2.3}$
B[+] ↦ y1:Y[+]
C[+] ↦ Y$y1
D[+] ↦ X[+]

Consider configuration link $CL_{2.3}$. In order to figure out under which conditions `y1:Y` is selected, we have to consider selection statements as well as instance creation statements for this feature. The cardinality of `y1:Y` and the cardinalities of its predecessors are set to `[1]`, through the application of the configuration link, if `B` is selected in the source configuration. If `C` is selected additionally, the instance `y1` of `Y` gets created. This means that `y1:Y` gets selected if both `B` and `C` are selected in the source configuration. As in the previous example, `X` gets selected if `D` is selected. Therefore, the constraint $\neg$(`y1:Y[+]` $\wedge$ `X[+]`) leads to the derived constraint that `B`, `C` and `D` cannot be selected together. This can be expressed by the multi feature link «`excludes [B, C, D]`». The last three examples show that instance creation statements require special consideration when dealing with cloned features. We formulate this as a further requirement for feature constraint propagation.

Requirement Req-7: *Feature constraint propagation has to respect instance creation statements in effects of configuration decisions in order to propagate feature links or statements referring to cloned features.*

$CL_{2.4}$
`B[+]` $\mapsto$ `y1:Y[+]`
`C[+]` $\mapsto$ `Y$y1`
`D[+]` $\mapsto$ `X[+]`
`true` $\mapsto$ `Y$y1`

In contrast to the previous example, configuration link $CL_{2.4}$ contains an additional configuration decision, which creates the instance `y1` of `Y`. In order to answer the question whether `y1:Y` is selected, we have to consider both configuration decisions creating the corresponding instance. We know that at least one of these configuration decisions has to be applied in order to create the instance. Thus, at least one of the corresponding criteria (`C[+]` of the second configuration decision or `true` of the fourth configuration decision) has to be fulfilled. Obviously, the criterion `true` is always fulfilled and, thereby, the second configuration decision is redundant[1] since it is covered completely by the fourth one. As a result, the effect `y1:Y[+]` of the first configuration decision is a sufficient condition for the selection of `y1:Y`. Hence, the propagation's result is the feature link «`B excludes D`». Although this example does not reveal new requirements, it shows how redundancies with respect to instance creation have to be handled. Note that there is no operator destructing an already created instance.

$CL_{2.5}$
`B[+]` $\mapsto$ `Y$y1, y1:Y[+]`
`C[+]` $\mapsto$ `Y$y2, y2:Y[+]`
`D[+]` $\mapsto$ `X[+]`

Configuration link $CL_{2.5}$ contains two configuration decisions creating and selecting different instances of `Y`. When propagating the target-side constraint $\neg$(`y1:Y[+]` $\wedge$

[1] Such redundancies are consciously supported by the concept of configuration links (cf. Section 2.3).

X[+]), we obviously have to consider only configuration decisions that affect the instance y1 and not configuration decisions that affect other instances. Hence, the propagation of this constraint along configuration link $CL_{2.5}$ is analogous to the propagation along configuration link $CL_{2.2}$ and leads to the same source-side feature link «B excludes D».

However, when considering the target-side feature link «X excludes Y» instead of the constraint from above, the result of the propagation along configuration link $CL_{2.5}$ is a different one since feature links are formulated on feature level (and not on instance level as feature constraints). This feature link states that X cannot be selected together with any instance of Y. It is not possible to express this feature link as equivalent feature constraint since feature constraints are defined to be propositional logic formulae on instance level. Predicates as "for all instances of Y" or "it exists an instance of Y" cannot be formulated (cf. Section 2.2). Consequently, feature constraint propagation needs a transformation for feature constraints and a different transformation for feature links. This is captured in the following requirement.

Requirement Req-8: *Feature constraint propagation has to provide different transformations for propagating feature constraints and feature links.*

Let us go back to the propagation of the feature link «X excludes Y» along configuration link $CL_{2.5}$. We know that X gets selected if the source-side feature D is selected. Since the selection of B as well as the selection of C lead to the selection of an instance of Y, neither B nor C may occur together with D. The propagation of the feature link «X excludes Y» leads to two source-side feature links «B excludes D» and «C excludes D». This example shows a special case when propagating feature links in connection with cloned features. Feature constraint propagation has to take all statements (in all effects of the configuration link) for all instances of the cloned features into account. We formulate this as further requirement.

Requirement Req-9: *Feature constraint propagation has to consider all configuration decisions with effects referring to any instance of cloned features in order to propagate feature links in connection with cloned features.*

$CL_{2.6}$
B[+] $\mapsto$ y1:Y.Z[+]
D[+] $\mapsto$ X[+]

Now we proceed to configuration link $CL_{2.6}$ and consider the feature link «X excludes Z» between two optional features. The second configuration decision states again that X gets selected if the source-side feature D is selected. The effect of the first configuration decision also applies operator [+] to an optional feature, however, this optional feature Z is a successor of the cloned feature Y. What happens if this effect takes place is that the cardinalities of y1:Y.Z and all its predecessors are set to [1]. Nevertheless, y1:Y.Z is not selected since one of these predecessors, the instance y1:Y, is not selected. This is caused by the fact that Y is a cloned feature and the instance y1 does not exist (resp. was not created). Summarizing,

we know that neither y1:Y.Z nor a different Z on instance level can get selected through the application of the configuration link. Therefore, all configurations that can be created by the configuration link fulfill the feature link «X excludes Z». Consequently, the result of the propagation is true.

$CL_{2.7}$
B[+] ↦ Y$y1, y1:Y.Z[+] D[+] ↦ X[+]

If we consider configuration link $CL_{2.7}$, which additionally holds a statement for the instance creation of y1, the propagation of the feature link «X excludes Y» results in the feature link «B excludes D», as supposed.

Whereas if we propagate the feature link «W excludes X» instead of the feature link from above along configuration link $CL_{2.6}$, the situation changes. We already know that the first configuration decision has the effect that the cardinalities of y1:Y.Z and all its predecessors, including W, are set to [1]. Since W is no successor of a cloned feature, this is sufficient for its selection. As we can see, this effect selects only predecessors of the cloned feature Y. All successors of y1:Y through y1:Y.Z are not selected, as already mentioned, since the instance y1 is not created by this effect. This means that we derive the propagated feature link «B excludes D». Propagating the feature link «W excludes X» along configuration link $CL_{2.7}$ leads to the same result.

Now we address inclusion instead of selection statements. Consider the feature constraint ¬(X[1] ∧ y1:Y.Z[1]) instead of the feature link from above and configuration link $CL_{2.6}$ again. We have already mentioned that the effect y1:Y.Z[+] does not select y1:Y.Z. However, since Z is an optional feature, y1:Y.Z gets included by this effect. Consequently, we derive the propagated feature link «B excludes D». Propagating the constraint ¬(X[1] ∧ y1:Y.Z[1]) along configuration link $CL_{2.7}$ produces the same result.

The last-mentioned examples reveal the next requirement for feature constraint propagation. It can be seen as an addition to Requirement Req-7, which states that instance creation statements in effects have to be considered when propagating feature links or statements referring to cloned features. The next requirement also addresses the fact that instance creation statements have to be considered but refers to successors of cloned features and not to themselves. Then, instance creation statements in effects have to be considered when propagating feature links or selection statements. However, instance creation statements in effects can be neglected if inclusion statements in feature constraints referring to successors of cloned features shall be propagated, as the examples above show.

Requirement Req-10: *Feature constraint propagation has to respect instance creation statements in effects of configuration decisions in order to propagate feature links or selection statements referring to successors of cloned features.*

$CL_{2.8}$
B[+] ↦ Y$y1, y1:Y[+]
C[+] ↦ y2:Y.Z[1]
D[+] ↦ X[+]

Let us continue with configuration link $CL_{2.8}$ and reconsider the feature link «X excludes Z». In order to figure out under which circumstances the optional feature Z is selected, we already know, by Requirement Req-5, that we have to consider all configuration decisions affecting this feature initially. The second configuration decision states that the selection of the source-side feature C includes Z. Since we are interested in the selection and not in the inclusion of Z, we consider the parent Y of Z in the next step. The first configuration decision affects Y by creating and selecting an instance y1. So, does Z get selected if the effects of the first two configuration decisions are applied together? If we look at the instance names of Y, we can see that this is not the case. The first configuration decision refers to the instance y1 and the second configuration decision to the instance y2 of Y. If we are interested in the selection or inclusion state of a feature, we always have to consider the feature on instance level and not on feature level. We cannot look for a feature Z to be selected since Z is ambiguous on instance level but we have to examine every feature Z on instance level. In our example, there is no feature Z on instance level that can be selected by configuration link $CL_{2.8}$. Consequently, the propagation of «X excludes Z» delivers the source-side constraint true. This example yields that feature constraint propagation has to take the instance level instead of the feature level of the target feature model into account, as captured in the next requirement.

Requirement Req-11: *Feature constraint propagation has to consider target-side features and their parent relationships on instance level (and not on feature level) in order to propagate feature links or selection statements in feature constraints.*

Now we proceed to cloned features on the source-side of a configuration link. Remember that criteria of configuration links are feature constraints and, therefore, formulated on instance level. We are not able to state criteria as "if any instance of a cloned feature exists" but we can only refer to concrete instances of cloned features.

$CL_{2.9}$
e1:E[+] ↦ W[+]
C[+] ↦ X[+]

Consider configuration link $CL_{2.9}$ and the target-side feature link «W excludes X». The first configuration decision implies the selection of target-side feature W if the criterion e1:E[+] holds. As already mentioned, the operator [+] occurring in criteria states the selection of a feature (i.e. the corresponding instance also has to exist). In our example, this means that W gets selected if the instance e1 of E exists and X gets selected if C is selected. Consequently, the propagation of the feature link «W excludes X» leads directly to the source-side feature constraint ¬(C[+] ∧ e1:E[+]). However, this constraint cannot be expressed by a source-side feature link since it is, in contrast to feature links, formulated on instance level and

refers to an instance of a cloned feature. It forbids that C occurs with the instance e1 of E together. In contrast, the feature link «C excludes E» states that C cannot occur with any instance of E together. This is a stronger condition than the resulting feature constraint.

$CL_{2.10}$
e1:E.F[+] $\mapsto$ W[+]
C[+] $\mapsto$ X[+]

The same problems arise for successors of cloned features, as configuration link $CL_{2.10}$ illustrates. The propagation of the target-side feature link «W excludes X» along this configuration link obviously leads to the source-side feature constraint $\neg$(C[+] $\wedge$ e1:E.F[+]), which cannot be expressed as a feature link for the same reasons. By these examples, feature links can only be extracted out of feature constraints if cloned features are not involved. This requirement is an addition to Requirement Req-1 and formulated in the following.

Requirement Req-12: *Feature constraint propagation cannot extract feature links out of constraints containing statements referring to instances of cloned features or their successors.*

$CL_{2.11}$
e1:E[+] $\mapsto$ W[+]
e2:E[+] $\mapsto$ X[+]

Configuration link $CL_{2.11}$ consists of two configuration decisions with similar effects. Both of them refer to the source-side feature E, however, they address different instances. It is straightforward that these statements are handled separately and we obtain the source side constraint $\neg$(e1:E[+] $\wedge$ e2:E[+]) if we propagate the feature link «W excludes X».

$CL_{2.12}$
B[1] $\wedge$ e1:E[1] $\mapsto$ W[+]
A[1] $\wedge$ e2:E.F[1] $\mapsto$ X[+]

Now we take configuration link $CL_{2.12}$ into account, which contains inclusion statements combined with cloned features in criteria. By target-side feature link «W excludes X», we know that every source configuration fulfilling the criteria of both configuration decisions results in an invalid target configuration. We can express this condition as constraint $\neg$((B[1] $\wedge$ e1:E[1]) $\wedge$ (A[1] $\wedge$ e2:E.F[1])), which is equivalent to $\neg$(A[1] $\wedge$ B[1] $\wedge$ e1:E[1] $\wedge$ e2:E.F[1]). As already mentioned, feature constraint propagation has to convert inclusion statements into equivalent selection statements (see Requirement Req-4) after transforming a constraint. A[1] $\wedge$ B[1] can be converted into B[+] because of the tree structure of the source feature model. Subsequently, B[+] $\wedge$ e1:E[1] can be combined to e1:E[+]. However, the statements e1:E[+] and e2:E.F[1] cannot be combined further since they refer to different instances of E. Consequently, the result of the propagation is the constraint

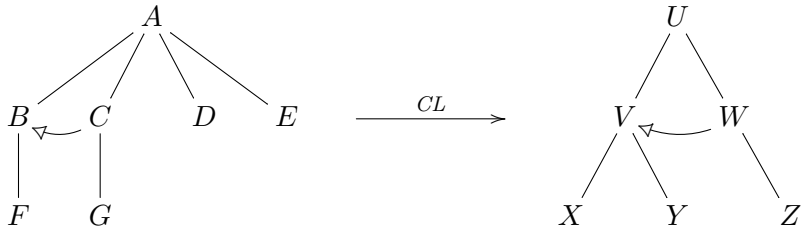


Figure 4.3: A configuration link between two feature models with inheritance

$\neg$(e1:E[+] $\wedge$ e2:E.F[1]). We can see that feature constraint propagation has to consider the instance level instead of the feature level of the source feature model when converting inclusion into selection statements. This is captured in the next requirement, which can be seen as addition to Requirement Req-4.

Requirement Req-13: *Feature constraint propagation has to consider parent relationships on instance level (and not on feature level) in order to convert inclusion statements into selection statements.*

All these examples with cloned features show that cloned features lead to more complex propagation algorithms. Especially because of instance creation statements and their semantics. These statements have to be handled in particular during the propagation.

4.3 Propagation with Feature Inheritance

In this section we take feature inheritance into account. Consider the example in Figure 4.3 and assume all features to be optional. The difficulties when dealing with feature inheritance are similar to those arising when dealing with cloned features because both concepts lead to the fact that the set of features on feature level and the set of features on instance level are not isomorphic.

$CL_{3.1}$
B[+] $\mapsto$ V.X[+]
C[+] $\mapsto$ W.X[+]
D[+] $\mapsto$ Z[+]

Let us start with inherited features in effects of configuration links. Consider configuration link $CL_{3.1}$ and the target-side feature constraint W.X[+] $\rightarrow$ Z[+]. This feature constraint refers to the inherited feature X that occurs as child of W. We cannot use the feature X within feature constraints since it is ambiguous on instance level. We can only address features on instance level (here the forms V.X or W.X of the feature X). We can directly see that the application of the configuration link selects W.X if the source-side feature C is selected and it selects Z if the source-side feature D is selected. Therefore, the target-side constraint W.X[+] $\rightarrow$ Z[+] can straightforwardly be propagated to the source-side feature link «C needs D».

In feature links we are able to address features on feature level, e.g. we are able to address the feature X. In order to propagate the feature link «X needs Z» instead of the feature constraint W.X[+] → Z[+], we have to consider the occurrences of all forms of the feature X on instance level (V.X and W.X). The first configuration decision of configuration link $CL_{3.1}$ states that V.X gets selected if the source-side feature B is selected. As a consequence, we can propagate the statement X to the source-side constraint B[+] ∨ C[+]. The feature link «X needs Z» can thus be propagated to the source-side constraint B[+]∨C[+] → D[+], which can be expressed by two feature links «B needs D» and «C needs D». From a methodological point of view, we have two possibilities to deal with feature links in combination with feature inheritance. The first one is to convert the feature link into an equivalent feature constraint (here V.X[+] ∨ W.X[+] → Z[+]) and to propagate this feature constraint. The second possibility is to propagate the feature link directly. Then we need different techniques for propagating feature links and feature constraints. Requirement Req-8 already states the necessity for these different techniques. So, this example underlines Requirement Req-8 and shows that all statements referring to any form of inherited features in all effects have to be considered when propagating feature links. This is formulated in the next requirement, which can be seen as adaptation of Requirement Req-9 to the concept of feature inheritance.

Requirement Req-14: *Feature constraint propagation has to consider all configuration decisions with effects referring to any form of inherited features on instance level in order to propagate feature links in connection with inheritance.*

$CL_{3.2}$
B[+] ↦ W.X[+]
C[+] ↦ Z[1]

We continue with configuration link $CL_{3.2}$ and the target-side feature link «X needs Z» again. We can directly see that the selection of B leads to the selection of a feature X on instance level. However, there is no configuration decision that selects Z. We already know that we also have to consider inclusion statements for Z, as stated in Requirement Req-5. Z gets included if C is selected. We continue with figuring out under which conditions W (the parent of Z) gets selected. The first configuration decision selects the feature W.X. This means that all parents of this feature are selected, too. In this context, we have to consider the parent relationship on instance level (the parent of W.X is W) and not the parent relationship on feature level (the parent of X is V). Thus, the effects of both configuration decisions together imply the selection of Z and, therefore, the propagation of the feature link «X needs Z» results in the source-side feature link «B needs C». This example does not reveal new requirements but shows the necessity of Requirement Req-11 again.

$CL_{3.3}$
B.F[+] ↦ V[+]
G[+] ↦ W[+]

Now we proceed with inherited features in criteria of configuration links. Consider the feature link «V excludes W» and configuration link $CL_{3.3}$. It seems quite obvious that the propagation results in the source-side feature link «F excludes G». However, this is not the case. Since the effect B.F[+] of the first configuration decision refers to feature F, which is inherited by C, the propagated feature constraint $\neg$(B.F[+] $\wedge$ G[+]) cannot be expressed as feature link. F has two forms on instance level: it occurs as child of B and as child of C. Accordingly, the feature link «F excludes G» is equivalent to the feature constraint $\neg$((B.F[+] $\vee$ C.F[+]) $\wedge$ G[+]), which is equivalent to $\neg$(B.F[+] $\wedge$ G[+]) $\wedge$ $\neg$(C.F[+] $\wedge$ G[+]).

$CL_{3.4}$
B.F[+] $\mapsto$ V[+]
C.F[+] $\mapsto$ V[+]
G[+] $\mapsto$ W[+]

On the contrary, if we propagate the target-side feature link «V excludes W» along configuration link $CL_{3.4}$, we exactly derive the mentioned constraint $\neg$((B.F[+] $\vee$ C.F[+]) $\wedge$ G[+]) and can thus extract feature link «F excludes G». These examples show that the extraction of feature links in combination with inherited features is not impossible, as in combination with cloned features (see Requirement Req-12), but it is much more complex than in the case without inherited features. In this example, the feature F has only two forms on instance level. If multiple inheritance relationships are contained in a feature model, a feature can eventually have numerous forms on instance level. The extraction of feature links is then even more complex. The two last-mentioned examples show that feature constraint propagation has to take all possible forms of inherited features into account for feature link extraction. This is captured in the following addition for Requirement Req-1.

Requirement Req-15: *Feature constraint propagation has to consider all possible forms of inherited features on instance level in order to extract feature links.*

$CL_{3.5}$
C.F[1] $\mapsto$ V[+]
G[+] $\mapsto$ W[+]

Finally, we take configuration link $CL_{3.5}$ into account. We can directly see that the inclusion of C.F and the selection of G lead to a target-side configuration violating the feature link «V excludes W». As a consequence, this feature link can be propagated to the source-side constraint $\neg$(C.F[1] $\wedge$ G[+]). In order to replace the contained inclusion statement by a selection statement, we have to analyze whether the parent of C.F is selected. We already know that there are two parent relationships (one on feature level and one on feature level) and we also figured out in Requirement Req-13 that we have to consider the one on instance level in order to convert inclusion statements into selection statements. The parent C of C.F (on instance level) is implicitly selected by the statement G[+] since C is a predecessor of G. Therefore, we can convert the propagated constraint into the equivalent constraint

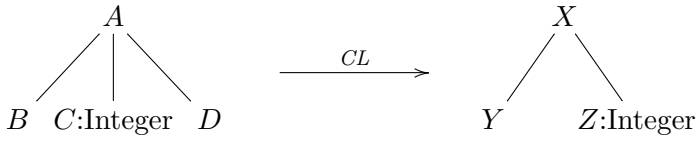


Figure 4.4: A configuration link between two feature models with parameterized features

$\neg$(C.F[+] $\wedge$ G[+]). We already know that we cannot extract a feature link out of this constraint by Requirement Req-15. This example does not reveal new requirements but shows the necessity for Requirement Req-13 even when dealing with feature inheritance. We identified this requirement during the analysis of cloned features above.

All these examples with feature inheritance show that this concept has similar effects on the propagation as the concept of cloned features. This is because both concepts imply that the sets of features on feature level and on instance level are not isomorphic. It follows that feature links cannot easily be converted into equivalent feature constraints anymore. In the case of feature inheritance, they can be expressed by more complex feature constraints and in the case of cloned features, they cannot be expressed as equivalent feature constraints at all.

4.4 Propagation with Parameterized Features

Figure 4.4 depicts a small example for feature models with parameterized features. Consider all features to be optional and the feature link «Y excludes Z». Note that feature constraints cannot address the values of parameterized features (e.g. Z=1 is not a valid feature constraint).

$CL_{4.1}$
C[+] $\mapsto$ Y[+]
B[+] $\mapsto$ Z[+]

Configuration link $CL_{4.1}$ does not assign a value to Z. As we can see, the occurrence of parameterized features does not affect the propagation of the feature link «Y excludes Z» in this example and we derive the source-side feature link «B excludes C».

$CL_{4.2}$
C[+] $\mapsto$ Y[+]
B[+] $\mapsto$ Z[+]
D[+] $\mapsto$ Z=1

If we add a third configuration decision that assigns a value to the parameterized feature Z to the configuration link (see $CL_{4.2}$), the propagation proceeds analogously.

This is caused by the fact that feature links (as well as feature constraints) refer only to the selection (resp. inclusion) states of features and not to the values of parameterized features. We can simply omit all configuration steps setting values of parameterized features.

Parameterized features in the source feature models of configuration links do not affect the propagation either. In this context, it is important to remember that criteria of configuration links are defined to be feature constraints and cannot thus contain statements referring to the values of parameterized features (e.g. `Z=1`).

4.5 Conclusions of the Problem Analysis

Since feature constraint propagation is based on logical substitution of propositional logic formulae, it can propagate arbitrary feature links and feature constraints. Further restrictions as, for example, feature groups and the instance count of cloned features are not addressed by the propagation. This means that the technique does not make the effects of these restrictions apparent in higher-level feature models. Nevertheless, these concepts can be used within feature models without interfering the propagation.

The examples of this chapter show the challenges and the complexity of feature constraint propagation. Especially advanced feature modeling concepts complicate the technique drastically. In addition, separating the notions of inclusion and selection and the possibility to formulate constraints and effects (of configuration decisions) which address inclusion as well as selection is very challenging for the definition of feature constraint propagation. Hence, an ad-hoc implementation is not possible and a well-thought-out concept, as presented in Chapter 5, is required. In addition, a sound formalization of feature constraint propagation and all its basics, as presented in Chapter 6, is useful to prevent ambiguities before the technique is implemented and evaluated (cf. Chapter 7).

For convenience, all requirements identified in this chapter are summarized in the following.

Req-1: *Feature constraint propagation has to provide techniques for extracting feature links out of constraints.*

Req-2: *Feature constraint propagation has to respect implicit effects arising from selection statements in effects of configuration decisions.*

Req-3: *Feature constraint propagation has to minimize resulting formulae and convert them into an appropriate form.*

Req-4: *Feature constraint propagation has to convert propagated inclusion statements into selection statements respecting the feature model's tree structure.*

Req-5: *Feature constraint propagation always has to consider inclusion as well as selection statements in effects of configuration decisions – even when propagating feature links or selection statements in feature constraints.*

Req-6: *Feature constraint propagation has to deal with the facts that mandatory features are always included and abstract features are always excluded.*

Req-7: *Feature constraint propagation has to respect instance creation statements in effects of configuration decisions in order to propagate feature links or statements referring to cloned features.*

Req-8: *Feature constraint propagation has to provide different transformations for propagating feature constraints and feature links.*

Req-9: *Feature constraint propagation has to consider all configuration decisions with effects referring to any instance of cloned features in order to propagate feature links in connection with cloned features.*

Req-10: *Feature constraint propagation has to respect instance creation statements in effects of configuration decisions in order to propagate feature links or selection statements referring to successors of cloned features.*

Req-11: *Feature constraint propagation has to consider target-side features and their parent relationships on instance level (and not on feature level) in order to propagate feature links or selection statements in feature constraints.*

Req-12: *Feature constraint propagation cannot extract feature links out of constraints containing statements referring to instances of cloned features or their successors.*

Req-13: *Feature constraint propagation has to consider parent relationships on instance level (and not on feature level) in order to convert inclusion statements into selection statements.*

Req-14: *Feature constraint propagation has to consider all configuration decisions with effects referring to any form of inherited features on instance level in order to propagate feature links in connection with inheritance.*

Req-15: *Feature constraint propagation has to consider all possible forms of inherited features on instance level in order to extract feature links.*

Chapter 5

Feature Constraint Propagation: Concept

In this chapter the technique of feature constraint propagation is rigorously introduced on a conceptual level. The requirements identified in Chapter 4 are revisited and form the basis for the definition of the technique. All steps of feature constraint propagation are described in detail and the runtimes of the underlying algorithms are approximated. The formalization, verification and evaluation of feature constraint propagation are not part of this chapter and follow later in this thesis.

After the introduction of the technique in Section 5.1, some possible extensions of feature constraint propagation are presented and discussed in Section 5.2. Finally, alternative approaches that could be used instead of feature constraint propagation are introduced and compared with the technique in Section 5.3.

5.1 The Technique of Feature Constraint Propagation in Six Steps

The technique of feature constraint propagation allows to transform arbitrary constraints of one feature model backwards along a configuration link into constraints of another feature model. It consists of six consecutive steps, which were identified according to the requirements of Chapter 4. First of all, the configuration link is expanded in order to make implicit effects explicit (cf. Requirement Req-2). Then, several reverse mappings of configured feature identifiers[1] (i.e. features on instance level) and features (i.e. features on feature level) are constructed in order to transform feature constraints and feature links separately from each other (cf. Requirement Req-8). Subsequently, a target-side constraint is transformed according to these mappings into a source-side constraint. Finally, three restructuring steps are applied to the resulting constraint. First, it is minimized and converted into a conjunctive normal form (cf. Requirement Req-3). Second, inclusion statements are replaced by selection statements as far as possible without changing the semantics

[1] In order to avoid ambiguities, we sometimes denote features on instance level by the term 'configured feature identifiers', as already mentioned in Section 2.2. With 'features' we usually mean features on feature level.

of the constraint (cf. Requirement Req-4) and, third, feature links are extracted out of the constraint (cf. Requirement Req-1). The final result of the propagation are source-side feature links and source-side feature constraints. They narrow the set of source configurations such that every source configuration fulfilling these constraints leads, through the application of the configuration link, to a target configuration fulfilling the original constraints. Vice versa, every source configuration violating these constraints results in a target configuration violating the original constraints (cf. Figure 1.3 and Figure 1.4). The correctness of these properties is shown in Chapter 6.

In the following, the six steps of feature constraint propagation are described. We illustrate every step of feature constraint propagation by applying it to our car product line depicted in Figure 2.5.

5.1.1 Step 1: Expand Configuration Link

The first step of feature constraint propagation is the expansion of the configuration link. Requirement Req-2 states that feature constraint propagation has to respect implicit effects that arise by selection statements in effects of configuration links. Our idea to meet this requirement is to replace all selection statements by inclusion statements in all effects of a configuration link. Configuration links without selection statements in effects do not produce any implicit effects. Note that deselection statements in effects do not create implicit effects since they are simply interpreted as exclusion statements. We decided to replace selection by inclusion statements not only in effects but also in criteria of all configuration decisions. This is required for our inclusion-to-selection algorithm, which is described later in Step 5. The result is an *expanded configuration link* according to the following definition.

Definition 5.1.1 (Expanded Configuration Link)**.** A configuration link is called *expanded* if it does not contain any selection statements (neither in criteria nor in effects of configuration decisions).

The notion “expanded” comes from the fact that the individual configuration decisions get longer by the expansion activity.

Remember that a configuration link consists of a set of configuration decisions, which, in turn, consist of a criterion and an effect each. Criteria are feature constraints (w.r.t. the source feature model) and effects are sets of atomic configuration steps. Because of this diversity of criteria and effects, we have to handle them separately when expanding the configuration link. In any criterion, every selection statement of a feature is replaced by an equivalent conjunction of inclusion statements for the feature itself and all its predecessors. On the other hand, in any effect, every selection statement of a feature is replaced by a set of inclusion statements for the feature itself and all its predecessors. Note that we have to respect the parent relationships on instance level in order to expand selection statements in a configuration link.

$$\texttt{true} \mapsto \begin{pmatrix} \texttt{Wiper\$front}, \\ \textbf{BodyElectronics[1]}, \\ \textbf{front:Wiper[1]} \end{pmatrix} \quad (5.1)$$

$$\textbf{USA[1]} \wedge \textbf{Market[1]} \mapsto \texttt{CruiseControl[1], Radar[1]} \quad (5.2)$$

$$\begin{pmatrix} (\textbf{USA[1]} \wedge \textbf{Market[1]}) \\ \vee(\textbf{Canada[1]} \wedge \textbf{Market[1]}) \end{pmatrix} \mapsto \texttt{CruiseControl[1]} \quad (5.3)$$

$$\textbf{Comfort[1]} \wedge \textbf{Model[1]} \mapsto \texttt{front:Wiper.RainSensor[1]} \quad (5.4)$$

$$\begin{pmatrix} \textbf{ComfortPlus[1]} \\ \wedge\textbf{Comfort[1]} \\ \wedge\textbf{Model[1]} \end{pmatrix} \mapsto \begin{pmatrix} \texttt{Wiper\$rear}, \\ \texttt{rear:Wiper[1]}, \\ \texttt{rear:Wiper.RainSensor[1]} \end{pmatrix} \quad (5.5)$$

$$\begin{pmatrix} (\textbf{Convertible[1]} \wedge \textbf{Model[1]}) \\ \wedge(\textbf{Comfort[1]} \wedge \textbf{Model[1]}) \end{pmatrix} \mapsto \texttt{RoofAutomatic[1]} \quad (5.6)$$

$$\neg(\textbf{Convertible[1]} \wedge \textbf{Model[1]}) \mapsto \texttt{RoofAutomatic[0]} \quad (5.7)$$

Figure 5.1: The expanded configuration link of Figure 2.5

Runtime Approximation

The iteration over the configuration link is dependent on the number of atomic statements (a statement denotes a configured feature identifier with a cardinality) of the configuration link ($|Statements|$) and the expansion of an individual statement is dependent on the depth of the feature models ($Depth$). In the worst-case all statements are selection statements, which have to be expanded. This leads to a worst-case runtime of $\mathcal{O}(|Statements| * Depth)$.

Example

If we apply the expansion algorithm to the configuration link of Figure 2.5, we obtain the expanded configuration link depicted in Figure 5.1. Changes are highlighted in bold.

5.1.2 Step 2: Calculate Reverse Mappings

The second step of feature constraint propagation, the calculation of the reverse mappings, is the centerpiece of the technique. It is the foundation for the transformation function of target-side constraints to source-side constraints, which is introduced in Step 3. The idea of this step is to calculate mappings from variables of target-side constraints to source-side constraints. By Requirement Req-8, the need for different transformations for propagating feature constraints and feature links arises. In order to define these different transformations, different reverse mappings are also required. Since feature constraints are formulated on instance level and can contain inclusion as well as selection statements, two kinds of mappings are required: *the reverse inclusion mapping of configured feature identifiers* and *the reverse selection*

mapping of configured feature identifiers. The former one maps every target-side configured feature identifier to a source-side constraint, such that the fulfillment of this constraint is a necessary and sufficient condition for the inclusion of the mapped configured feature identifier. In other terms, only every source configuration that fulfills the resulting constraint leads, through the application of the configuration link, to a target configuration in which the mapped configured feature identifier is included. The second mentioned mapping does the same with respect to the selection of a target-side configured feature identifier. This means that the fulfillment of the resulting constraint is a necessary and sufficient condition for the selection of the mapped configured feature identifier. In contrast to feature constraints, feature links are defined on feature level and evaluated with respect to selection. Therefore, a third reverse mapping in order to transform feature links has to be defined: *the reverse selection mapping of features*. The idea of this mapping is similar to the already described reverse selection mapping of configured feature identifiers: it maps a target-side feature to a source-side feature constraint such that the fulfillment of this constraint is a necessary and sufficient condition for the selection of a configured feature identifier pointing to the mapped feature. We do not need a reverse inclusion mapping of features since feature links are always evaluated with respect to selection.

Reverse Inclusion Mapping of Configured Feature Identifiers

According to Requirement Req-6 and Requirement Req-7, we have to make a case distinction regarding the cardinalities of features in order to define the reverse inclusion mapping of configured feature identifiers $RevInc_{CL}^{\mathcal{CFID}}$ with respect to the given configuration link *CL*. Let *cfid* be a target-side configured feature identifier.

Case 1: *cfid* points to a mandatory feature. According to Requirement Req-6, we know that mandatory features are always included.

$$RevInc_{CL}^{\mathcal{CFID}}(cfid) = \texttt{true}$$

Case 2: *cfid* points to an abstract feature. According to Requirement Req-6, we know that abstract features are always excluded, i.e. never included.

$$RevInc_{CL}^{\mathcal{CFID}}(cfid) = \texttt{false}$$

Case 3: *cfid* points to an optional feature. Optional features are included if at least one configuration decision that includes the feature is applied. Since an exclude is prioritized over an include, the additional condition that no configuration decision that excludes the feature is applied has to hold. By Requirement Req-2, we know that we also have to consider implicit effects of configuration links. Remember that we have already expanded the configuration link to meet this requirement. All implicit effects are made explicit and we do not have to care for this requirement anymore. Let $\varphi_1, \varphi_2, ...$ be all criteria of configuration decisions (of the expanded configuration link) that include *cfid* and $\psi_1, \psi_2, ...$ be all criteria of configuration decisions (of the expanded configuration link) that exclude *cfid*.

$$RevInc_{CL}^{\mathcal{CFID}}(cfid) = (\varphi_1 \vee \varphi_2 \vee ...) \wedge \neg(\psi_1 \vee \psi_2 \vee ...)$$

Case 4: *cfid* points to an instance of a cloned feature. In the case of cloned features, we know, by Requirement Req-7, that we also have to consider instance creation statements in effects. At least one configuration decision that creates the instance of *cfid* has to be applied since the existence of the corresponding instance is a necessary condition for the configured feature identifier to be included. Let $\varphi_1, \varphi_2, ...$ be all criteria of configuration decisions (of the expanded configuration link) that set the cardinality of *cfid* to `[1]` and $\psi_1, \psi_2, ...$ be all criteria of configuration decisions (of the expanded configuration link) that set the cardinality of *cfid* to `[0]` and, additionally, $\vartheta_1, \vartheta_2, ...$ be all criteria of configuration decisions (of the expanded configuration link) that create the instance of *cfid*. Remember that setting the cardinality of an instance of a cloned feature to `[1]` is not a sufficient condition for its inclusion (cf. Chapter 4).

$$RevInc_{CL}^{\mathcal{CFID}}(cfid) = (\varphi_1 \vee \varphi_2 \vee ...) \wedge \neg(\psi_1 \vee \psi_2 \vee ...) \wedge (\vartheta_1 \vee \vartheta_2 \vee ...)$$

Algorithm 5.1 shows the pseudo-code for the calculation of the reverse inclusion mapping of configured feature identifiers. Initially, the reverse inclusion mapping is empty. The algorithm iterates over all configuration steps of all configuration decisions (lines 2 and 3). A configuration step is ignored if (1) it does not set the cardinality of a configured feature identifier (e.g. it could create an instance of a cloned feature or set the value of a parameterized feature; line 4), (2) the reverse inclusion mapping for this configured feature identifier has already been calculated (line 6) or (3) the configured feature identifier points to a cloned feature itself (and not to an instance of a cloned feature; line 7). For every (non-ignored) configured feature identifier *cfid*, the algorithm iterates one more time over all configuration steps of all configuration decisions (lines 8 and 9) and analyzes all criteria of configuration steps affecting the same configured feature identifier *cfid*. In the case of optional features, only configuration steps including or excluding the configured feature identifier are considered (lines 16, 17 and 20) and in the case of cloned features, creation statements for the corresponding instance are additionally considered (lines 11 and 12). All criteria of the corresponding configuration decisions are collected in sets and, finally, connected to logical formulae according to the cases defined above (from line 27). Note that the reverse inclusion mapping does not have to be calculated for configured feature identifiers pointing to abstract or mandatory features because they are always excluded, resp. included, as mentioned above.

Runtime Approximation

The iteration over the statements of all effects of the expanded configuration link (denoted by *ExEfStatements*) in lines 2 and 3 of Algorithm 5.1 has a runtime of $\mathcal{O}(|ExEfStatements|)$. In the worst-case, every iteration consists of another iteration over the statements of all effects in lines 8 and 9 with the same runtime $\mathcal{O}(|ExEfStatements|)$. Further statements within the loops have a constant runtime and can therefore be neglected in the $\mathcal{O}$-calculus. The `makeDisjunction` operation iterates over all identified formulae and connects them with disjunctions. In the worst-case, it has to connect all criteria of the configuration link. Since every configuration decision has exactly one criterion, this operation has a worst-case runtime

Algorithm 5.1: Calculation of the reverse inclusion mapping

Data: expanded configuration link *CL*
Result: reverse inclusion mapping RevIncCFID

```
1  RevIncCFID ← ∅
2  foreach configuration decision (φ, Steps) of CL do
3      foreach step s of Steps do
4          if not isSetCardinalityStatement(s) then continue
5          cfid ← getCfid(s); incCriteria ← ∅; excCriteria ← ∅; instCriteria ← ∅
6          if cfid ∈ dom(RevIncCFID) then continue; // already calculated
7          if cfid points to a cloned feature itself then continue
8          foreach configuration decision (φ', Steps') of CL do
9              foreach step s' of Steps' do
10                 if getCfid(s') = cfid then
11                     if isCreationStatement(s') then // cloned feature
12                         if getInstanceName(s') = getInstanceName(s) then
13                             instCriteria ← instCriteria ∪ {φ'}
14                         end
15                     end
16                     if isSetCardinalityStatement(s') then
17                         if getCardinality(s') = [1] then
18                             incCriteria ← incCriteria ∪ {φ'}
19                         end
20                         if getCardinality(s') = [0] then
21                             excCriteria ← excCriteria ∪ {φ'}
22                         end
23                     end
24                 end
25             end
26         end
27         if cfid points to an instance of a cloned feature then
28             RevIncCFID ← RevIncCFID ∪
29                                 {cfid ↦ makeDisjunction (incCriteria)
30                                         ∧¬(makeDisjunction (excCriteria))
31                                         ∧ makeDisjunction (instCriteria)}
32         else
33             RevIncCFID ← RevIncCFID ∪
34                                 {cfid ↦ makeDisjunction (incCriteria)
35                                         ∧¬(makeDisjunction (excCriteria))}
36         end
37     end
38 end
```

of $\mathcal{O}(|ConfigurationDecisions|)$ with $|ConfigurationDecisions|$ being the number of configuration decisions. It is used up to three times for every statement of the configuration link's effects. Since the number of configuration decisions is always smaller than the number of statements of all effects, the runtime of the `makeDisjunction` operation can be neglected in the $\mathcal{O}$-calculus. This leads to a worst-case runtime of $\mathcal{O}(|ExEfStatements|^2)$ for the algorithm.

$$
\begin{aligned}
&\mathcal{O}(|ExEfStatements|) * (\mathcal{O}(|ExEfStatements|) + 3 * \mathcal{O}(|ConfigurationDecisions|)) \\
=&\mathcal{O}(|ExEfStatements|) * (\mathcal{O}(|ExEfStatements|) + \mathcal{O}(|ConfigurationDecisions|)) \\
=&\mathcal{O}(|ExEfStatements|) * (\mathbf{max}(\mathcal{O}(|ExEfStatements|), \mathcal{O}(|ConfigurationDecisions|))) \\
=&\mathcal{O}(|ExEfStatements|) * \mathcal{O}(|ExEfStatements|) \\
=&\mathcal{O}(|ExEfStatements|^2)
\end{aligned}
$$

The algorithm can be adapted such that the reverse inclusion of individual configured feature identifiers is only calculated if it is required. In this case, the outer loops in lines 2 and 3 are not required. The calculation of the reverse inclusion mapping for a given configured feature identifier has accordingly a worst-case runtime of $\mathcal{O}(|ExEfStatements|)$.

Example

In order to illustrate the reverse inclusion mapping, we apply it to every target-side configured feature identifier occurring in the configuration link of the example in Figure 2.5. It is based on the expanded configuration link depicted in Figure 5.1. The numbers on the right side refer to the configuration decisions of this configuration link which influence the corresponding mapping.

$$
\begin{aligned}
&RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics}) \\
&\qquad = \texttt{true} && 5.1
\end{aligned}
$$

$$
\begin{aligned}
&RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper}) \\
&\qquad = \texttt{true} \wedge \texttt{true} && 5.1
\end{aligned}
$$

$$
\begin{aligned}
&RevInc_{CL}^{\mathcal{CFID}}(\texttt{CruiseControl}) \\
&\qquad = \begin{pmatrix} \texttt{USA[1]} \\ \wedge\texttt{Market[1]} \end{pmatrix} \vee \begin{pmatrix} (\texttt{USA[1]} \wedge \texttt{Market[1]}) \\ \vee(\texttt{Canada[1]} \wedge \texttt{Market[1]}) \end{pmatrix} && 5.2,\ 5.3
\end{aligned}
$$

$$
\begin{aligned}
&RevInc_{CL}^{\mathcal{CFID}}(\texttt{Radar}) \\
&\qquad = \texttt{USA[1]} \wedge \texttt{Market[1]} && 5.2
\end{aligned}
$$

$$
\begin{aligned}
&RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper.RainSensor}) \\
&\qquad = \texttt{Comfort[1]} \wedge \texttt{Model[1]} && 5.4
\end{aligned}
$$

$$RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper}) = \begin{pmatrix} \texttt{ComfortPlus[1]} \\ \wedge\texttt{Comfort[1]} \\ \wedge\texttt{Model[1]} \end{pmatrix} \wedge \begin{pmatrix} \texttt{ComfortPlus[1]} \\ \wedge\texttt{Comfort[1]} \\ \wedge\texttt{Model[1]} \end{pmatrix} \quad 5.5$$

$$RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper.RainSensor}) = \begin{pmatrix} \texttt{ComfortPlus[1]} \\ \wedge\texttt{Comfort[1]} \\ \wedge\texttt{Model[1]} \end{pmatrix} \quad 5.5$$

$$RevInc_{CL}^{\mathcal{CFID}}(\texttt{RoofAutomatic}) = \begin{pmatrix} (\texttt{Convertible[1]} \wedge \texttt{Model[1]}) \\ \wedge(\texttt{Comfort[1]} \wedge \texttt{Model[1]}) \end{pmatrix} \wedge \neg\neg \begin{pmatrix} \texttt{Convertible[1]} \\ \wedge\texttt{Model[1]} \end{pmatrix} \quad 5.6,\ 5.7$$

Reverse Selection Mapping of Configured Feature Identifiers

The reverse selection mapping of configured feature identifiers $RevSel_{CL}^{\mathcal{CFID}}$ answers the question under which conditions a target-side configured feature identifier is selected. Since the reverse inclusion mapping of configured feature identifiers answers the subordinate question under which conditions a configured feature identifier is included, we use it as basis for the reverse selection mapping of configured feature identifiers. By Requirement Req-5, we know that we have to consider inclusion as well as selection statements in effects of configuration links in order to propagate selection statements. Since expanded configuration links contain only inclusion statements, we can neglect special cases for selection statements. Let *cfid* be a target-side configured feature identifier. We distinguish the following two cases.

Case 1: *cfid* points to a root feature. In the case of root features, the notions of selection and inclusion are equivalent.

$$RevSel_{CL}^{\mathcal{CFID}}(cfid) = RevInc_{CL}^{\mathcal{CFID}}(cfid)$$

Note that $RevInc_{CL}^{\mathcal{CFID}}$ already meets Requirement Req-6 and Requirement Req-7.

Case 2: *cfid* points to a non-root feature. A configured feature identifier pointing to a non-root feature gets selected through the application of the configuration link if and only if it gets included itself and its parent gets selected. By Requirement Req-11, we know that we have to respect the parent relationship *Parent* on instance level and not the one on feature level. Another important requirement for this case is that the reverse selection mapping has to respect instance creation in effects in order to map selection statements for successors of cloned features (cf. Requirement Req-10).

$$RevSel_{CL}^{\mathcal{CFID}}(cfid) = RevInc_{CL}^{\mathcal{CFID}}(cfid) \wedge RevSel_{CL}^{\mathcal{CFID}}(Parent(cfid))$$

Note that Requirement Req-10 is met implicitly by the definition of the underlying reverse inclusion mapping.

Runtime Approximation

The reverse selection mapping of a configured feature identifier is dependent on the depth of the target feature model (*Depth*) and has a worst-case runtime of $\mathcal{O}(Depth)$ because the reverse inclusion mapping for all configured feature identifiers has already been calculated. In order to calculate the reverse selection mapping for all configured feature identifiers of the configuration link, one iteration over all statements of all effects of the expanded configuration link (*ExEfStatements*) is required. This leads to a worst-case runtime of $\mathcal{O}(Depth * |ExEfStatements|)$ for this step.

Example

We illustrate the reverse selection mapping of configured feature identifiers by applying it stepwise to some selected configured feature identifiers of our example. The reverse inclusion mapping of the configured feature identifiers has already been calculated in the previous step.

$$
\begin{aligned}
&RevSel_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper.RainSensor})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper.RainSensor}) \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper.RainSensor}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper})\\
&\qquad \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper.RainSensor}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{front:Wiper})\\
&\qquad \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})
\end{aligned}
$$

$$
\begin{aligned}
&RevSel_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper}) \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})
\end{aligned}
$$

$$
\begin{aligned}
&RevSel_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper.RainSensor})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper.RainSensor}) \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper.RainSensor}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper})\\
&\qquad \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})\\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper.RainSensor}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{rear:Wiper})\\
&\qquad \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})
\end{aligned}
$$

$$
\begin{aligned}
&RevSel_{CL}^{\mathcal{CFID}}(\texttt{Radar}) \\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{Radar}) \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{CruiseControl}) \\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{Radar}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{CruiseControl}) \\
&\qquad \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics}) \\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{Radar}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{CruiseControl}) \\
&\qquad \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})
\end{aligned}
$$

$$
\begin{aligned}
&RevSel_{CL}^{\mathcal{CFID}}(\texttt{RoofAutomatic}) \\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{RoofAutomatic}) \wedge RevSel_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics}) \\
&\quad = RevInc_{CL}^{\mathcal{CFID}}(\texttt{RoofAutomatic}) \wedge RevInc_{CL}^{\mathcal{CFID}}(\texttt{BodyElectronics})
\end{aligned}
$$

Reverse Selection Mapping of Features

It seems likely that the reverse selection mapping of features $RevSel_{CL}^{F}$ is based on a reverse inclusion mapping of features (analogous to the mappings of configured feature identifiers) but this is not possible because, according to Requirement Req-11, we have to consider the parent relationship on instance level and not the one on feature level. Hence, the reverse selection mapping of features is based on the reverse selection mapping of configured feature identifiers, which already respects the parent relationship on instance level. All requirements that are met by the underlying mappings of configured feature identifiers are also met by this mapping of features. When dealing with cloned features, all instances have to be considered according to Requirement Req-9. Moreover, if feature inheritance occurs, all forms of inherited features on instance level have to be considered, as Requirement Req-14 states. Let F be a target-side feature and $cfid_{F,1}, cfid_{F,2}, ...$ be all target-side configured feature identifiers that are contained in the effect of a configuration decision of the expanded configuration link and point to F.

$$RevSel_{CL}^{F}(F) = RevSel_{CL}^{\mathcal{CFID}}(cfid_{F,1}) \vee RevSel_{CL}^{\mathcal{CFID}}(cfid_{F,2}) \vee ...$$

Runtime Approximation

The algorithm contains an iteration over all features $|Features|$, which has a runtime of $\mathcal{O}(|Features|)$. For every cloned feature F, an iteration over all configured feature identifiers that are mapped by the reverse selection mapping of configured feature identifiers is conducted. The already calculated mapping results of configured feature identifiers pointing to F are connected by disjunctions. The last-mentioned iteration has a runtime of $\mathcal{O}(|ExEfStatements|)$. The set $ExEfStatements$ denotes, as already mentioned, all statements of all effects of the expanded configuration link. This leads to a worst-case runtime of $\mathcal{O}(|Features| * |ExEfStatements|)$ for the calculation of the reverse selection mapping of all features.

Example

We apply the reverse selection mapping of features to some features of our example. The reverse selection mapping of the configured feature identifiers has already been calculated in the previous step.

$$\begin{aligned} &RevSel^{F}_{CL}(\texttt{Radar}) \\ &\quad = RevSel^{\mathcal{CFID}}_{CL}(Radar) \end{aligned}$$

$$\begin{aligned} &RevSel^{F}_{CL}(\texttt{RainSensor}) \\ &\quad = RevSel^{\mathcal{CFID}}_{CL}(front : Wiper.RainSensor) \\ &\qquad \vee RevSel^{\mathcal{CFID}}_{CL}(rear : Wiper.RainSensor) \end{aligned}$$

5.1.3 Step 3: Transform Logical Formulae

In this step target-side feature constraints and feature links are transformed into source-side feature constraints. We know, by Requirement Req-8, that we have to handle feature constraints and feature links separately. In the case of feature constraints, we replace inclusion statements for configured feature identifiers by their reverse inclusion mappings and, analogously, selection statements for configured feature identifiers by their reverse selection mappings. In the case of feature links, we initially express them by logical formulae and replace the features by their reverse selection mappings afterwards. This allows to transform every propositional logic formula over configured feature identifiers and features. The approach is therefore not limited to the three mentioned feature link types. The transformation of feature constraints with respect to configuration link CL is denoted by $Trafo^{\mathcal{CFID}}_{CL}$ and the one of feature links is denoted by $Trafo^{F}_{CL}$.

Runtime Approximation

The algorithm iterates over all statements *CStatements* of the constraint to be propagated and replaces them by their reverse mappings. Because the mappings have already been calculated, this step has a runtime of $\mathcal{O}(|CStatements|)$.

Example

We apply the transformations to our example in Figure 2.5. The reverse mappings are shown above.

$$\begin{aligned} &Trafo^{\mathcal{CFID}}_{CL}(\neg(\texttt{RoofAutomatic[+]} \wedge \texttt{rear:Wiper[+]})) \\ &\quad = \neg(RevSel^{\mathcal{CFID}}_{CL}(\texttt{RoofAutomatic}) \wedge RevSel^{\mathcal{CFID}}_{CL}(\texttt{rear:Wiper})) \end{aligned}$$

$$Trafo^{F}_{CL}(\neg(\texttt{RainSensor} \wedge \texttt{Radar})) \\ = \neg(RevSel^{F}_{CL}(\texttt{RainSensor}) \wedge RevSel^{F}_{CL}(\texttt{Radar}))$$

5.1.4 Step 4: Minimize Logical Formulae

Requirement Req-3 states that we have to minimize the resulting source-side constraints and convert them into an appropriate form after the transformation. There are various well-known minimization algorithms for logical formulae. Most of them were originally developed for optimization of programmable logic arrays (PLAs) [RSV87]. These algorithms get a representation of a logical formula as input and deliver an equivalent statement in minimal disjunctive or conjunctive normal form, i.e. a disjunction of conjunctions or a conjunction of disjunctions, in which the number of terms is minimal. Two very popular algorithms for logic minimization are Karnaugh mapping [Kar53] and the Quine-McCluskey algorithm [McC56]. Although the Quine-McCluskey algorithm is more efficient than Karnaugh mapping when dealing with more than four variables, its application is limited because the runtime grows exponentially with the number of variables. This is due to the fact that the problem of Boolean formula minimization is very complex – it is at least in the complexity class NP-complete [UVSv06]. Consequently, formulae with numerous variables should be minimized with faster but not exact heuristics. In our use case it is not essential whether the solution is really minimal or a good approximation, as long as the formula is equivalent to the original formula. The ESPRESSO algorithm [RSV87] is the most popular heuristic algorithm for the minimization of logical formulae. It is used in this step of feature constraint propagation because it delivers good results in an adequate runtime. Alternatively, every other minimization algorithm could be used as well.

The second part of Requirement Req-3 claims that resulting constraints have to be converted in an "appropriate form". Since several feature links and feature constraints in feature models are implicitly connected via conjunction, we decided for a conjunctive, instead of a disjunctive, normal form as representative. We can then add single terms as separate constraints to the source model later. This is motivated by practice. When engineers create constraints for feature models they generally do not formulate one single complex constraint but several smaller constraints.

Runtime Approximation

The runtime of this step is dependent on the used minimization algorithm. However, the problem of minimizing logical formulae is, as already mentioned, at least NP-complete. There are several variants of popular logic minimization problems, which differ in their inputs. Table 5.1 summarizes these variants. In addition, the complexity classes and references to supplementing literature of all minimization problems are depicted. The listed supplementing literature contains the proofs that the individual problems are in the specified complexity classes. In our approach the inputs are arbitrary propositional logic formulae. NP-complete problems probably

Input	Complexity class	Supplementing literature
(incomplete or full) truth tables	NP-complete	[UVSv06]
formulae in normal forms	$\sum_2^P$-complete	[Uma98, UVSv06]
arbitrary formulae	$\sum_2^P$-complete	[BU11]

Table 5.1: Variants of minimization problems of propositional logic formulae and their complexity classes

require exponential time, $\sum_2^P = NP^{NP}$-complete problems probably require exponential time with access to an NP oracle [UVSv06]. It seems impossible that the minimization problem can be resolved in polynomial time. Even P=NP would not resolve whether this problem can be solved in polynomial time. The more probable case that P$\neq$ NP would prove that this problem cannot be solved in polynomial time. Consequently, we estimate an exponential runtime for this step $\mathcal{O}(2^{|CStatements|})$ with $|CStatements|$ being the number of statements of the constraint.

Example

The application of the minimization to the resulting constraints of our example, calculated in the previous step, leads to the following.

$$Trafo_{CL}^{\mathcal{CFID}}(\neg(\texttt{RoofAutomatic[+]} \wedge \texttt{rear:Wiper[+]}))$$
$$\equiv \neg\texttt{Convertible[1]} \vee \neg\texttt{ComfortPlus[1]} \vee \neg\texttt{Comfort[1]} \vee \neg\texttt{Model[1]}$$

$$Trafo_{CL}^{F}(\neg(\texttt{RainSensor} \wedge \texttt{Radar}))$$
$$\equiv \neg\texttt{Comfort[1]} \vee \neg\texttt{Model[1]} \vee \neg\texttt{USA[1]} \vee \neg\texttt{Market[1]}$$

5.1.5 Step 5: Lift Inclusion to Selection Statements

In this step of feature constraint propagation the minimized constraints are processed further. The idea of this step is to replace inclusion statements by equivalent selection statements, as far as possible (cf. Requirement Req-4). For this purpose, the tree structure of the source feature model has to be respected. Besides the fact that feature constraints get shorter and more understandable by this step, it is also required as preparation for the next step – the extraction of feature links. The *inclusion-to-selection algorithm* works only with conjunctive normal forms. It is a pure syntactic algorithm, in the sense that it analyzes the structure of the formula and not its semantics. This strategy is more efficient than a semantic analysis of the formula. However, it does not necessarily produce the optimal result. Several optimization steps preserving the semantics of the formula are performed by the algorithm. Their optimization goal is to replace as many inclusion statements as possible by selection statements. All these steps are based on two quite obvious facts.

Fact 5.1.2. A configured feature identifier pointing to a root feature is selected if and only if it is included.

Fact 5.1.3. A configured feature identifier pointing to a non-root feature is included if and only if it is included itself and its parent is selected.

Before we start with the description of the inclusion-to-selection algorithm, we introduce some terms to avoid ambiguities. Consider the feature constraint in conjunctive normal form $(\neg cfid_1\texttt{[1]} \vee cfid_2\texttt{[+]}) \wedge ....$ We call $(\neg cfid_1\texttt{[1]} \vee cfid_2\texttt{[+]})$ a term of the constraint. A *term* consists of a disjunction of statements for configured feature identifiers. A *statement* is a negated or a non-negated inclusion or selection statement for a configured feature identifier, i.e. a negated or a non-negated configured feature identifier with a cardinality `[1]` or `[+]`. We call a statement *negated* if it contains a negation. In the example, $\neg cfid_1\texttt{[1]}$ is a negated inclusion statement for $cfid_1$ and $cfid_2\texttt{[+]}$ is a non-negated selection statement for $cfid_2$.

The inclusion-to-selection algorithm consists of three optimization steps, the initial optimization, the inner term optimization and the cross term optimization. Each of them is described by several replacement rules. The algorithm starts with the initial optimization. In this step all inclusion statements are replaced by selection statements. Then the inner term optimization, which handles statements within single terms of the conjunctive normal form, and the cross term optimization, which deals with several terms of the conjunctive normal form, are iteratively applied to selection statements. The algorithm stops if no optimization step can be applied anymore. Replacement rules are given in the form $LHS \Longrightarrow RHS$. A rule can only be applied to a formula containing the left-hand side pattern LHS. This pattern is then replaced by the right-hand side pattern RHS. The idea of expressing this step of feature constraint propagation as replacement rules and the notation are affected by algebraic graph and high-level transformation [EEPT06]. Let $cfid_R, cfid_P, cfid_C, cfid_{C,1}, cfid_{C,2}, ...$ be configured feature identifiers and φ be an arbitrary feature constraint. We assume that $cfid_R$ points to a root feature and that $cfid_P$ is the parent of $cfid_C, cfid_{C,1}, cfid_{C,2}, ...$ Note that we have to respect the parent relationship on instance level and not the one on feature level here, according to Requirement Req-13.

Initial optimization. In the initial step of the inclusion-to-selection algorithm, inclusion statements for configured feature identifiers pointing to root features are replaced by selection statements. This is expressed by the following rule, which is applied to all configured feature identifiers pointing to root features.

$$cfid_R\texttt{[1]} \Longrightarrow cfid_R\texttt{[+]} \tag{5.8}$$

The correctness of this step follows directly from Fact 5.1.2.

Inner term optimization with respect to a configured feature identifier $cfid_P$. The inner term optimization regards only a single term of a conjunctive normal form. It consists of several layered replacement rules. In the first step, rules

of the first layer are applied to a selection statement for the given configured feature identifier $cfid_P$ as often as possible. Rules of this layer replace inclusion statements for children of $cfid_P$ by selection statements. The rule of the second layer is applied to $cfid_P$ subsequently. It deletes the selection statement for $cfid_P$ if it became redundant by the application of layer 1 rules.

Layer 1 rules ("selection rules"):

$$\neg cfid_C\,[1] \vee \neg cfid_P\,[+] \Longrightarrow \neg cfid_C\,[+] \vee \neg cfid_P\,[+] \tag{5.9}$$
$$cfid_C\,[1] \vee \neg cfid_P\,[+] \Longrightarrow cfid_C\,[+] \vee \neg cfid_P\,[+] \tag{5.10}$$

Layer 2 rules ("deletion rules"):

$$\neg cfid_C\,[+] \vee \neg cfid_P\,[+] \Longrightarrow \neg cfid_C\,[+] \tag{5.11}$$

The correctness of rule 5.9 follows from Fact 5.1.3 since $(\neg cfid_C\,[1] \vee \neg cfid_P\,[+]) \equiv \neg(cfid_C\,[1] \wedge cfid_P\,[+])$. This rule does not delete the redundant statement $\neg cfid_P\,[+]$ on the right-hand side because the term could contain more children of $cfid_P$, which also have to be processed. In order to illustrate the correctness of rule 5.10, we convert the constraint $(cfid_C\,[1] \vee \neg cfid_P\,[+])$ into the equivalent constraint $\neg(\neg cfid_C\,[1] \wedge cfid_P\,[+])$. The selection of $cfid_P$, the parent of $cfid_C$, is a necessary condition for the fulfillment of the inner conjunction. Hence, we can replace $cfid_C\,[1]$ by $cfid_C\,[+]$ (even though $cfid_C$ is negated) and obtain the formula $\neg(\neg cfid_C\,[+] \wedge cfid_P\,[+])$. The statement $cfid_P\,[+]$ cannot be removed in this case since it is not redundant. The formula $\neg(\neg cfid_C\,[+] \wedge cfid_P\,[+])$ can be converted into the right-hand side of the rule. Rule 5.11 is correct since $cfid_C\,[+] \wedge cfid_P\,[+]$ is equivalent to $cfid_C\,[+]$. This holds because the selection of the child already implies the selection of the parent. The removal of the statement $\neg cfid_P\,[+]$ is only allowed if the term contains a negated selection statement for a child.

Cross term optimization with respect to a configured feature identifier $cfid_P$. The cross term optimization takes several terms of a conjunctive normal form into account. It consists, analogously to the inner term optimization, of several layered replacement rules. Layer 1 rules are rules that replace inclusion statements for children of $cfid_P$ by selection statements. They are applied as often as possible to $cfid_P$. Then the deletion rules of layer 2 are applied in order to remove redundant selection statements for $cfid_P$.

Layer 1 rules ("selection rules"):

$$(cfid_C\,[1] \vee \varphi) \wedge (cfid_P\,[+] \vee \varphi) \Longrightarrow (cfid_C\,[+] \vee \varphi) \wedge (cfid_P\,[+] \vee \varphi) \tag{5.12}$$
$$(\neg cfid_C\,[1] \vee \varphi) \wedge (cfid_P\,[+]) \Longrightarrow (\neg cfid_C\,[+] \vee \varphi) \wedge (cfid_P\,[+]) \tag{5.13}$$
$$(cfid_C\,[1] \vee \varphi) \wedge (cfid_P\,[+]) \Longrightarrow (cfid_C\,[+] \vee \varphi) \wedge (cfid_P\,[+]) \tag{5.14}$$

Layer 2 rules ("deletion rules"):

$$(cfid_C\texttt{[+]} \vee \varphi) \wedge (cfid_P\texttt{[+]} \vee \varphi) \Longrightarrow (cfid_C\texttt{[+]} \vee \varphi) \tag{5.15}$$

$$\left(\begin{array}{r}(cfid_{C,1}\texttt{[+]} \vee cfid_{C,2}\texttt{[+]} \vee ...) \\ \wedge (cfid_P\texttt{[+]})\end{array}\right) \Longrightarrow (cfid_{C,1}\texttt{[+]} \vee cfid_{C,2}\texttt{[+]} \vee ...) \tag{5.16}$$

The correctness of rule 5.12 is implied by Fact 5.1.3 since $(cfid_C\texttt{[1]} \vee \varphi) \wedge (cfid_P\texttt{[+]} \vee \varphi)$ is equivalent to $(cfid_C\texttt{[1]} \wedge cfid_P\texttt{[+]}) \vee \varphi$. The redundant selection statement for $cfid_P$ is not removed by this rule because there is the possibility that more terms containing children of $cfid_P$ exist, which also have to be processed. The rules 5.13 and 5.14 are correct since the selection of $cfid_P$ is a necessary condition for the fulfillment of the left-hand side formulae. We can therefore replace inclusion statements for the child $cfid_C$ of the configured feature identifier $cfid_P$ by selection statements. However, the selection statement for $cfid_P$ may not be removed. The deletion rule 5.15 of selection rule 5.12 removes the redundant term $(cfid_P\texttt{[+]} \vee \varphi)$. This rule is correct since $(cfid_C\texttt{[+]} \vee \varphi) \wedge (cfid_P\texttt{[+]} \vee \varphi) \equiv (cfid_C\texttt{[+]} \wedge cfid_P\texttt{[+]}) \vee \varphi \equiv cfid_C\texttt{[+]} \vee \varphi$. Rule 5.16 is the deletion rule of selection rule 5.14. If a term contains solely selection statements for children of $cfid_P$, then the selection statement for $cfid_P$ itself can be removed since it is already covered by the fact that at least one of its children has to be selected.

Algorithm 5.2 shows the pseudo-code of the inclusion-to-selection algorithm. It iterates over all statements of the formula (line 2), replaces all inclusion statements for configured feature identifiers pointing to root features by selection statements and collects these statements in a queue. Subsequently, as long as the queue is not empty (line 8), a new selection statement is removed. If this statement is negated, the inner term optimization (line 11) is accomplished for this statement. Otherwise, the cross term optimization (line 13) is used for this statement. Both optimizations are implemented as functions and get the statement as parameter. Each of these functions applies the above described replacement rules only to the given statement. All statements getting selected by these functions are then returned and added to the queue of new selection statements. This approach is comparable to a breadth-first search in a graph in contrast to a depth-first search.

The inner term optimization (Algorithm 5.3) gets a new selection statement as input and iterates over all statements of its term (line 4). If a child of the configured feature identifier of the input statement is found (line 5) and it is not already selected (line 6), it gets selected and the new selection statement is added to the queue of new selection statements. If the child is negated (line 10), the input statement is marked to be deleted. Before the function returns all new selection statements, the input statement is deleted if it is marked (line 15).

The algorithm of the cross term optimization (Algorithm 5.4) is more complicated than the algorithm of the inner term optimization. It iterates over all terms of the formula (line 4). If the term of the input statement is a singleton[2] (line 5), every child of the configured feature identifier of the input statement gets selected within a loop (line 7). If a term contains solely negated statements of children of

[2] i.e. it contains only one single statement

Algorithm 5.2: Inclusion-to-selection algorithm

Data: feature constraint φ in (minimal) conjunctive normal form without selection statements
Result: optimized feature constraint in (minimal) conjunctive normal form with selection statements

```
1  newSelectionStatements ← ∅ // queue of new selection statements
2  foreach statement s of φ do
       /* a statement of a formula is a negated or non-negated
          configured feature identifier with a cardinality        */
3      if getCfid(s) points to a root feature then
4          setSelection(s) // rule 5.8
5          newSelectionStatements.push(s)
6      end
7  end
8  while not newSelectionStatements.isEmpty do
9      s ← newSelectionStatements.pop
10     if isNegated(s) then
11         newSelectionStatements.pushAll(innerTermOptimization(s))
           // see Algorithm 5.3
12     else
13         newSelectionStatements.pushAll(crossTermOptimization(s))
           // see Algorithm 5.4
14     end
15 end
```

this configured feature identifier (line 18), the term of the input statement is marked to be deleted. In the other case, if the term of the input statement is not a singleton (line 19), the algorithm iterates over all statements of the term (line 20) of the outer loop (line 4). If the term contains a negated statement for a child of the configured feature identifier of the input statement (lines 21 and 22), it is checked whether the remaining parts of the term (without this statement) are equivalent[3] to the term of the input statement (line 23). Then the found statement is (if it is not yet a selection statement) replaced by a selection statement, added to the queue of new selection statements and the term of the input statement is marked to be deleted. Before the algorithm returns all new selection statements for further processing, the term of the input statement is deleted if it is marked (line 35).

Runtime Approximation

Let *MinCStatements* be all statements and *MinCTerms* be all terms of the minimized constraint. Since we approximate the worst-case runtime, we use the number of all statements of the constraint as upper bound for all statements of a term.

[3]This does not mean semantic equivalence. It is checked whether both terms contain exactly the same statements.

Algorithm 5.3: Inclusion-to-selection algorithm: inner term optimization

```
   Function: innerTermOptimization
   Input: negated statement s of term t of constraint φ
   Output: queue of newly selected statements newSelectionStatements
 1 begin
 2     newSelectionStatements ← ∅ // queue of new selection statements
 3     delete ← false
 4     foreach statement s' ≠ s of t do
 5         if getCfid(s') is a child of getCfid(s) then
 6             if not isSelection(s') then
 7                 setSelection(s') // rules 5.9 and 5.10
 8                 newSelectionStatements.push(s')
 9             end
10             if isNegated(s') then
11                 delete ← true // see left-hand side of rule 5.11
12             end
13         end
14     end
15     if delete then
16         delete s of t // rule 5.11
17     end
18     return newSelectionStatements
19 end
```

The inner term optimization (Algorithm 5.3) contains a loop over all statements of a term in line 4 and the loop's body has a constant runtime. Therefore, the algorithm has a runtime of $\mathcal{O}(|MinCStatements|)$.

The cross term optimization (Algorithm 5.4) iterates over all terms of the minimized constraint in line 4 (runtime $\mathcal{O}(|MinCTerms|)$). In order to approximate the worst-case runtime of the conditional statement in line 5 within the loop, we have to consider only the case with the maximum runtime. The if-case contains a loop over all statements of the term in line 7 with a runtime of $\mathcal{O}(|MinCStatements|)$. The runtime of this loop's body is constant. Consequently, the if-case of the conditional statement in line 5 has a runtime of $\mathcal{O}(|MinCStatements|)$. The else-case of the conditional statement in line 5 also contains a loop over all statements of the term in line 20 with a runtime of $\mathcal{O}(|MinCStatements|)$. However, the body of this loop contains a check if a term is equivalent to another term in line 23. This operation iterates over all statement of the first term and checks if every statement is also contained in the second term. This means that the operation in line 23 iterates for every statement of the first term over all statements of the second term and has therefore a worst-case runtime of $\mathcal{O}(|MinCStatements|^2)$. Consequently, the else-case of the conditional statement in line 5 has a worst-case runtime of $\mathcal{O}(|MinCStatements|^3)$. Altogether, the cross term optimization has a worst-case runtime of $\mathcal{O}(|MinCTerms| * |MinCStatements|^3)$.

Algorithm 5.4: Inclusion-to-selection algorithm: cross term optimization

```
   Function: crossTermOptimization
   Input: non-negated statement s of term t of constraint φ
   Output: queue of newly selected statements newSelectionStatements
 1 begin
 2     newSelectionStatements ← ∅ // queue of new selection statements
 3     delete ← false
 4     foreach term t' ≠ t of φ do
 5         if isSingleton(t) then
 6             onlyNonNegatedChildren ← true
 7             foreach statement s' of t' do
 8                 if getCfid(s') is a child of getCfid(s) then
 9                     if not isSelection(s') then
10                         setSelection(s') // rules 5.13 and 5.14
11                         newSelectionStatements.push(s')
12                     end
13                     if isNegated(s') then onlyNonNegatedChildren ← false
14                 else
15                     onlyNonNegatedChildren ← false
16                 end
17             end
18             if onlyNonNegatedChildren and t' contains at least one statement
                   then delete ← true // see left-hand side of rule 5.16
19         else // term is not a singleton
20             foreach statement s' of t' do
21                 if not isNegated(s') then
22                     if getCfid(s') is a child of getCfid(s) then
23                         if t' without s' is equivalent to t without s then
24                             if not isSelection(s') then
25                                 setSelection(s') // rule 5.12
26                                 newSelectionStatements.push(s')
27                             end
28                             delete ← true
                               // see left-hand side of rule 5.15
29                         end
30                     end
31                 end
32             end
33         end
34     end
35     if delete then delete t of φ // rules 5.15 and 5.16
36     return newSelectionStatements
37 end
```

In order to approximate the runtime of the overall inclusion-to-selection algorithm (see Algorithm 5.2), we have to consider the runtimes of the loops in lines 2 and 8. The loop in line 2 iterates over all statements of the constraint (runtime $\mathcal{O}(|MinCStatements|)$) and its body has a constant runtime. The while-loop in line 8 iterates, in the worst-case, over all statements of the constraint (runtime $\mathcal{O}(|MinCStatements|)$) as well. It contains a conditional statement in line 10. We have already shown that the runtime of the cross term optimization $\mathcal{O}(|MinCTerms| * |MinCStatements|^3)$ is greater than the one of the inner term optimization $\mathcal{O}(|MinCStatements|)$. We therefore have to consider only the else-case of the conditional statement in line 10. Consequently, the while-loop in line 8 has a worst-case runtime of $\mathcal{O}(|MinCTerms| * |MinCStatements|^4)$. We obtain an overall runtime of $\mathcal{O}(|MinCStatements|) + \mathcal{O}(|MinCTerms| * |MinCStatements|^4) = \mathcal{O}(|MinCTerms| * |MinCStatements|^4)$ for the inclusion-to-selection algorithm.

Example

We apply the inclusion-to-selection algorithm to the resulting formulae of the previous step. The numbers on the right side denote the applied replacement rules.

$\neg$`Convertible[1]` $\vee$ $\neg$`ComfortPlus[1]` $\vee$ $\neg$`Comfort[1]` $\vee$ $\neg$`Model[1]`
$\Longrightarrow$ $\neg$`Convertible[1]` $\vee$ $\neg$`ComfortPlus[1]` $\vee$ $\neg$`Comfort[1]` $\vee$ $\neg$`Model[+]` 5.8
$\Longrightarrow$ $\neg$`Convertible[+]` $\vee$ $\neg$`ComfortPlus[1]` $\vee$ $\neg$`Comfort[1]` $\vee$ $\neg$`Model[+]` 5.9
$\Longrightarrow$ $\neg$`Convertible[+]` $\vee$ $\neg$`ComfortPlus[1]` $\vee$ $\neg$`Comfort[+]` $\vee$ $\neg$`Model[+]` 5.9
$\Longrightarrow$ $\neg$`Convertible[+]` $\vee$ $\neg$`ComfortPlus[1]` $\vee$ $\neg$`Comfort[+]` 5.11
$\Longrightarrow$ $\neg$`Convertible[+]` $\vee$ $\neg$`ComfortPlus[+]` $\vee$ $\neg$`Comfort[+]` 5.9
$\Longrightarrow$ $\neg$`Convertible[+]` $\vee$ $\neg$`ComfortPlus[+]` 5.11

$\neg$`Comfort[1]` $\vee$ $\neg$`Model[1]` $\vee$ $\neg$`USA[1]` $\vee$ $\neg$`Market[1]`
$\Longrightarrow$ $\neg$`Comfort[1]` $\vee$ $\neg$`Model[+]` $\vee$ $\neg$`USA[1]` $\vee$ $\neg$`Market[1]` 5.8
$\Longrightarrow$ $\neg$`Comfort[1]` $\vee$ $\neg$`Model[+]` $\vee$ $\neg$`USA[1]` $\vee$ $\neg$`Market[+]` 5.8
$\Longrightarrow$ $\neg$`Comfort[+]` $\vee$ $\neg$`Model[+]` $\vee$ $\neg$`USA[1]` $\vee$ $\neg$`Market[+]` 5.9
$\Longrightarrow$ $\neg$`Comfort[+]` $\vee$ $\neg$`USA[1]` $\vee$ $\neg$`Market[+]` 5.11
$\Longrightarrow$ $\neg$`Comfort[+]` $\vee$ $\neg$`USA[+]` $\vee$ $\neg$`Market[+]` 5.9
$\Longrightarrow$ $\neg$`Comfort[+]` $\vee$ $\neg$`USA[+]` 5.11

5.1.6 Step 6: Extract Feature Links

According to Requirement Req-1, feature constraint propagation provides the functionality to extract feature links out of constraints. We distinguish three types of feature links (cf. Section 2.2): `needs`, `excludes` and `alternative`. Every term of the conjunctive normal form resulting from Step 5 is analyzed and converted into an equivalent feature link if possible. This step is partly inspired by the feature model extraction approach presented in [CW07]. The extraction algorithm consists of four consecutive steps, which are described in the following.

Since the algorithm is a pure syntactic algorithm and does not analyze the semantics of the formula, it is highly dependent on the structure of the formula, i.e. on the results of the last two steps of feature constraint propagation (minimization and lifting inclusion to selection statements). Thus, it does not necessarily produce the optimal result. This means that not every possible feature link can be extracted if the structure of the formula is inappropriate.

Identify candidates for feature links. In the first step of the extraction algorithm all terms are analyzed. A term has to fulfill the following conditions to be a candidate for being converted into a feature link.

1. it has to contain at least two statements[4]
2. it may not contain any inclusion statements
3. it may not contain statements referring to instances of cloned features or to their successors

№ 1 results from the fact that feature links are always defined between two or more features. № 2 has to hold since feature links are evaluated with respect to selection. The rationale for № 3 is Requirement Req-12.

All terms fulfilling these conditions are classified into the following categories.

- *candidates for* `needs` *feature links:*
  all terms containing negated and non-negated statements

$$(\neg cfid_1\,[+] \vee \neg cfid_2\,[+] \vee ... \vee cfid_i\,[+] \vee cfid_{i+1}\,[+] \vee ...)$$

- *candidates for* `excludes` *feature links:*
  all terms containing solely negated statements

$$(\neg cfid_1\,[+] \vee \neg cfid_2\,[+] \vee ...)$$

- *candidates for* `alternative` *feature links:*
  all terms containing solely non-negated statements

$$(cfid_1\,[+] \vee cfid_2\,[+] \vee ...)$$

Lift candidates to feature level. In this step all candidates for the extraction of feature links are lifted from instance to feature level. This means that all configured feature identifiers of these terms are replaced by the features they are pointing to. This seems trivial since terms that contain configured feature identifiers pointing to cloned features have already been eliminated in the previous step. However, a term could contain configured feature identifiers pointing to successors of features of which other features inherit. In this case, a simple replacement of the configured feature identifier by the corresponding feature would change the semantics of the

[4]If multi feature links (cf. Section 2.2) shall not be allowed, “at least two” has to be changed to “exactly two”.

constraint, as already illustrated in Chapter 4. This results from the fact that these features have more than one form on instance level, i.e. there are different configured feature identifiers pointing to them. According to Requirement Req-15, all possible forms of inherited features on instance level have to be considered in order to replace them by features. We therefore calculate all possible forms for concerned features and apply the following replacement rules, which preserve the semantics of the constraint. Let `F` and `I` be features such that `I` is, in contrast to `F`, a successor of a feature of which another feature inherits, i.e. there is a predecessor `P` of `I` that is inherited by another feature `G`. Furthermore, let $cfid_F$ be a configured feature identifier pointing to `F` (there is exactly one) and $cfid_{I,1}, cfid_{I,2}, ...$ be all existing configured feature identifiers (not only configured feature identifiers contained in the constraint) pointing to `I`.

$$cfid_F\,[+] \Longrightarrow F \tag{5.17}$$

$$(cfid_{I,1}\,[+] \vee cfid_{I,2}\,[+] \vee ...) \Longrightarrow I \tag{5.18}$$

$$(\neg cfid_{I,1}\,[+] \vee \varphi) \wedge (\neg cfid_{I,2}\,[+] \vee \varphi) \wedge ... \Longrightarrow (\neg I \vee \varphi) \tag{5.19}$$

Rule 5.17 expresses the direct replacement of configured feature identifiers by the feature they are pointing to. This rule is only applicable to features which are not successors of features that are inherited by other features since the configured feature identifier is unique in this case, i.e. there does not exist another configured feature identifier pointing to the same feature. Rule 5.18 describes that a disjunction of all existing configured feature identifiers pointing to `I` can be replaced by the feature `I`. The correctness of this rule can be seen directly since the statement `I` on feature level is fulfilled if a configured feature identifier pointing to `I` is selected. Note that rule 5.17 could also be seen as a special case of rule 5.18, in which only one configured feature identifier pointing to `I` exists. Nevertheless, we formulated rule 5.17 separately to underline that this step is really trivial if no inheritance is included. In order to illustrate the correctness of rule 5.19, we convert $(\neg cfid_{I,1}\,[+] \vee \varphi) \wedge (\neg cfid_{I,2}\,[+] \vee \varphi) \wedge ...$ into the equivalent statement $(\neg cfid_{I,1}\,[+] \wedge \neg cfid_{I,2}\,[+] \wedge ...) \vee \varphi$. This is equivalent to $\neg(cfid_{I,1}\,[+] \vee cfid_{I,2}\,[+] \vee ...) \vee \varphi$. According to rule 5.18, we obtain $\neg I \vee \varphi$.

All these rules are applied to all in the previous step identified candidates as long as possible. Consequently, the lifting algorithm can deal with constraints containing any number of inherited features. All terms that cannot be lifted completely (i.e. terms with remaining configured feature identifiers) or that now contradict to criterion Nº 1 (mentioned in the previous step of the extraction algorithm) are removed from the corresponding set of candidates for feature link extraction. The lifting operation is revoked for them. With the lifting operation we achieve the following goals: (a) the sets of candidates are narrowed by rejecting candidates which cannot be converted into feature links because of inheritance (this meets Requirement Req-15) and (b) all remaining candidates are expressed by formulae on feature level.

The algorithms implementing the application of the defined replacement rules are straightforward. Exemplarily, the implementation is shown in Algorithm 5.5 for replacement rule 5.19 in very abstract pseudo-code. With respect to the implementation, this is the most complicated replacement rule because several iterations over

all statements are required. The algorithm starts with an iteration over all statements of all candidates and looks for negated statements of a concrete form of a given inherited feature F (line 3). If such a statement s is found, it is stored in a collection and an inner loop is used to iterate over all possible forms of F (line 5). Within this loop, negated statements for the actual form are searched in other candidates (line 7). For every negated statement of a form, it is checked whether the term of the corresponding statement contains exactly the same statements[5] as the term of the statement s (line 8). This equivalence check is the same as in Algorithm 5.4. If a term fulfills this condition, the new statement is stored and the algorithm continues with the next form. If such terms are found for all possible forms of F (line 18), the replacement defined in rule 5.19 is applied to all stored statements (resp. terms).

Extract `needs` and `excludes` feature links. In this step of the extraction algorithm all candidates for `needs` and `excludes` feature links are converted into feature links according to the following listing. This means that they are removed from the feature constraint and added as feature links. Note that all remaining candidates have already been converted into formulae on feature level in the previous step.

- candidates for `needs` feature links

  $$(\neg F_1 \vee \neg F_2 \vee ... \vee F_i \vee F_{i+1} \vee ...)$$

  are converted into

  $$\text{«}[F_1, F_2, ...] \texttt{ needs } [F_i, F_{i+1}, ...]\text{»}$$

- candidates for `excludes` feature links

  $$(\neg F_1 \vee \neg F_2 \vee ...)$$

  are converted into

  $$\text{«}\texttt{excludes } [F_1, F_2, ...]\text{»}$$

The correctness of this conversion follows from pure propositional logic: $(\neg F_1 \vee \neg F_2 \vee ... \vee F_i \vee F_{i+1} \vee ...) \equiv \neg(F_1 \wedge F_2 \wedge ...) \vee (F_i \vee F_{i+1} \vee ...) \equiv (F_1 \wedge F_2 \wedge ...) \rightarrow (F_i \vee F_{i+1} \vee ...) \equiv$ «$[F_1, F_2, ...]$ `needs` $[F_i, F_{i+1}, ...]$» and $(\neg F_1 \vee \neg F_2 \vee ...) \equiv \neg(F_1 \wedge F_2 \wedge ...) \equiv$ «`excludes` $[F_1, F_2, ...]$».

Extract `alternative` feature links. The extraction of `alternative` feature links is more complicated than the extraction of `needs` and `excludes` links because they cannot be expressed by a single term in the conjunctive normal form.

- candidates for `alternative` feature links

  $$(F_1 \vee F_2 \vee ...)$$

  together with an `excludes` feature link for every pair of contained features

  $$\text{«}F_1 \texttt{ excludes } F_2\text{»}, \text{«}F_1 \texttt{ excludes } ...\text{»}, \text{«}F_2 \texttt{ excludes } ...\text{»}, ...$$

  are converted into

  $$\text{«}\texttt{alternative } [F_1, F_2, ...]\text{»}$$

[5] the different forms of F are ignored during this check

Algorithm 5.5: Extract feature links: part of the lifting algorithm (implements only the application of rule 5.19)

Data: candidates for feature link extraction
Data: any inherited feature F
Data: all forms of the feature F on instance level
Result: candidates for feature link extraction after the application of rule 5.19 to F

```
1  success ← false
2  x ← first form of F
3  foreach statement s = ¬x[+] in a candidate c do
4  |  statementsToReplace ← {s} // the concrete occurrences to replace
5  |  foreach form x' ≠ x of F do
6  |  |  success ← false
7  |  |  foreach statement s' = ¬x'[+] in a candidate c' ≠ c do
8  |  |  |  if c' without x' is equivalent to c without x then
9  |  |  |  |  statementsToReplace.push(s')
10 |  |  |  |  success ← true
11 |  |  |  |  break // continue with next form of F
12 |  |  |  end
13 |  |  end
14 |  |  if not success then // no equivalent term for form x' of F
15 |  |  |  break // continue with next occurrence of x
16 |  |  end
17 |  end
18 |  if success then // rule 5.19 can be applied
19 |  |  s ← statementsToReplace.pop
   |  |  // s = ¬x[+] with x being a form of F
20 |  |  replace s by ¬F in candidate
21 |  |  remove all terms of statements in statementsToReplace from constraint
22 |  end
23 end
```

This conversion is correct because an `alternative` feature link between the features $F_1, F_2, ...$ is equivalent to the disjunction of all features $F_1 \vee F_2 \vee ...$ ("at least one of the features has to be selected") together with their pairwise exclusiveness «F_1 `excludes` F_2», «F_1 `excludes` ...», «F_2 `excludes` ...» and so on ("if one the features is selected, then all others may not be selected"). Conversion of a candidate means that the candidate as well as the comprised `excludes` feature links are removed and the `alternative` feature link is added. All candidates which cannot be converted into `alternative` feature links because of absent `excludes` feature links remain as feature constraints. The lifting operation is revoked for them.

The final result of feature constraint propagation is a set of feature links (all extracted feature links) and a set of feature constraints (all remaining terms of the conjunctive normal form after the extraction of feature links).

Runtime Approximation

We assume *SMinCStatements* to be all statements and *SMinCTerms* to be all terms of the constraint after the application of the inclusion-to-selection algorithm. Moreover, let *Features* be all features of the source feature model. Since we approximate the worst-case runtime, we use the number of all statements of the constraint as upper bound for all statements of a term.

Identify candidates for feature links. The identification of candidates for feature links requires only one iteration over all statements *SMinCStatements*. This leads to a runtime of $\mathcal{O}(|SMinCStatements|)$.

Lift candidates to feature level. In the worst-case all terms were identified to be candidates for the extraction and have to be processed in lifting operation. The most expensive part of the lifting algorithm is the application of replacement rule 5.19. This part of the algorithm for a feature F is depicted in very abstract pseudo-code in Algorithm 5.5. The outer loop in line 3 iterates over all statements of all candidates (runtime $\mathcal{O}(|SMinCStatements|)$). If a negated form of the feature F is found, the inner loop in line 5 iterates over all possible forms of F. We denote the number of all possible forms of feature F with $|Forms|$. Consequently, the loop in line 5 has the runtime $\mathcal{O}(|Forms|)$. In line 7 is another loop that iterates over all statements again (runtime $\mathcal{O}(|SMinCStatements|)$). It contains the check if two terms contain the same statements (the order does not matter) in line 8. This operation has already been used in line 23 of Algorithm 5.4. It has a worst-case runtime of $\mathcal{O}(|SMinCStatements|^2)$. The application of the replacement (the if-case of the conditional statement in line 18) has a runtime of $\mathcal{O}(|Forms|)$ since it contains an iteration over all statements to be replaced (resp. terms to be removed) in line

21. Summarizing, the depicted algorithm has the following worst-case runtime.

$$
\begin{aligned}
&\mathcal{O}(|SMinCStatements|) \\
&\quad * (\mathcal{O}(|Forms|) * \mathcal{O}(|SMinCStatements|) * \mathcal{O}(|SMinCStatements|^2) + \mathcal{O}(|Forms|)) \\
&= \mathcal{O}(|SMinCStatements|) * (\mathcal{O}(|Forms| * |SMinCStatements|^3) + \mathcal{O}(|Forms|)) \\
&= \mathcal{O}(|SMinCStatements|) * \mathbf{max}((\mathcal{O}(|Forms| * |SMinCStatements|^3), \mathcal{O}(|Forms|))) \\
&= \mathcal{O}(|SMinCStatements|) * \mathcal{O}(|Forms| * |SMinCStatements|^3) \\
&= \mathcal{O}(|SMinCStatements|^4 * |Forms|)
\end{aligned}
$$

In the worst-case the replacement rule 5.19 has to be applied to all features *Features*. This leads to a worst-case runtime of $\mathcal{O}(|Features| * |SMinCStatements|^4 * |Forms|)$ for the lifting step.

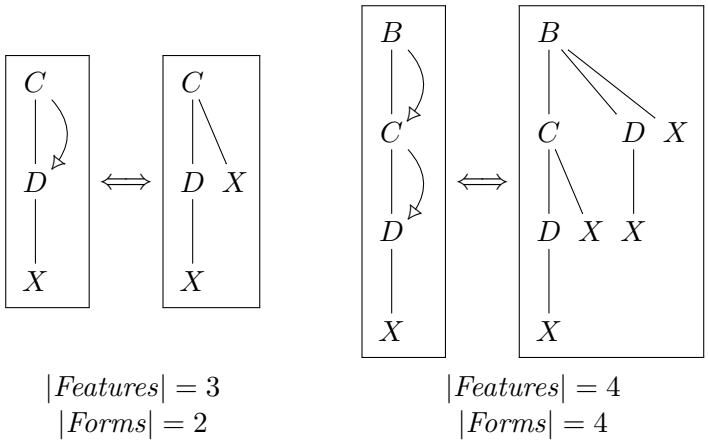
The number of forms of a feature on instance level ($|Forms|$) is usually very small in practice since most feature models with feature inheritance contain only a few inheritance relationships. However, because we approximate the worst-case runtime, we are interested in the maximum number of forms of a feature regarding the feature count $|Features|$. A feature can have a maximum of $2^{|Features|-2}$ forms on instance level. Figure 5.2 illustrates this correlation between the feature count $|Features|$ and the maximum number of forms $|Forms|$ of a feature X. Because of this correlation, the worst-case runtime of the lifting step can be expressed by $\mathcal{O}(|Features| * |SMinCStatements|^4 * 2^{|Features-2|})$. It is exponential dependent on the feature count of the model. Although this worst-case is extremely unlikely, this runtime approximation shows that the occurrence of many inheritance relationships in the source model can influence the runtime of the lifting step drastically. Without inheritance in the source model this step only has a runtime of $\mathcal{O}(|SMinCStatements|)$ (only rule 5.17 has to be applied).

Extract `needs` and `excludes` feature links. In order to extract `needs` and `excludes` feature links out of the candidates, only one iteration over all statements is required. This leads to a runtime of $\mathcal{O}(|SMinCStatements|)$.

Extract `alternative` feature links. The algorithm for the extraction of `alternative` feature links checks for every pair of features of a candidate if there is an `excludes` feature link between them. For this purpose, the algorithm contains two nested loops, both iterating over all statements of a candidate (approximated with the upper bound $\mathcal{O}(|SMinCStatements|^2)$). In every iteration, a pair of features is picked and the algorithm looks for extracted `excludes` feature links between these features. The number of extracted `excludes` feature links can be estimated as the number of terms $|SMinCTerms|$. The mentioned operations have to be applied to all candidates for `alternative` feature links. The number of terms $|SMinCTerms|$ limits the maximum number of candidates for `alternative` feature links. We obtain a worst-case runtime of $\mathcal{O}(|SMinCTerms|^2 * |SMinCStatements|^2)$ for the extraction of `alternative` feature links.

Example

The application of the extraction algorithm to our example is quite trivial since it does not contain feature inheritance. Consider the already calculated formu-



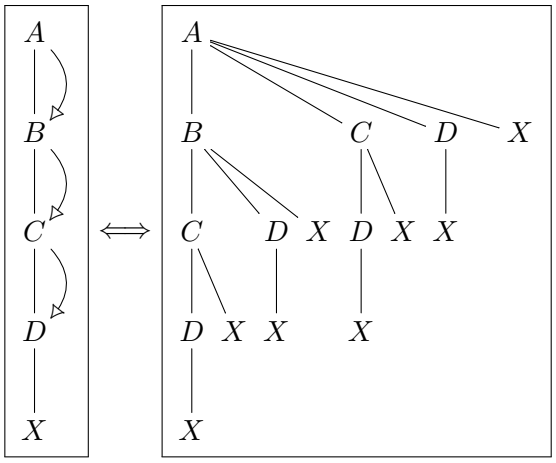


Figure 5.2: Correlation between feature count and the maximum number of forms of a feature X

lae ¬Convertible[+] ∨ ¬ComfortPlus[+] and ¬Comfort[+] ∨ ¬USA[+]. Both of them are identified as candidates for excludes feature links because they contain only negated statements. The lifting of these formulae to feature level leads to ¬Convertible ∨ ¬ComfortPlus and ¬Comfort ∨ ¬USA by repeated application of the replacement rule 5.17. Finally, the feature links «Convertible excludes ComfortPlus» and «Comfort excludes USA» are extracted. This is the final result of the application of feature constraint propagation to the example in Figure 2.5.

In a practical setting this result would surely mean that there are errors in the model because a technical dependency affects the variability of the overall product line. However, we do not want to discuss this here because this section only introduces the concept of feature constraint propagation.

Summary

The presented technique of feature constraint propagation consists of six consecutive steps and allows to propagate arbitrary target-side constraints along a configuration link to source-side constraints. Its centerpiece, the transformation (including the calculation of the reverse mappings), is based on the replacement of variables in target-side constraints by source-side constraints. In addition, there is a preprocessing (the expansion of the configuration link) and a postprocessing (the minimization, the lifting and the extraction of feature links). The postprocessing aims to restructure the constraint such that it is "as understandable as possible". This means for a resulting constraint that (1) it should be as short as possible, (2) it should contain as less as possible inclusion statements and (3) all parts that can be expressed as feature links should be extracted. The described postprocessing does not necessarily produce optimal results with respect to these three goals.

All steps of feature constraint propagation are required to meet the requirements presented in Chapter 4. We have already mentioned the fulfillment of the individual requirements in the description above. Table 5.2 summarizes the assignment of requirements to steps of the propagation and confirms that all requirements are met by the technique. The first requirements of all steps in the table are written in italics because they imply the necessity of the corresponding step. The other requirements are subordinate requirements referring to specific steps of feature constraint propagation.

We approximated the runtimes for all steps of feature constraint propagation. All steps can be conducted in polynomial time, i.e. they are in the complexity class P, besides the minimization of the constraint (Step 4) and the lifting of constraints from instance to feature level (a part of Step 6). We argued that the worst-case of the lifting step is extremely unlikely and the runtime of this step is, in most cases, appropriate. The most expensive step is thus the minimization of the constraint. Our approach to manage this problem is the use of a heuristic minimization algorithm. Nevertheless, this step remains crucial for the overall runtime.

Preprocessing	Transformation		Postprocessing		
Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 5**	**Step 6**
Req-2	*Req-8*		*Req-3*	*Req-4*	*Req-1*
	Req-5			Req-13	Req-12
	Req-6				Req-15
	Req-7				
	Req-9				
	Req-10				
	Req-11				
	Req-14				

Table 5.2: Assignment of requirements to individual steps of feature constraint propagation.

5.2 Advanced Considerations on Feature Constraint Propagation

There are several interesting possibilities to extend feature constraint propagation. Some of them are combinations with existing techniques and approaches, others refer to different use cases or technical details of configuration links and yet others improve the applicability of the technique itself. This section provides a summary of several extensions, their motivations and basic ideas.

In order to keep the technique of feature constraint as simple and accessible as possible, the integration of these extensions is not part of this thesis.

5.2.1 Combining Propagation and Automated Feature Model Analyses

Automated feature model analyses deal with the fully automated extraction of information from feature models, e.g. the number of products of a feature model can be calculated. There are a lot of different operations, techniques and tools. A comprehensive literature review of them can be found in [BSRC10] and formal definitions for the operations are presented in [Ben07] (cf. Section 3.4). Some proposed analysis operations are a useful complement to feature constraint propagation. They are presented and discussed in the following.

Void Feature Model

This operation answers the question whether a feature model is void or not. A feature model is called *void* if it does not represent any products, i.e. there is no valid configuration. This operation is a useful addition to feature constraint propagation because a non-void feature model can become void by adding constraints. The technique of feature constraint propagation does not check which impacts the propagated constraints have to the source feature model. If the propagated constraint is `false`, the source feature model always becomes void and no further check is required. However, a propagated constraint which is not equivalent to `false` can

(a) Non-void model... (b) ...becomes void

Figure 5.3: A non-void feature model becomes void by adding a constraint

also lead to a void source feature model, as depicted in Figure 5.3. Therefore, the *void feature model* operation should be applied after the propagation in order to check whether the source feature model became void through the propagation. If this is the case, the overall variability specification contains an error (either in one of the feature models or in the configuration link).

Number of Products

This operation calculates the number of products represented by a feature model. This is an interesting operation for the combination with feature constraint propagation because it illustrates how much the set of valid source configurations is narrowed by the propagation. The number of products can be calculated before and after the propagation. The difference between these values shows how many configurations which were valid in the original source feature model lead to invalid target configurations. These configurations were identified to be invalid with respect to the overall product line by feature constraint propagation. In a hierarchically organized product line the number of products can be calculated for every level of abstraction and for every view on the variability after feature constraint propagation was applied.

Anomalies Detection

Feature models can contain several undesirable properties, e.g. redundancies or contradictions, called anomalies. The *anomalies detection* operations identify these anomalies and provide information about them. According to [BSRC10], there are five main types of anomalies introduced in the following.

- **Dead features.** Dead features are features that cannot appear in any valid configuration. Figure 5.4 depicts some examples for dead features (encircled). Although feature links are defined only between features `B` and `C`, another feature `D` can become dead.

- **Conditionally dead features.** Features being dead under certain circumstances (e.g. if another feature is selected or deselected) are called *conditionally dead*. Some examples are depicted in Figure 5.5. Conditionally dead features are encircled.

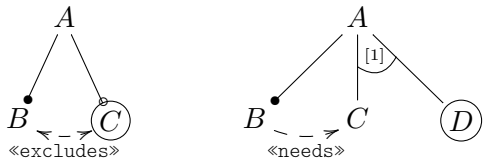


Figure 5.4: Examples for dead features

Figure 5.5: Examples for conditionally dead features

- **False optional features.** Optional features that are included in all valid configurations of a feature model are called *false optional*. Figure 5.6 shows some examples for false optional features (encircled).
- **Wrong cardinalities.** If a feature group has a cardinality that cannot be instantiated, this cardinality is called wrong. The same case can arise for cloned features. Figure 5.7 shows some examples. Wrong cardinalities can result from feature constraints and feature links together with feature groups or cloned features.
- **Redundancies.** A redundancy in a feature model is a semantic information that is modeled in multiple ways. Figure 5.8 shows some examples for redundant feature links.

All mentioned anomalies can arise in the source feature model of a configuration link through feature constraint propagation. Therefore, the application of several anomalies detection operations is a reasonable complement for feature constraint propagation. Most important operations are the detection of dead features and

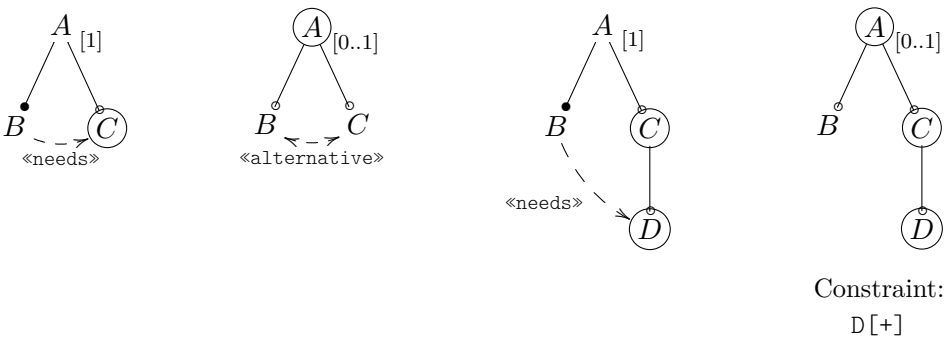


Figure 5.6: Examples for false optional features

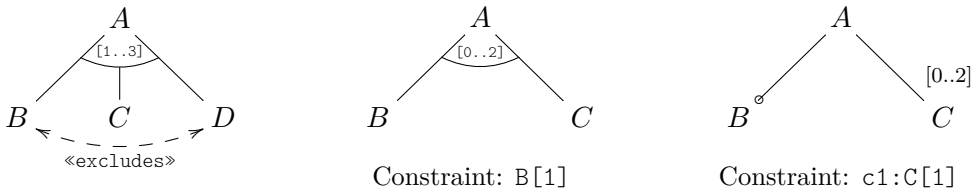


Figure 5.7: Examples for wrong cardinalities

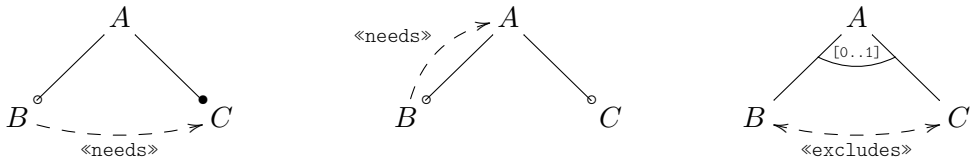


Figure 5.8: Examples for redundancies

redundant constraints. Redundant constraints should not be added to the source feature model by feature constraint propagation. The technique of feature constraint propagation itself does not check for redundancies. The application of automated redundancy checks after the propagation would allow to filter redundant constraints of the propagation easily. If one or more features become dead through the propagation, it is a strong indication for an error in the overall variability specification (either in a configuration link or in a feature model). The detection of false optional features, wrong cardinalities or conditionally dead features delivers some additional information about the model. If false optional features or wrong cardinalities are present after feature constraint propagation, some refactorings could be applied such that the corresponding cardinalities are narrowed to cardinalities that can be instantiated. It is important to make sure that the semantics of the feature model is not changed by this activity.

Explanations and Corrections

An *explanations* operation takes a feature model and an analysis operation, e.g. an anomalies detection operation, and returns the reasons for the result of the corresponding analysis operation, e.g. "Feature `C` is dead because of the `excludes` feature link between the mandatory feature `B` and `C`.". A *corrections* operation returns a set of corrective explanations for the same inputs, e.g. "Remove the `excludes` feature link between `B` and *C* or make `B` optional.". These operations could be used to present the local reasons and corrections for a newly formed anomaly after feature constraint propagation.

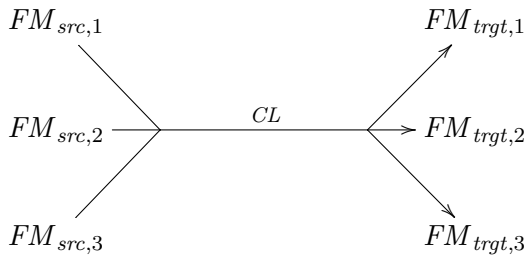


Figure 5.9: A configuration link with multiple source and target feature models

5.2.2 Propagation along Configuration Links with multiple Sources and Targets

Up to now, we have considered configuration links as 1:1 associations between feature models. Each configuration link had exactly one source and one target feature model. However, the concept of configuration link is also compatible with multiple source and target feature models [Rei08], as depicted in Figure 5.9. Configuration links with one source and multiple target feature models can simply be subdivided into separate configuration links. This splitting is not possible for configuration links with more than one source model. In this case, all source feature models could be merged into one feature model because we allow feature models with more than one parent feature. Note that this activity is also possible for multiple target feature models. Summarizing, configuration links with multiple source and target features can be reduced to configuration links from one source model to one target model. This means that the technique of feature constraint propagation can be used without any adaptations. However, the subdivision of a configuration link or the merging of feature models is not really feasible in practice. We therefore investigate feature constraint propagation for these configuration links without any subdivision or merging. The case of multiple target feature models is unproblematic for feature constraint propagation since the technique does not change the target model. All target feature models are internally processed as one feature model but they are not really merged together. In contrast, the case of multiple source feature models raises a problem: dependencies between features of different feature models can appear through the propagation. In this case, the source feature models cannot be used as single feature models anymore. It is not possible to avoid this situation because it arises from the given constraints and the configuration link. The negligence of these "cross-model" constraints would lead to the fact that the correctness of the target configurations derived through the application of the configuration link cannot be ensured anymore. An example is depicted in Figure 5.10. The propagated source-side feature link «`B excludes E`» (dotted in the figure) relates features of two different feature models. In order to express this dependency, both source feature models have to be merged.

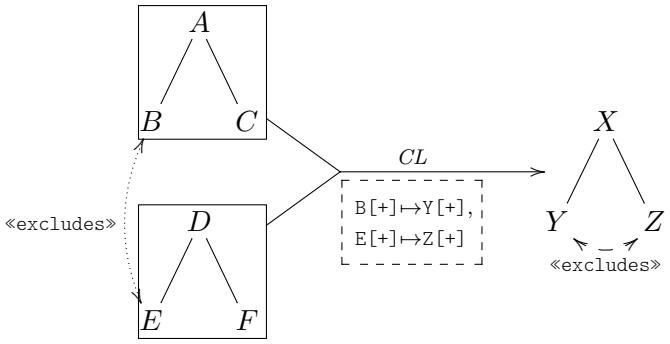


Figure 5.10: Propagation with multiple source feature models

5.2.3 Compatibility with Incremental Configuration

Incremental configuration is the activity of deriving a target configuration not in one single step but in several consecutive steps through the application of configuration links to source configurations [Rei09]. The idea is that, during the configuration of the source feature model, the derived effects in the target configuration can be made visible immediately. A refinement of the source configuration leads automatically to the corresponding refinement of the target configuration through the application of the configuration link. During incremental product configuration, the manual refinement of the target configuration is allowed as well. We have considered configuration links as functions that can be applied to source configurations so far and that return new target configurations. In the case of incremental configuration, a configuration link gets, besides a source configuration, an existing target configuration, which shall be refined. The application of the configuration link is, in principle, analogous to the case with a new target configuration. However, after the cardinality of a configured feature identifier was calculated, it is checked whether this configured feature identifier is already configured in the target configuration or not. If it is already configured, the calculated cardinality is combined with the existing cardinality (an exclusion has priority over an inclusion). Otherwise, the cardinality is set to the calculated cardinality. This means that an already included feature in the target configuration can only be excluded manually or by an explicit exclusion statement in the configuration link. This might seem counterintuitive on the first sight, but the intention of this semantics is to avoid unexpected overriding of inclusions coming from other configuration links or manual refinement of the target configuration.

The combination of incremental configuration and feature constraint propagation leads to some problems. Consider the example in Figure 5.11(a). The source-side feature link «`B excludes C`» is derived through the application of feature constraint propagation. Now consider the incremental configuration steps in Table 5.3. We start with an empty source and an empty target configuration. In step 1, the source-side feature `B` is selected. This leads to the effect that `Y` gets selected in the target configuration. The exclusion of `B` in step 2 does not affect the target configuration at all. In step 3, the selection of `C` leads to an invalid target configuration because the feature link «`Y excludes Z`» is violated. Note that all source configurations in Table 5.3 fulfill the propagated source-side feature link. Nevertheless, the incremental

configuration leads to an invalid target configuration. The crucial part is step 2 because the state of an already configured feature is changed (the included feature `B` is removed). It seems that the limitation of the source feature model's refinement to unconfigured features (i.e. the state of configured features may not be changed) makes feature constraint propagation compatible with incremental configuration. The example in Figure 5.11(b) shows that this is not the case. The propagation of the target-side feature link leads to the source-side feature link «`C needs B`». Table 5.4 shows the incremental configuration steps which lead to an invalid target configuration, although, all source configurations fulfill the propagated source-side feature link and only unconfigured features are configured in all steps. Starting with an empty source configuration, we derive a target configuration in which `Y` is selected. In step 1, we select `B` in the source configuration, which does not affect the target configuration at all. In step 2, we additionally select `C` in the source configuration and derive an invalid target configuration since both `Y` and `Z` are selected.

These examples show that feature constraint propagation is not compatible with incremental configuration. However, the technique can be extended such that it becomes compatible with incremental configuration. The extended version of feature constraint propagation gets a target configuration to be refined as additional input and propagates target-side constraints with respect to this configuration. For this purpose, the reverse inclusion mapping (cf. Section 5.1.2) has to be adapted for all configured feature identifiers that are already configured in the configuration.

- Every configured feature identifier *cfid* that is included in the configuration has to be mapped to $\neg(\psi_1 \vee \psi_2 \vee ...)$ with $\psi_1, \psi_2, ...$ being all criteria of configuration decisions (of the expanded configuration link) that exclude *cfid*. If such criteria for *cfid* do not exist, the configured feature identifier has to be mapped to `true`.
- Every configured feature identifier *cfid* that is excluded in the configuration has to be mapped to `false`.

In the examples of Figure 5.11(a) and Figure 5.11(b), the extended reverse inclusion mapping with respect to the target configuration $\{X[1], Y[1]\}$ is given by the following.

$\texttt{X} \mapsto \texttt{true}$	included in the target configuration
$\texttt{Y} \mapsto \texttt{true}$	included in the target configuration
$\texttt{Z} \mapsto \texttt{A[1]} \wedge \texttt{C[1]}$	not included in the target configuration

Consequently, the propagation of the target-side feature link with respect to the target configuration $\{X[1], Y[1]\}$ delivers, in both examples, the constraint $\neg\texttt{C[+]}$. The semantics of the extended version of feature constraint propagation differs from the one of classical feature constraint propagation as introduced in Section 5.1: every source configuration fulfilling the propagated constraints leads, through the application of the configuration link, to a valid refinement of the given target configuration (i.e. it fulfills the original constraints). Vice versa, every source configuration violating the propagated constraints results in an invalid refinement of the target configuration (i.e. it violates the original constraints). In our examples, the source

step	source configuration	derived target configuration
	$\emptyset$	$\emptyset$
1	$\{A[1], B[1]\}$	$\{X[1], Y[1]\}$
2	$\{A[1]\}$	$\{X[1], Y[1]\}$
3	$\{A[1], C[1]\}$	$\{X[1], Y[1], Z[1]\}$

Table 5.3: Incremental configuration steps w.r.t. the model in Figure 5.11(a)

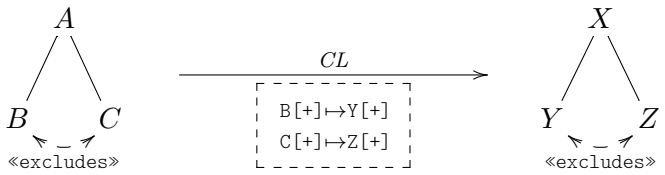
step	source configuration	derived target configuration
	$\emptyset$	$\{X[1], Y[1]\}$
1	$\{A[1], B[1]\}$	$\{X[1], Y[1]\}$
2	$\{A[1], B[1], C[1]\}$	$\{X[1], Y[1], Z[1]\}$

Table 5.4: Incremental configuration steps w.r.t. the model in Figure 5.11(b)

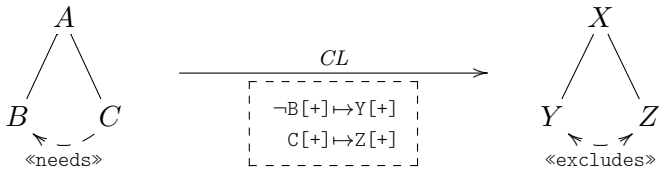
configurations of the steps leading to the invalid target configurations (step 3 in Table 5.3 and step 2 in Table 5.4) do not fulfill the propagated constraint $\neg$`C[+]`. The resulting refinements are invalid.

Figure 5.11(c) depicts an example for incremental configuration with multiple configuration links. A target configuration can be derived through the application of configuration link CL_1 to a source configuration. Then, incremental configuration can be used in order to refine this target configuration through the application of CL_2 to a source configuration of this configuration link. The application of feature constraint propagation to both configuration links obviously delivers `true`. However, if incremental configuration is used with the valid source configurations $\{A[1], B[1]\}$ and $\{D[1], E[1]\}$, `Y` as well as `Z` become selected, which violates the target-side feature link. Analogous to the examples above, the mentioned extension of feature constraint propagation can deal with this situation. After the configuration link CL_1 was applied, the extended version of feature constraint propagation can be applied with respect to CL_2 and the target configuration $\{X[1], Y[1]\}$ and delivers the source-side constraint $\neg$`E[+]`. This constraint makes the source configuration $\{D[1], E[1]\}$ invalid, as supposed.

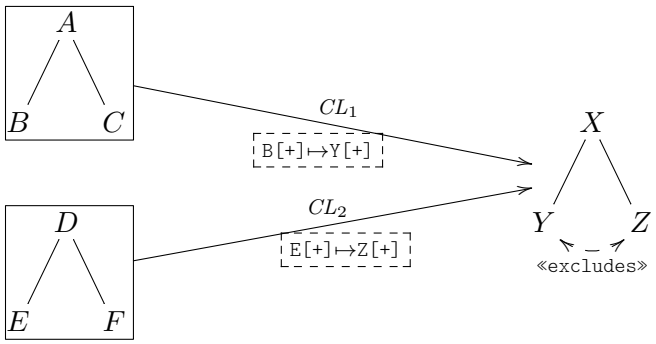
Summarizing, we showed in this section that feature constraint propagation is not compatible with incremental configuration. We can, however, adapt the technique such that this compatibility holds by extending the reverse inclusion mapping. Then, feature constraint propagation additionally gets a target configuration as input and delivers source-side constraints that a source configuration has to fulfill in order to represent a valid refinement for the given target configuration. This means that feature constraint has to be applied after every step of incremental configuration. Further research is required to clearly define and verify this extension of feature constraint propagation and to prove its compatibility with incremental configuration.



(a) Single configuration link 1



(b) Single configuration link 2



(c) Multiple configuration links

Figure 5.11: Examples for incremental configuration

5.2.4 Propagating Multiple Constraints Together

If the target feature model of a configuration link has more than one constraint, there are two possibilities of propagating them: (1) they can be propagated separately from each other or (2) they can be propagated together (i.e. combined and then propagated). Interestingly, both approaches can lead to different (but equivalent) source constraints. This comes from the fact that the applied postprocessing (cf. steps 4 to 6 in Section 5.1) only processes propagated constraints and does not consider existing constraints of the source feature model. We illustrate this by means of the example in Figure 5.12. The separate propagation of the three target-side feature links obviously leads to three source-side feature links.

$$\begin{aligned} \text{«X excludes Y»} &\implies \text{«B excludes C»} \\ \text{«Z needs Y»} &\implies \text{«D needs C»} \\ \text{«Z needs X»} &\implies \text{«D needs B»} \end{aligned}$$

If the three target-side constraints are combined and propagated together, the following steps are applied.

«X excludes Y» and «Z needs Y» and «Z needs X»	
$\implies \neg(\texttt{A[1]} \wedge \texttt{B[1]} \wedge \texttt{A[1]} \wedge \texttt{C[1]})$	transformation
$\wedge(\texttt{A[1]} \wedge \texttt{D[1]} \rightarrow \texttt{A[1]} \wedge \texttt{C[1]})$	
$\wedge(\texttt{A[1]} \wedge \texttt{D[1]} \rightarrow \texttt{A[1]} \wedge \texttt{B[1]})$	
$\equiv (\neg\texttt{A[1]} \vee \neg\texttt{D[1]}) \wedge (\neg\texttt{A[1]} \vee \neg\texttt{B[1]} \vee \neg\texttt{C[1]})$	minimization
$\equiv \neg\texttt{D[+]} \wedge (\neg\texttt{B[+]} \vee \neg\texttt{C[+]})$	inclusion-to-selection
$\equiv \neg\texttt{D[+]}$ and «B excludes C»	extract feature links

The three separately propagated feature links from above are equivalent to these propagated constraints. The postprocessing of the propagated constraint detects the anomaly that feature D is dead. Besides dead features, the postprocessing of feature constraint propagation can also detect other anomalies, e.g. void feature models, if they completely result from the propagated constraint. If target-side constraints are propagated separately, anomalies arising from their combination cannot be identified through feature constraint propagation. In order to identify them after the propagation, any anomalies detection techniques can be used (cf. Section 5.2.1). Anomalies arising from existing source-constraints or feature groups can never be identified through feature constraint propagation, as already discussed in Section 5.2.1.

5.2.5 Compatibility with Advanced Feature Constraints

One important point of configuration links [Rei08] is the support of partial configurations. Roughly spoken, a configuration is called partial if it is not completely defined, e.g. if a feature is unconfigured (neither included nor excluded). The application of configuration links often produces partial configurations. With feature constraints as introduced in this thesis (in this section called *basic feature constraints*

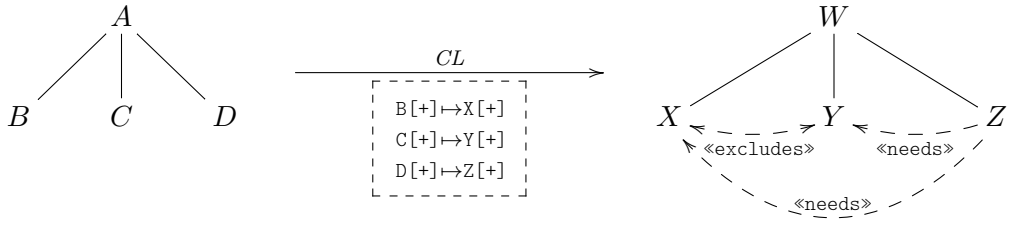


Figure 5.12: Example for the propagation of multiple constraints

in the following) it is not possible to express that a feature is unconfigured. We can only state that a feature is present (i.e. included or selected) or not. *Advanced feature constraints* provide the possibility to address the unconfigured state in constraints. However, is there a use case for advanced feature constraints in practice? In most cases, constraints are formulated with respect to selection or inclusion of features (resp. configured feature identifiers). At first sight, constraints considering unconfigured features seem to be dispensable for practical use. On closer inspection, however, it is conspicuous that the omission of the unconfigured state in constraints has the consequence that deselection and exclusion cannot be addressed either when dealing with partial configurations. In the case of full configurations, the constraint that a feature is excluded is equivalent to the constraint that it is not included. In partial configurations, however, this equivalence does not hold: the constraint that a feature is not included is fulfilled if the feature is excluded or unconfigured. In practice, constraints regarding the exclusion (or deselection) of a feature are popular and the use of partial configurations together with configuration links is useful. The same problems arise for criteria of configuration decisions. Without advanced feature constraints, it is, for example, not possible to formulate a configuration decision that is applied only if a distinguished configured feature identifier is excluded (in a partial configuration). Summarizing, advanced feature constraints provide more expressiveness when dealing with partial configurations. They allow to address selection `[+]`, inclusion `[1]`, deselection `[-]`, exclusion `[0]` and the unconfigured state `[0..1]`.

The question arises which effects the use of advanced feature constraints has on feature constraint propagation. Consider the example in Figure 5.13(a) and the six steps of feature constraint propagation presented in Section 5.1. The expansion of the configuration link has to be extended to the expansion of the deselection statement $\texttt{B[-]} \Longrightarrow \texttt{A[0]} \vee \texttt{B[0]}$. The reverse mappings and the transformation can be used without any adaption. However, all postprocessing steps (cf. steps 4 to 6 in Section 5.1) have to be adapted in order to be compatible with the propagated exclusion statements. The propagation of the target-side feature link obviously results in the source-side feature constraint $\neg(\texttt{B[-]} \wedge \texttt{C[+]})$. If only full configurations are used, this constraint could be expressed by the source-side feature link «`C needs B`» (cf. Figure 5.11(b)). In the general case, the constraint is not equivalent to this feature link: the configuration $\{\texttt{A[1]}, \texttt{B[0..1]}, \texttt{C[1]}\}$ fulfills the constraint $\neg(\texttt{B[-]} \wedge \texttt{C[+]})$

but not the feature link «C needs B». The example in Figure 5.13(b) contains a target-side feature model with an advanced feature constraint. In order to propagate this constraint, the reverse mappings have to be extended by a reverse deselection mapping of configured feature identifiers in order to map Y[-]. This mapping answers the question under which conditions a target-side configured feature identifier gets deselected. To define this mapping and to map exclusion statements, a reverse exclusion mapping of configured feature identifiers is additionally required. In the example, the application of the reverse deselection mapping to Y delivers A[1]∧B[1]. The transformation of feature constraint propagation has to be adapted such that it uses the new mappings. The propagation of the target-side constraint Y[-] ∨ ¬Z[+] results in the source-side feature link «C needs B».

The two examples show that all six steps of feature constraint propagation have to be extended to be compatible with advanced feature constraints.

1. The expansion algorithm in step 1 also has to replace deselection statements in criteria by exclusion statements.

2. Additional reverse mappings (the reverse exclusion mapping, the reverse deselection mapping and the reverse unconfigured mapping of configured feature identifiers) are required in step 2.

3. The transformation in step 3 has to be adapted to the extended reverse mappings. It also has to substitute exclusion statements, deselection statements and statements addressing the unconfigured state.

4. The minimization in step 4 has to be compatible with multi-valued formulae (e.g. the ESPRESSO-MV algorithm [RSV87]).

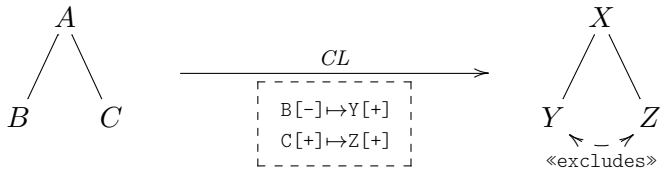
5. The inclusion-to-selection algorithm in step 5 has to be enriched by an additional exclusion-to-deselection algorithm.

6. The extraction of feature links in step 6 also has to be extended to be compatible with advanced feature constraints.

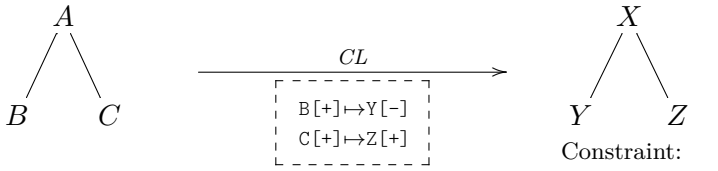
In Section 6.4 and Section 6.5, we formally define advanced feature constraints and the required adaption of the transformation step (including the required reverse mappings) and show its correctness. Further research is required in order to adapt the other steps of feature constraint propagation to advanced feature constraints. This is not part of this thesis.

5.2.6 Forward Feature Constraint Propagation

Instead of propagating feature constraints from low-level models to higher-level, they can also be propagated in the opposite direction. We call this *forward feature constraint propagation* or short *forward propagation*. The motivation for this technique is to make implicit constraints resulting from constraints of higher-level models accessible on lower-levels. This technique is not as relevant for the practical use of configuration links as backward propagation because it does not reveal errors in the variability specification or avoid the derivation of invalid configurations. It only



(a) Advanced feature constraints in criteria



(b) Advanced feature constraints in target-side feature model

Figure 5.13: Examples for advanced feature constraints

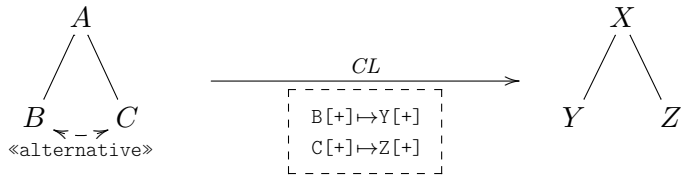


Figure 5.14: Example for forward feature constraint propagation

makes the implicit exclusion of configurations (by the configuration link) explicit (by adding constraints). Forward propagation is only briefly introduced in this thesis. Further research is required to formally define, verify, evaluate and implement this technique. In order to avoid ambiguities, feature constraint propagation from lower-level to higher-level models is called *backward propagation* in this section.

Consider the example in Figure 5.14. The first idea for forward propagation was to define it as substitution of variables in logical formulae, similar to backward propagation. This would mean that the propagation of the source-side feature link leads to a target-side `alternative` feature link between `Y` and `Z`. Features `B` or `C` have to be selected in the source feature model, the selection of `B` implies the selection of `Y` and the selection of `C` implies the selection of `Z`. On the first sight this seems reasonable, however, it is not. Consider the source configuration with only feature `B` being selected. This leads, through the application of the configuration link, to a partial target configuration with only `Y` being selected. The feature `Z` is unconfigured. The propagated `alternative` feature link between `Y` and `Z` would forbid to manually select the unconfigured feature `Z`. This shows that a propagated `alternative` feature link is too strong in this example. The idea of substituting variables in logical formulae, which properly works for backward propagation, does not work for forward propagation.

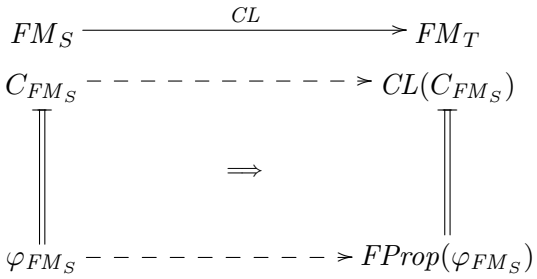


Figure 5.15: Correctness of forward feature constraint propagation

What is the exact intention of forward propagation? It has to narrow the set of target configurations such that only configurations which (1) cannot be derived through the application of the configuration link or (2) cannot manually be created by configuring unconfigured features of a derived configuration become invalid. This means for the example that the feature constraint `Y[+]` $\vee$ `Z[+]` can be propagated. Every valid source configuration leads to a derived configuration fulfilling this constraint (№ 1 above), i.e. either `Y` or `Z` is selected in every derived configuration. Every unconfigured feature (`Y` or `Z`) can then manually be selected or deselected without violating the propagated constraint (№ 2 above). Basing on these considerations, we introduce the concept of forward propagation in the following.

Forward Feature Constraint Propagation: Concept

Analogous to backward propagation, forward feature constraint propagation consists of six consecutive steps. However, it is not based on the substitution of variables in constraints but on the calculation of configuration steps that are always applied during the application of the configuration link to any valid source configuration. In order to consider implicit effects of configuration decisions, the constraints to be propagated as well as the configuration link are expanded in the first step. Then, the applied configuration steps are calculated and logical formulae are extracted. Finally, the same three postprocessing steps as in the case of backward propagation are applied: the formulae are minimized, inclusion statements are lifted to selection statements and feature links are extracted. Up to now, forward propagation works only for basic feature models and not for advanced concepts. It fulfills the correctness property but not the minimality property, as illustrated in Figure 5.15. This means that every configuration fulfilling a source-side constraint leads to a configuration fulfilling the propagated target-side constraint (correctness). However, it does not hold that every source-side configuration violating a source-side constraint leads to a configuration which does not fulfill the propagated target-side constraint (minimality).

$$C_{FM_S} \models \varphi_{FM_S} \Longrightarrow CL(C_{FM_S}) \models FProp(\varphi_{FM_S}) \quad \text{correctness}$$
$$C_{FM_S} \models \varphi_{FM_S} \not\Longleftarrow CL(C_{FM_S}) \models FProp(\varphi_{FM_S}) \quad \text{minimality does not hold}$$

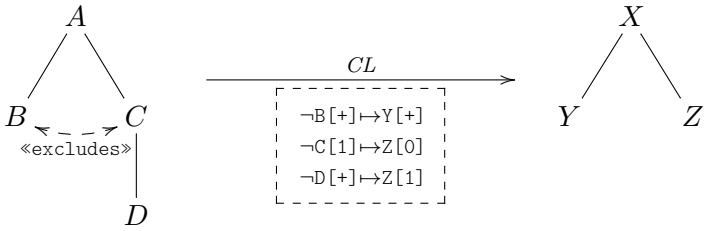


Figure 5.16: Example for illustrating the technique of forward propagation

$$\begin{aligned} \neg(\texttt{A[1]} \wedge \texttt{B[1]}) &\mapsto \texttt{X[1]}, \texttt{Y[1]} \\ \neg\texttt{C[1]} &\mapsto \texttt{Z[0]} \\ \neg(\texttt{A[1]} \wedge \texttt{C[1]} \wedge \texttt{D[1]}) &\mapsto \texttt{Z[1]} \end{aligned}$$

Figure 5.17: The expanded configuration link of Figure 5.16

The achievement of minimality is not possible for forward propagation. Consider the trivial source-side constraint `F[1]`, which states that a specific configured feature identifier `F` has to be included in every configuration, and a configuration link that does not contain `F` in any criteria. We can only propagate `true` because the source-side constraint does not affect the target feature model at all. Now consider a source configuration that violates the constraint (i.e. `F` is not included). The application of the configuration link to this invalid configuration leads to a valid target configuration (every configuration fulfills the propagated constraint `true`), which shows that the propagated constraint is not minimal.

In the following, we illustrate the six steps of forward propagation by the example in Figure 5.16.

Step 1: Expand configuration link and constraints

In the first step, the configuration link as well as the constraints to be propagated are expanded (cf. Section 5.1.1). If a feature link shall be propagated, it has to be expressed as equivalent feature constraint at first. Note that this is not always possible in the case of advanced feature models.

The expanded configuration link of the example is depicted in Figure 5.17 and the preprocessing of the feature link is shown in the following.

$$\begin{aligned} &\texttt{«B excludes C»} \\ &\quad\equiv \neg(\texttt{B[+]} \wedge \texttt{C[+]}) && \text{convert to feature constraint} \\ &\quad\equiv \neg((\texttt{A[1]} \wedge \texttt{B[1]}) \wedge (\texttt{A[1]} \wedge \texttt{C[1]})) && \text{expand feature constraint} \end{aligned}$$

Step 2: Calculate applied configuration steps

The applied configuration steps for a constraint to be propagated are all configuration steps that are contained in configuration decisions whose criteria are fulfilled by the constraint. This means that a configuration step is in this set if the criterion of its configuration decision is implied by the constraint. We propose to convert the constraint into a disjunctive normal form and to calculate a set of applied configuration steps for every term of the resulting constraint in order to allow a more precise propagation, i.e. to make the propagated constraint as strong as possible.

In the example, the constraint to be propagated in disjunctive normal form is $\neg\texttt{A[1]} \vee \neg\texttt{B[1]} \vee \neg\texttt{C[1]}$. Now we check which criteria of the configuration link are implied by the individual terms of the constraint.

$\neg\texttt{A[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{B[1]})$	holds, X[1], Y[1] are applied
$\neg\texttt{A[1]} \rightarrow \neg\texttt{C[1]}$	does not hold
$\neg\texttt{A[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{C[1]} \wedge \texttt{D[1]})$	holds, Z[1] is applied
$\neg\texttt{B[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{B[1]})$	holds, X[1], Y[1] are applied
$\neg\texttt{B[1]} \rightarrow \neg\texttt{C[1]}$	does not hold
$\neg\texttt{B[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{C[1]} \wedge \texttt{D[1]})$	does not hold
$\neg\texttt{C[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{B[1]})$	does not hold
$\neg\texttt{C[1]} \rightarrow \neg\texttt{C[1]}$	holds, Z[0] is applied
$\neg\texttt{C[1]} \rightarrow \neg(\texttt{A[1]} \wedge \texttt{C[1]} \wedge \texttt{D[1]})$	holds, Z[1] is applied

This leads to the following sets of applied configuration steps.

- term $\neg$A[1]: {X[1], Y[1], Z[1]}
- term $\neg$B[1]: {X[1], Y[1]}
- term $\neg$C[1]: {Z[0], Z[1]}

Step 3: Extract logical formulae

Now logical formulae are extracted out of the sets of applied configuration steps by combining the contained configured feature identifiers with conjunctions. If more than one configuration step for a configured feature identifier occurs in a set (e.g. one stating to include and one stating to exclude a configured feature identifier), these configuration steps are combined as during the application of configuration links. Contradictions between configuration steps are resolved in the defined way. Exclusion statements are prioritized over inclusion statements. Finally, the resulting formulae are connected by disjunctions. If basic feature constraints are used (cf. Section 5.2.5), deselection statements are converted into negated inclusion statements (i.e. to weaker constraints).

In the example, the applied configuration steps Z[0] and Z[1] of the term $\neg$C[1] are combined to Z[0], which is converted into the weaker constraint $\neg$Z[1].

- term ¬A[1]: X[1] ∧ Y[1] ∧ Z[1]
- term ¬B[1]: X[1] ∧ Y[1]
- term ¬C[1]: ¬Z[1]

This leads to the following overall constraint.

$$(\texttt{X[1]} \wedge \texttt{Y[1]} \wedge \texttt{Z[1]}) \vee (\texttt{X[1]} \wedge \texttt{Y[1]}) \vee \neg\texttt{Z[1]}$$

Steps 4-7: Restructure logical formulae

Finally, the resulting formulae are restructured analogous to steps 4-6 of backward propagation. These steps have already been described in detail in Section 5.1.4, Section 5.1.5 and Section 5.1.6.

The restructuring of the formula leads to the following final constraint in the example.

$$\texttt{Y[+]} \vee \neg\texttt{Z[1]}$$

This means that only configurations fulfilling this constraint can be obtained through the application of the configuration link to a valid source configuration and refining the result. Valid source configurations are those with B or C being not selected. If B is not selected, Y gets selected and Z can be configured manually without running the risk of violating the propagated constraint. Otherwise, if C is not selected, either Y gets selected (analogous to the previous case) or Z gets excluded. In the last-mentioned case, X and Y can be configured manually without risking the violation of the propagated constraint. This shows the correctness of the propagation in the example. Now consider the invalid source configuration with all features being included. The application of the configuration link would lead to a valid target configuration with all features being unconfigured. This shows that the minimality of the propagation does not hold in the example.

5.3 Comparison with Alternative Approaches

In this section we compare two alternative approaches with feature constraint propagation and discuss their shortcomings.

5.3.1 Reasoning about Feature Models and Configuration Links

We have already presented a selection of analysis operations basing on reasoning (i.e. some kind of constraint solving) techniques in Section 5.2.1. If we think of using such reasoning based techniques as alternative to feature constraint propagation (as already briefly discussed in Section 1.1), the following well-known operations are particularly interesting [Ben07].

- *valid product*: checks a given product for validity
- *void feature model*: checks if a feature model is void (cf. Section 5.2.1)
- *all products*: calculates all products

These operations can also be applied if a product line is hierarchically organized with configuration links. For this purpose, the whole variability specification (i.e. all feature models, configuration links and constraints) has to be converted into a suitable representation before the analysis operation can be applied. In general, it is formulated as some kind of constraint satisfaction problem and a constraint solver is used to accomplish the analysis. In this setting, it is not sufficient to formulate all configuration decisions as logical formulae and to combine them by conjunction but the exact semantics of configuration links has to be included (especially the resolution of contradictory configuration decisions has to be respected).

This allows, for example, (a) to check whether a high-level configuration is valid with respect to the overall variability specification, i.e. the configuration leads to valid configurations of all lower-level models (operation *valid product*), (b) to check the satisfiability of the overall product line (operation *void feature model*) and (c) to identify the set of valid products of the overall product line (operation *all products*). The last-mentioned operation also allows to identify all valid products of the individual feature models by filtering the result according to the features of the corresponding model. This is exactly what feature constraint propagation does. So why should we use feature constraint propagation? As elegant and effective reasoning based approaches may be, they are faced with the following shortcomings.

1. **Access to the entire variability specification is required.** If a highly complex product line is organized hierarchically, it comprises numerous subordinate product lines, each represented by its own feature model. In order to use a constraint solver to apply the mentioned analysis operations, we need access to the entire variability specification, i.e. all feature models, all configuration links and all constraints. The access to only one configuration link and its incident feature models is not sufficient for reasoning based approaches because the individual feature models are not self-contained (cf. Nº 2). This means that not all locally valid target configurations of a configuration link have to be valid with respect to the overall variability specification as well. A locally valid target configuration could lead to an invalid configuration of a subordinate product line. Since the resulting constraint satisfaction problem contains the entire variability specification of the overall product line, it becomes very complex. Although modern constraint solvers contain highly efficient algorithms, the enormous complexity of large-scale industrial product lines can constitute a problem [BSTRC06]. Besides the complexity, the required access to the entire variability specification can pose a critical problem whenever a subordinate product line is part of a separate legal entity (e.g. an external supplier) because of business confidentiality. In this case, reasoning based approaches are not usable at all. Feature constraint propagation, in contrast, does not need access to the entire variability specification. The propagation can be accomplished stepwise along every configuration link of the variability specification (bottom-up). It only needs access to one configuration link and its incident feature models. In principle, steps 1 to 3 (cf. Section 5.1) need only access to the configuration link and the target feature model, step 4 does not need access to any model and steps 5 to 6 need only access

to the source feature model. This means that the propagation is, in principle, even applicable if every stakeholder has only access to one feature model. Since the propagation can be applied locally, it has to solve only local – and therefore smaller – problems and does not have to consider the overall product line. However, it has to be applied for every configuration link of the entire variability specification bottom-up in order to make the implicit constraints explicit in all feature models.

2. **The actual meaning of a target-side constraint within the taxonomy of the source feature model is not revealed.** The result of the analysis operations are, for the operations *valid product* and *void feature model*, only `true` or `false` (and possibly a reason for the answer) or, for the operation *all products*, a list of all valid products. Feature constraint propagation, in contrast, returns source-side constraints expressing the impacts of the target-side constraints within the taxonomy of the source feature model. While being entirely redundant to the overall variability specification, the propagated constraints make all implicit effects of constraints of lower-level models visible on higher-levels. This can be extremely helpful, for example, when debugging hierarchically organized product lines because unacceptable constraints indicate errors in the model. As an example consider Figure 2.5: the propagated constraint that the comfort package is unavailable in the U.S. because of a technical dependency is surely unacceptable from a marketing point of view, as already mentioned. Without making the implicit effects of lower-level feature models explicit in higher-level models, such information remains hidden and is practically unascertainable for some stakeholders. Although the storage of redundant information in data models contradicts a fundamental principle of computer science, adding significant propagated constraints to higher-level models can be very useful because the models become self-contained. This allows the engineers to locally configure a feature model without paying regard to related feature models because all locally valid configurations are also valid with respect to the overall product line (i.e. they lead to valid configurations of all lower-level feature models)[6]. In addition, individual feature models can be used as interfaces for the variability of subordinate product lines, which allows local application of reasoning based approaches.

3. **The analysis operation *all products* is unfeasible for models with a large number of products.** The operation *all products* is the only analysis operation that is able to identify all source configurations that lead to valid target configurations. These source configurations have to be analyzed and it has to be checked whether all desired products are in this set. This operation could theoretically be used instead of feature constraint propagation. However, the operation has to be applied to the overall product line and cannot be applied stepwise, as stated in Nº 1. In large-scale product lines, there can be an enormous amount of resulting products. In this case, their enumeration is

[6]This property only holds if configuration links do not produce configurations that violate a cardinality restriction.

unfeasible, as [BRCTS06] states. In contrary, feature constraint propagation can be used with such models since its result is not a set of products but a set of constraints, which narrows the set of products. For example, if all source configurations lead to valid target configurations, the analysis operation would enumerate all products, whereas feature constraint propagation would simply propagate `true`.

Naturally, feature constraint propagation has not only advantages over reasoning based approaches. One disadvantage, for example, is the runtime. Modern constraint solvers can deal with feature models with thousands of features [Men09]. However, feature constraint propagation can be applied locally and does not need to take the overall variability specification into account, as constraint solving approaches have to do.

Although reasoning based analysis techniques cannot be used to propagate constraints between feature models as feature constraint propagation does (since they tend to solve constraints and not to transform or restructure them), we argued that they can, in principle, be used as alternative to feature constraint propagation with the above-mentioned shortcomings. The application of feature constraint propagation before reasoning based approaches are used, however, allows to combine the advantages of both approaches (cf. Section 5.2.1). Reasoning based approaches can locally be applied to one feature model after the propagation. Then, all local results of the analysis operations also hold for the overall variability specification. One more advantage of the combination of feature constraint propagation and analysis techniques is that existing analysis algorithms and tools can be used and do not have to be adapted because only single feature models have to be analyzed and configuration links can be neglected.

5.3.2 The Brute-Force Approach

The idea of the brute-force approach is trivial. In general, a brute-force algorithm systematically enumerates all possible candidates for the solution and checks every candidate whether it fulfills the problem statement or not. In our case, this means that we have to enumerate all valid configurations of a feature model, apply the configuration link to each of them and check the derived target configurations for validity. If a target configuration is not valid, we mark the original source configuration as invalid as well. This activity has to be performed for every configuration link bottom up in order to identify all valid configurations with respect to the overall product line on all levels.

Besides the shortcomings mentioned in Section 5.3.1, which also apply to the brute-fore approach, this approach is not feasible for large-scale product lines because of the enormous number of configurations to be checked. The upper bound for the number of (full) configurations of a feature model with only optional features is $2^{|F|}$, with F being the set of features, since every feature can be included or excluded. This means that already a feature model with 100 features can theoretically have over a nonillion ($2^{100} \approx 1.267 * 10^{30}$) configurations. Admittedly, the number of real products is, in practice, significantly smaller because of the tree structure of the

feature model (in classical feature modeling approaches, the selection of a feature implies the selection of its parent). However, if we use configuration links, we have already motivated the separation of inclusion and selection semantics. Configuration links can contain configuration decisions which are applied if a distinguished feature is included (regardless of the state of its parent). Therefore, the brute-force approach also has to consider inclusion semantics and not only selection statements in order to identify all valid products with respect to the overall product line, which needs much more effort. Advanced concepts (e.g. cloned features or feature inheritance) also increase the number of configurations. Furthermore, if partial configurations are allowed, the number of configuration grows, for a feature model with only optional features, from $2^{|F|}$ to $3^{|F|}$.

Chapter 6

Feature Constraint Propagation: Formalization and Verification

Even though the technique of feature constraint propagation has already been introduced in detail in this thesis, we additionally provide a sound formalization of the technique. This helps to get a clearer comprehension of feature constraint propagation and removes ambiguities. In addition, it makes the semantics of feature constraint propagation transparent. This means that the exact semantics of the technique is not "hidden" in a tool but explicitly described in an general notation, which poses as basis for the implementation (cf. Section 7.1). The formalization itself represents a fundamental research result, which extends the concept of configuration links. Furthermore, it serves as basis for the verification of feature constraint propagation. We formally define the properties correctness and minimality and show that feature constraint propagation fulfills them.

In order to introduce the formal definition of feature constraint propagation, we need an accurate formalization of the underlying basic concepts. In literature, a range of different formalizations of feature models and configurations can be found. These formalizations differ in their underlying theories (set theory, logic, formal languages, etc.), their covered feature modeling concepts (cardinalities, parameterized features, inheritance, etc.) and in their formal syntax and semantics. Since feature constraint propagation uses the concept of configuration links, which was introduced and formalized in [Rei08], it is nearby that the formal definition of feature constraint propagation is based on this formalization. We therefore chose a formalization in pure set theory and first-order predicate logic.

This chapter starts with the formal definition of feature models in Section 6.1 and the formal definition of configuration links in Section 6.2. Feature constraint propagation is formalized and its correctness and minimality are proven in Section 6.3. Section 6.4 contains the formal definition of advanced feature constraints. Finally, the formalization of feature constraint propagation for advanced feature constraints and the proofs for its correctness and minimality are presented in Section 6.5.

6.1 Formal Definition of Feature Models

This section introduces the formalization of feature models, which is based on the formalization of advanced feature models in [Rei08]. Most definitions presented in this section are similar to those in [Rei08]. However, some properties in this thesis are formulated more detailed and others are defined in a different way. These extensions are necessary for the definition of feature constraint propagation and the accomplished proofs. Major changes are marked.

In order to cover all advanced concepts of feature models, we need a set of instance names $\mathcal{IN}$ and a set of all possible values of parameterized features $\mathcal{V}$. Both of these sets can be arbitrary. Their definitions depend on the domain of discourse and the used data-types. We only postulate the existence of a wildcard *NoInstance* that is not contained in the set of instance names (i.e. $NoInstance \notin \mathcal{IN}$). This wildcard describes that a feature has no instances. In the following, we denote the set of propositional logic formulae over the set of variables X by $Form(X)$.

The following definition of feature models contains feature links and feature constraints as part of the feature model itself. Their semantics and the definition of configured feature identifiers, which uniquely identify a feature on instance level, are presented later in this section.

Definition 6.1.1 (Feature Model)**.** A feature model FM is given by

$$\begin{aligned} FM = (&F_{FM}, Parent_{FM}, Inherit_{FM}, Group_{FM}, Card_{FM}, Type_{FM}, \\ &Constraints_{FM}, FLinks_{FM}) \end{aligned}$$

with

- the (finite) set of features F_{FM}
- relation $Parent_{FM} \subseteq F_{FM} \times F_{FM}$, where $(p, c) \in Parent_{FM}$ means that feature p is the parent of c
- relation $Inherit_{FM} \subseteq F_{FM} \times F_{FM}$, where $(f_1, f_2) \in Inherit_{FM}$ means that feature f_1 inherits (the children) of feature f_2
- equivalence relation $Group_{FM} \subseteq F_{FM} \times F_{FM}$, where $(f_1, f_2) \in Group$ means that features f_1 and f_2 are in a feature group
- function $Card_{FM} : F_{FM} \cup (F_{FM_{/Group_{FM}}} \backslash \{M| \ |M| = 1\}) \rightarrow \mathcal{P}(\mathbb{N}) \backslash \emptyset$ mapping each feature and each feature group to a valid cardinality
- partial function $Type_{FM} : F_{FM} \rightsquigarrow \mathcal{P}(\mathcal{V}) \backslash \emptyset$ mapping each parameterized feature to its type
- set of feature links $FLinks_{FM} = \{excludes_{FM}, alternative_{FM}, needs_{FM}\}$ with
  - (finite) symmetric relation $excludes_{FM} \subseteq F_{FM} \times F_{FM}$ describing `excludes` feature links

  - (finite) symmetric relation $alternative_{FM} \subseteq F_{FM} \times F_{FM}$ describing `alternative` feature links
  - (finite) relation $needs_{FM} \subseteq F_{FM} \times F_{FM}$ describing `needs` feature links

- (finite) set of feature constraints[1] $Constraints_{FM} \subseteq Form(\mathcal{CFIDI}_{FM})$

where the following conditions hold.

$$G = (F_{FM}, Parent_{FM}) \text{ is a forest of arborescences} \tag{6.1}$$

$$G = (F_{FM}, Inherit_{FM} \cup Parent_{FM}) \text{ is an acyclic digraph} \tag{6.2}$$

$$(f_1, f_2) \in Group_{FM} \implies \exists p \in F_{FM} : \begin{pmatrix} (p, f_1) \in Parent_{FM} \\ \wedge (p, f_2) \in Parent_{FM} \end{pmatrix} \tag{6.3}$$

Condition 6.1 describes how the features of a feature model can be organized: the graph formed by the features and the parent relationships has to be a forest of arborescences. An arborescence is a directed, rooted out-tree, i.e. a directed graph with a root node and exactly one path from the root node to any other node [Tut01]. A forest is a disjoint union of trees. Allowed inheritance relationships are described by condition 6.2: the graph defined by the features as nodes and the parent and inheritance relationships as edges has to be an acyclic digraph (DAG). This is a necessary condition to avoid infinite parent-child relationships, which could arise by inheritance. Note that condition 6.2 is stronger than the corresponding condition in [Rei08, condition 4.29] (it additionally comprises the parent relationships). Finally, condition 6.3 states that grouped features have the same parent. Obviously, root features cannot belong to groups.

For convenience, we define the partial function $parentOf_{FM} : F \rightsquigarrow F$ with

$$parentOf_{FM}(c) = p \iff (p, c) \in Parent_{FM}$$

and the following abbreviations for a feature f.

$$\begin{aligned}
f \in FM &\iff f \in F_{FM} \\
isRoot_{FM}(f) &\iff parentOf_{FM}(f) \text{ is not defined} \\
isLeaf_{FM}(f) &\iff \nexists c \in F_{FM} : parentOf_{FM}(c) = f \\
isParameterized_{FM}(f) &\iff Type_{FM}(f) \text{ is defined} \\
isOptional_{FM}(f) &\iff Card_{FM}(f) = \{0, 1\} \\
isMandatory_{FM}(f) &\iff Card_{FM}(f) = \{1\} \\
isCloned_{FM}(f) &\iff \begin{pmatrix} Card_{FM}(f) \neq \{0\} \\ \wedge \neg isOptional_{FM}(f) \\ \wedge \neg isMandatory_{FM}(f) \end{pmatrix}
\end{aligned}$$

[1] A feature constraint is a propositional logic formula over configured feature identifiers $\mathcal{CFIDI}_{FM}$ that do not point to a cloned feature itself (introduced later in Definition 6.1.4). Note that the formalization of feature constraints does not provide an abbreviation for selection, as introduced in Section 2.1.

Furthermore, for the sake of convenience, the indices can be neglected if the feature model is obvious.

In the following, we introduce configurations of feature models and their validity. Because of the concepts of cloned features and feature inheritance, there is no bijective relation between the features on feature level and the features on instance level (cf. Section 2.2, especially Table 2.4). Therefore, we have to introduce some additional concepts for the unique identification of a feature on instance level before we are able to define configurations and their validity.

When using cloned features, we need the possibility to assign names to instances of features and to identify certain instances. For this purpose, we define the concept of instance specifications.

Definition 6.1.2 (Instance Specification)**.** Given a feature model FM. An instance specification is a tuple $is = (f, i)$ with

$$f \in FM$$
$$i \in \mathcal{IN} \cup \{NoInstance\}$$

where the following condition holds.

$$\neg isCloned_{FM}(f) \implies i = NoInstance \tag{6.4}$$

All possible instance specifications of a given feature model FM form the set $\mathcal{IS}_{FM}$.

An instance specification can either point to an instance of a cloned feature (if $i \neq NoInstance$) or to the cloned feature itself (if $i = NoInstance$). If a feature is not cloned, an instance specification cannot contain an instance name (condition 6.4).

For an instance specification $is = (f, i)$, we define the following abbreviations.

$$f(is) = f$$
$$i(is) = i$$
$$isInst(is) \iff i \neq NoInstance$$

Note that $isInst(is) \implies isCloned_{FM}(f(is))$ holds by definition.

Definition 6.1.3 (Equality of Instance Specifications)**.** Two instance specifications $is = (f, i)$ and $is' = (f', i')$ are called *equal*, written $is = is'$, if and only if $f = f' \wedge i = i'$.

We have already defined parent relationships between features. However, if we take inheritance into account, a feature on feature level may have (multiple forms with) different parents on instance level, as already mentioned. Consider a feature P with a child C and a feature F which inherits of P. Then the feature C has two parents on instance level: a direct one (P) and an inherited one (F). This

relationship is expressed by the auxiliary relation $Parent_{FM}^{Inh} \subseteq F_{FM} \times F_{FM}$ defined by:

$$(p,c) \in Parent_{FM}^{Inh} \iff \left(\begin{array}{l} parentOf_{FM}(c) = p \\ \\ \vee \exists f \in F_{FM} : \left(\begin{array}{c} (p,f) \in Inherit_{FM} \\ \wedge (f,c) \in Parent_{FM}^{Inh} \end{array} \right) \end{array} \right)$$

Instance specifications (see Definition 6.1.2) provide the possibility to identify an individual instance of a cloned feature. However, this is not sufficient for identifying a concrete feature in a configuration. If a cloned feature has a cardinality greater than one in a configuration, not only the feature is duplicated itself but also its successors. Consider a cloned feature P with a child-feature C and a configuration with two instances p_1 and p_2 of P. Then, we need the possibility to address the child C of p_1 and the child C of p_2 separately. The same problem of uniquely identifying features on instance level arises when inheritance is used. The already known concept of configured feature identifiers solves this problem.

Definition 6.1.4 (Configured Feature Identifier)**.** Given a feature model FM. Then a configured feature identifier is defined as $cfid = (is_0, is_1, ..., is_n)$ with

$$isRoot_{FM}(f(is_0)) \tag{6.5}$$

$$\forall i \in \mathbb{N}, 0 \leq i \leq n : is_i \in \mathcal{IS}_{FM} \tag{6.6}$$

$$\forall i \in \mathbb{N}, 1 \leq i \leq n : (f(is_{i-1}), f(is_i)) \in Parent_{FM}^{Inh} \tag{6.7}$$

$$\forall i \in \mathbb{N}, 0 \leq i < n : isCloned_{FM}(f(is_i)) \implies i(is_i) \neq NoInstance \tag{6.8}$$

All configured feature identifiers of a feature model FM form the set $\mathcal{CFID}_{FM}$ and all configured feature identifiers with condition 6.8 holding also for $i = n$ (i.e. they do not point to a cloned feature itself) form the set $\mathcal{CFIDI}_{FM} \subseteq \mathcal{CFID}_{FM}$.

Roughly spoken, a configured feature identifier is a sequence of instance specifications (condition 6.6) that starts at a root feature (condition 6.5). This sequence forms a path through the parent relationship graph with respecting inheritance relationships (condition 6.7). Condition 6.8 states that every instance of a cloned feature has an instance name. In addition, condition 6.8 expresses two very important facts: a configured feature identifier can point to a cloned feature itself (if the condition does not hold for the last feature of the sequence) or to an instance of a cloned feature (if the condition also holds for the last feature of the sequence). To simplify matters, we did not mention configured feature identifiers pointing to cloned features themselves in the previous chapters (cf. Table 2.4). However, they are required for the formal definition of feature constraints and feature constraint propagation. The separation of configured feature identifiers that do not point to cloned features themselves (set $\mathcal{CFIDI}_{FM}$) is not included in the original formalization [Rei08]. Note that the set of configured feature identifiers is isomorphic to the set of features in the case of basic feature models.

For convenience, we define the following abbreviations and auxiliary functions.

$$f((is_0, is_1, ..., is_n)) = f(is_n)$$
$$i((is_0, is_1, ..., is_n)) = i(is_n)$$
$$isInst((is_0, is_1, ..., is_n)) = isInst(is_n)$$

$$Parent : \mathcal{CFID}_{FM} \rightsquigarrow \mathcal{CFID}_{FM} \text{ with}$$
$$Parent((is_0, is_1, ..., is_n)) = (is_0, is_1, ..., is_{n-1}) \quad \text{for all } n > 0$$

$$NoInst : \mathcal{CFID}_{FM} \rightarrow \mathcal{CFID}_{FM} \text{ with}$$
$$NoInst(((f_0, i_0), (f_1, i_1), ..., (f_n, i_n))) = ((f_0, i_0), (f_1, i_1), ..., (f_n, NoInstance))$$

Note that this parent function differs from the parent relations $Parent_{FM}$ and $Parent_{FM}^{Inh}$. Besides the fact that *Parent* is a function and $Parent_{FM}$ and $Parent_{FM}^{Inh}$ are relations, there is a serious difference between them. The function *Parent* is, in contrast to $Parent_{FM}$ and $Parent_{FM}^{Inh}$, defined on instance level. Consequently, *Parent* is applicable to configured feature identifiers and not to features. It identifies the unique parent of a configured feature identifier.

The next definition describes the equality of two configured feature identifiers. We need two different notions of equality.

Definition 6.1.5 (Equality of Configured Feature Identifiers)**.** Two configured feature identifiers $cfid = (is_0, is_1, ..., is_n)$ and $cfid' = (is'_0, is'_1, ..., is'_m)$ are called *equal*, written $cfid = cfid'$, if and only if $n = m \wedge is_i = is'_i$ for $i \in \{0, ..., n\}$.

They are called *equal up to the instance*, written $cfid =_f cfid'$, if and only if $NoInst(cfid) = NoInst(cfid')$.

Note that $(cfid =_f cfid') \Longrightarrow (f(cfid) = f(cfid'))$ (equivalence does not hold).

Now we are ready to define configurations. For this purpose, we introduce the set of configuration activities and, subsequently, the concept of configurations itself.

Definition 6.1.6 (Set of all Configuration Activities)**.** The set of all configuration activities $\mathcal{CA}$ is defined as $\mathcal{CA} = \mathcal{P}(\mathbb{N})\backslash\emptyset \times \mathcal{P}(\mathcal{IN}) \times \mathcal{P}(\mathcal{V})$.

There are three configuration activities that can be applied to a feature model: (1) the cardinality of a feature can be narrowed, (2) an instance of a cloned feature can be created and (3) the type of a parameterized feature can be narrowed. Narrowing down the type of a parameterized feature to a singleton means the assignation of a concrete value. Since the following definition of a configuration allows all these configuration activities, this approach can seamlessly be connected with staged configuration [CHE04, CHE05a].

Definition 6.1.7 (Configuration)**.** A configuration of a feature model *FM* is a partial function $C : \mathcal{CFID}_{FM} \rightsquigarrow \mathcal{CA}$ with $C(cfid) = (c, i, t)$ fulfilling the following conditions.

$$c \subseteq Card_{FM}(f(cfid)) \tag{6.9}$$
$$\neg isCloned_{FM}(f(cfid)) \Longrightarrow i = \emptyset \tag{6.10}$$

$$isInst(cfid) \implies c \subseteq \{0,1\} \ \wedge \ i = \emptyset \tag{6.11}$$

$$\neg isParameterized_{FM}(f(cfid)) \implies t = \emptyset \tag{6.12}$$

$$\begin{pmatrix} isParameterized_{FM}(f(cfid)) \\ \wedge\, isCloned_{FM}(f(cfid)) \\ \wedge\, \neg isInst(cfid) \end{pmatrix} \implies t = \emptyset \tag{6.13}$$

$$\begin{pmatrix} isParameterized_{FM}(f(cfid)) \\ \wedge\, \neg isCloned_{FM}(f(cfid)) \end{pmatrix} \implies t \subseteq Type_{FM}(f(cfid)) \tag{6.14}$$

$$\begin{pmatrix} isParameterized_{FM}(f(cfid)) \\ \wedge\, isInst(cfid) \end{pmatrix} \implies t \subseteq Type_{FM}(f(cfid)) \tag{6.15}$$

The set of all configurations of a feature model FM is denoted with $\mathcal{C}_{FM}$.

Condition 6.9 makes sure that the assigned cardinality of a configured feature identifier is valid (i.e. it is narrowed with respect to the cardinality of the feature). Condition 6.10 states that instance creation is only allowed if a configured feature identifier points to a cloned feature. There are no limitations for the set of created instances besides being a subset of all valid instance names. In addition to the creation of instances, the cardinalities of the individual instances of cloned features can be set separately. Concrete instances of cloned features are handled like non-cloned features, i.e. their cardinalities may not be greater than one and instance names may not be assigned to them (condition 6.11). Naturally, the assignment of data values to configured feature identifiers pointing to non-parameterized features or cloned features themselves is not allowed (conditions 6.12 and 6.13). Types of configured feature identifiers pointing to non-cloned parameterized features or instances of cloned parameterized features may only be narrowed with respect to their base types (conditions 6.14 and 6.15). Note that the definition of configurations is more detailed than the one in [Rei08], especially for cloned features and their instances. This is important for this thesis because the validity of configurations plays an important role. Although the validity of configurations is defined later in this section, we can already see that not every configuration respecting this definition is valid, e.g. a cloned feature can have an invalid number of instances.

Again, we introduce some abbreviations for convenience. Let $(c, i, t) \in \mathcal{CA}$ be a configuration activity, C be a configuration and $cfid \in \mathcal{CFID}$ be a configured feature identifier. Then we define the following.

$$\begin{aligned} Card((c,i,t)) &= c \\ Inst((c,i,t)) &= i \\ Type((c,i,t)) &= t \end{aligned}$$

$$\begin{aligned} Card_C(cfid) &= Card(C(cfid)) \\ Inst_C(cfid) &= Inst(C(cfid)) \\ Type_C(cfid) &= Type(C(cfid)) \end{aligned}$$

When dealing with configurations, it is important to know which features are selected and which are deselected. We first define inclusion, exclusion and the unconfigured state, which do not consider the tree-structure of the feature model. We need these concepts for the definition of feature constraint propagation. In the second step, we define selection and deselection depending on the definition of inclusion and exclusion. Note that these terms are defined for configured feature identifiers (i.e. on instance level) and not for features (i.e. on feature level).

Definition 6.1.8 (Inclusion, Exclusion and Unconfigured State)**.** Given a feature model FM with a configuration C and the partial function $\tau_C : \mathcal{CFIDI}_{FM} \rightsquigarrow \{\top, \bot\}$ with

$$\tau_C(cfid) = \top \iff \begin{cases} Card_C(cfid) = \{1\} & \text{if } Card_{FM}(f(cfid))=\{0,1\} \\ \top & \text{if } Card_{FM}(f(cfid))=\{1\} \\ \bot & \text{if } Card_{FM}(f(cfid))=\{0\} \\ \left(\begin{array}{l} Card_C(cfid) = \{1\} \\ \wedge\, i(cfid) \\ \quad \in Inst_C(NoInst(cfid)) \end{array}\right) & \text{if } isCloned_{FM}(f(cfid)) \end{cases}$$

$$\tau_C(cfid) = \bot \iff \begin{cases} Card_C(cfid) = \{0\} & \text{if } Card_{FM}(f(cfid))=\{0,1\} \\ \bot & \text{if } Card_{FM}(f(cfid))=\{1\} \\ \top & \text{if } Card_{FM}(f(cfid))=\{0\} \\ \left(\begin{array}{l} Card_C(cfid) = \{0\} \\ \vee\, i(cfid) \\ \quad \notin Inst_C(NoInst(cfid)) \end{array}\right) & \text{if } isCloned_{FM}(f(cfid)) \end{cases}$$

A configured feature identifier $cfid \in \mathcal{CFIDI}_{FM}$ is...

- included in C if and only if $\tau_C(cfid) = \top$,
- excluded in C if and only if $\tau_C(cfid) = \bot$,
- unconfigured / undefined in C if and only if $\tau_C(cfid)$ is not defined.

For obvious reasons, a configured feature identifier pointing to an optional feature is included if its cardinality is set to one and excluded if it is set to zero. Configured feature identifiers pointing to mandatory features are always included and those pointing to abstract features are always excluded. The definition is more complex for cloned features. An instance of a cloned feature is included if its cardinality is set to one and the instance is created (i.e. the set of created instances of the cloned feature contains the instance name). In contrast, an instance of a cloned feature is excluded if its cardinality is set to zero or the respecting instance is not created (i.e. the set of created instances of the cloned feature does not contain the instance name). The given definition can handle partial configurations: every configured feature identifier being neither included nor excluded is called unconfigured. Note that this definition is more detailed than the one of the original formalization [Rei08]. In addition, we changed the notion of the unconfigured state (in [Rei08] included and

excluded configured feature identifiers with an unconfigured parent were denoted as unconfigured as well).

The following definition of selection and deselection is based on the definition above and respects the tree-structure of the feature model.

Definition 6.1.9 (Selection and Deselection)**.** Given a feature model FM and the partial function $\sigma_C : \mathcal{CFIDI}_{FM} \rightsquigarrow \{\top, \bot\}$ with

$$\begin{aligned} \sigma_C(cfid) = \top &\iff \tau_C(cfid) = \top \\ &\quad \wedge (isRoot_{FM}(f(cfid)) \vee \sigma_C(Parent(cfid)) = \top) \\ \sigma_C(cfid) = \bot &\iff \tau_C(cfid) = \bot \\ &\quad \vee (\neg isRoot_{FM}(f(cfid)) \wedge \sigma_C(Parent(cfid)) = \bot) \end{aligned}$$

A configured feature identifier $cfid \in \mathcal{CFIDI}_{FM}$ is...

- selected in C, written $C \vdash cfid$, if and only if $\sigma_C(cfid) = \top$,
- deselected in C, written $C \nvdash cfid$, if and only if $\sigma_C(cfid) = \bot$.

The recursive definition of selection and deselection is quite obvious. A configured feature identifier is selected if it is included and its parent is selected. It is deselected if it is excluded or its parent is deselected. Configured feature identifiers with an unconfigured predecessor are neither selected nor deselected. Note that a cloned feature itself cannot be marked as included, selected, excluded or deselected – only the instances can. In this definition the parent function ($Parent$) on instance level and not a parent relation ($Parent_{FM}$ or $Parent_{FM}^{Inh}$) on feature level is used since selection, deselection and the unconfigured state are defined on instance level only.

We have already mentioned that not every configured feature identifier has to be configured in a configuration. This leads to the notion of partial and full configurations, which is defined in the next step.

Definition 6.1.10 (Partial and Full Configurations)**.** A configuration C of a feature model FM is called *partial configuration* if and only if

$$\exists cfid \in \mathcal{CFIDI}_{FM} : (\neg C \vdash cfid \wedge \neg C \nvdash cfid) \tag{6.16}$$

$$\vee \, |Type_C(cfid)| > 1 \tag{6.17}$$

$$\vee \, \exists cfid \in \mathcal{CFID}_{FM} : |Card_C(cfid)| > 1 \tag{6.18}$$

Otherwise C is called *full configuration*.

Obviously, a configuration is partial if it contains configured feature identifiers in unconfigured state (condition 6.16). Furthermore, a configuration is partial if the type or the cardinality of a configured feature identifier is not narrowed to a singleton (conditions 6.17 and 6.18). Note that condition 6.18 also has to hold for configured feature identifiers pointing to cloned features themselves and not only for those pointing to their instances. The consideration of cloned features in this definition is more detailed than in [Rei08].

Now we are ready to define the semantics of feature constraints and feature links. Formal semantics are required for the verification of feature constraint propagation

and are not contained in [Rei08]. Feature constraints are formulated on instance level and evaluated with respect to inclusion[2] and feature links are formulated on feature level and evaluated with respect to selection. Hence, we define two distinct assignment functions: one for formulae on instance level and one for formulae on feature level. Subsequently, we map feature links to propositional logic formulae on feature level and use the last-mentioned assignment function for evaluating these formulae.

Definition 6.1.11 (Semantics of Feature Constraints)**.** Given a feature model FM with a configuration C. The assignment of the configured feature identifiers (resp. features) is given by function $B_C^{\mathcal{CFID}} : \mathcal{CFIDI}_{FM} \rightarrow \{\top, \bot\}$ (resp. $B_C^F : F_{FM} \rightarrow \{\top, \bot\}$) with

$$\begin{aligned} B_C^{\mathcal{CFID}}(cfid) &\iff \tau_C(cfid) = \top \\ B_C^F(f) &\iff \exists cfid \in \mathcal{CFIDI}_{FM} : f(cfid) = f \wedge C \vdash cfid \end{aligned}$$

The configuration C satisfies a feature constraint $\varphi \in Form(\mathcal{CFIDI}_{FM})$ (resp. $\varphi \in Form(F_{FM})$), written $C \models \varphi$, if and only if $B_C^{\mathcal{CFID}^*}(\varphi) = \top$ (resp. $B_C^{F^*}(\varphi) = \top$), where $B_C^{\mathcal{CFID}^*} : Form(\mathcal{CFIDI}_{FM}) \rightarrow \{\top, \bot\}$ (resp. $B_C^{F^*} : Form(F_{FM}) \rightarrow \{\top, \bot\}$) is the induced evaluation function. Otherwise C does not satisfy φ, written $C \not\models \varphi$.

For convenience, we define the following abbreviations for a set of constraints.

$$\begin{aligned} C \models Constraints_{FM} &\iff \forall \varphi \in Constraints_{FM} : C \models \varphi \\ C \not\models Constraints_{FM} &\iff \exists \varphi \in Constraints_{FM} : C \not\models \varphi \end{aligned}$$

In principle, the expression of arbitrary propositional logic constraints on instance level and on feature level is possible. Nevertheless, we address only three popular types of constraints on feature level, according to [Rei08, Rei09]: `needs`, `excludes` and `alternative` feature links.

Definition 6.1.12 (Semantics of Feature Links)**.** Given a feature model FM with feature links $FLinks_{FM} = \{excludes_{FM}, alternative_{FM}, needs_{FM}\}$ and a configuration C of FM. The configuration C satisfies a feature link $(X, Y) \in excludes_{FM}$ (resp. $(X, Y) \in alternative_{FM}$, $(X, Y) \in needs_{FM}$) if and only if $C \models (X\, excludes_{FM}\, Y)$ (resp. $C \models (X\, alternative_{FM}\, Y)$, $C \models (X\, needs_{FM}\, Y)$), defined by

$$\begin{aligned} C \models (X\, excludes_{FM}\, Y) &\iff C \models \neg(X \wedge Y) \\ C \models (X\, alternative_{FM}\, Y) &\iff C \models (X \veebar Y) \equiv (X \wedge \neg Y) \vee (\neg X \wedge Y) \\ C \models (X\, needs_{FM}\, Y) &\iff C \models (X \rightarrow Y) \end{aligned}$$

[2] Remember that the formalization does not provide an abbreviation for selection.

Analogously as for feature constraints, we define some abbreviations for feature links.

$$\begin{aligned} C \not\models (X\, excludes_{FM}\, Y) &\iff \neg C \models (X\, excludes_{FM}\, Y) \\ C \not\models (X\, alternative_{FM}\, Y) &\iff \neg C \models (X\, alternative_{FM}\, Y) \\ C \not\models (X\, needs_{FM}\, Y) &\iff \neg C \models (X\, needs_{FM}\, Y) \end{aligned}$$

$$\begin{aligned} C \models excludes_{FM} &\iff \forall (X,Y) \in excludes_{FM} : C \models (X\, excludes_{FM}\, Y) \\ C \not\models excludes_{FM} &\iff \exists (X,Y) \in excludes_{FM} : C \not\models (X\, excludes_{FM}\, Y) \\ C \models alternative_{FM} &\iff \forall (X,Y) \in alternative_{FM} : C \models (X\, alternative_{FM}\, Y) \\ C \not\models alternative_{FM} &\iff \exists (X,Y) \in alternative_{FM} : C \not\models (X\, alternative_{FM}\, Y) \\ C \models needs_{FM} &\iff \forall (X,Y) \in needs_{FM} : C \models (X\, needs_{FM}\, Y) \\ C \not\models needs_{FM} &\iff \exists (X,Y) \in needs_{FM} : C \not\models (X\, needs_{FM}\, Y) \end{aligned}$$

$$\begin{aligned} C \models FLinks_{FM} &\iff \begin{pmatrix} C \models excludes_{FM} \\ \wedge\ C \models alternative_{FM} \\ \wedge\ C \models needs_{FM} \end{pmatrix} \\ C \not\models FLinks_{FM} &\iff \neg C \models FLinks_{FM} \end{aligned}$$

After all, we still have to define the validity of configurations. This definition is more detailed than the one of [Rei08]. It clearly separates between inclusion and selection semantics. In addition, it takes the number of selected instances of cloned features, group cardinalities and the fulfillment of constraints into account. For this purpose, we initially define the set of all included instances of a configured feature identifier. This set is defined for a given configuration C and a configured feature identifier *cfid* that points to a cloned feature itself (not to an instance) and that is configured by the configuration C. It contains all in C included configured feature identifiers pointing to an instance of the feature that *cfid* points to.

Definition 6.1.13 (The Set of all Included Instances of a Configured Feature Identifier)**.** For a configured feature identifier $cfid \in \mathcal{CFID}_{FM}$ of a feature model FM with $isCloned_{FM}(f(cfid))$, $\neg isInst(cfid)$ and $C(cfid)$ is defined, the set of all included instances of this configured feature identifier with respect to configuration C is given by

$$\mathcal{CFID}_{C,cfid} = \left\{ cfid' \in \mathcal{CFID}_{FM} \;\middle|\; \begin{pmatrix} cfid' =_f cfid \\ \wedge\ isInst(cfid') \\ \wedge\ \tau_C(cfid') = \top \end{pmatrix} \right\}$$

Definition 6.1.14 (Validity of Configurations)**.** A configuration C of a feature model FM is called *valid* if the following conditions hold.

1. All cloned features which are root features or whose parents are selected have a valid number of instances:

for all $cfid \in \mathcal{CFID}_{FM}$ with the conditions $isCloned_{FM}(f(cfid))$, $\neg isInst(cfid)$ and $(isRoot_{FM}(cfid) \vee C \vdash Parent(cfid))$, it holds that

$$\begin{aligned} |\mathcal{CFID}_{C,cfid}| &\in Card_C(cfid) && \text{if } C(cfid) \text{ is defined} \\ 0 &\in Card_{FM}(f(cfid)) && \text{if } C(cfid) \text{ is not defined} \end{aligned}$$

2. All groups with a selected parent have a valid number of selected members: for every feature group $GP \in F_{FM_{/Group_{FM}}} \setminus \{M| \ |M| = 1\}$ the set of parents is defined as $parents_{GP} = \{p \in F_{FM} | \exists f \in F_{FM} : f \in GP \wedge (p, f) \in Parent_{FM}^{Inh}\}$ (note that a root feature cannot belong to a group). For every selected configured feature identifier $cfid_p \in \{cfid \in \mathcal{CFIDI}_{FM} | f(cfid) \in parents_{GP} \wedge C \vdash cfid\}$ the group cardinality of its children has to be valid: $|\{cfid \in \mathcal{CFIDI}_{FM} | Parent(cfid) = cfid_p \wedge f(cfid) \in GP \wedge C \vdash cfid\}| \in Card_{FM}(GP)$

3. C fulfills all feature links of FM:
   $C \models FLinks_{FM}$

4. C fulfills all feature constraints of FM:
   $C \models Constraints_{FM}$

Otherwise C is called *invalid*.

This definition also respects feature inheritance and cloned features since the parent function $Parent_{FM}^{Inh}$ to identify all parents of a feature on feature level and the parent function $Parent$ to identify the parent of a feature on instance level are used.

The first two conditions have to hold only for features that are root features or whose parents are selected. Roughly spoken, if a feature is not selected, we do not care about the cardinalities of its children. The major idea of these conditions is that selection semantics plays the main role in practice. Inclusion of features is only used as an intermediate state when using configuration links. All successors of excluded features are deselected anyway. From a practical point of view, they are not present. This definition is consistent to the concept of [Rei08] and also implemented in the feature modeling tool CVM, which is introduced in Section 7.1.1.

Note that feature constraint propagation only addresses Nº 3 and Nº 4 above because only feature links and feature constraints are propagated (cf. Section 2.4).

6.2 Formal Definition of Configuration Links

According to the formal definition of feature models given in Section 6.1, we present a formalization of configuration links in this section. This formalization is based on the formalization in [Rei08]. Although most definitions are similar to the original ones, some properties are formulated more detailed and others are defined in a different way. Major changes are marked. Moreover, some properties of the given definitions are proven with regard to the correctness and minimality proof of feature constraint propagation.

The first definition in this section introduces the concept of configuration steps. A configuration step can be seen as an atomic step of an overall configuration activity, for example setting the cardinality of a configured feature identifier or assigning a set of created instances to a configured feature identifier.

Definition 6.2.1 (Configuration Step)**.** A configuration step for a feature model FM is defined as $cs_{FM} = (cfid, c, i, t)$ with $cfid \in \mathcal{CFID}_{FM}, (c, i, t) \in \mathcal{CA}$, where the following conditions hold.

$$c \subseteq Card_{FM}(f(cfid)) \tag{6.19}$$

$$\neg isCloned_{FM}(f(cfid)) \implies i = \emptyset \tag{6.20}$$

$$isInst(cfid) \implies c \subseteq \{0, 1\} \ \wedge \ i = \emptyset \tag{6.21}$$

$$\neg isParameterized_{FM}(f(cfid)) \implies t = \emptyset \tag{6.22}$$

$$\left(\begin{array}{l} isParameterized_{FM}(f(cfid)) \\ \wedge\, isCloned_{FM}(f(cfid)) \\ \wedge\, \neg isInst(cfid) \end{array}\right) \implies t = \emptyset \tag{6.23}$$

$$\left(\begin{array}{l} isParameterized_{FM}(f(cfid)) \\ \wedge\, \neg isCloned_{FM}(f(cfid)) \end{array}\right) \implies t \subseteq Type_{FM}(f(cfid)) \tag{6.24}$$

$$\left(\begin{array}{l} isParameterized_{FM}(f(cfid)) \\ \wedge\, isInst(cfid) \end{array}\right) \implies t \subseteq Type_{FM}(f(cfid)) \tag{6.25}$$

The set of all configuration steps of a feature model FM is denoted with $\mathcal{CS}_{FM}$.

Note that the conditions of this definition (conditions 6.19 to 6.25) are equivalent to those of configurations (conditions 6.9 to 6.15). The ideas behind these conditions have already been described in Section 6.1 under the definition of configurations (Definition 6.1.7). This means that a single configuration activity contains the same information as a configuration for a single configured feature identifier. The reason for the separate definition is that several (possibly contradicting) configuration steps can be formulated for one configured feature identifier. The mapping of a configured feature identifier to more than one configuration step is not possible with a function because of the right-uniqueness. Note that the formalization of configuration steps does not provide abbreviations for selection or deselection of configured feature identifiers.

For a configuration step $(cfid, c, i, t) \in \mathcal{CS}$, we introduce the following abbreviations.

$$\begin{aligned} Card_{cs}((cfid, c, i, t)) &= Card((c, i, t)) \\ Inst_{cs}((cfid, c, i, t)) &= Inst((c, i, t)) \\ Type_{cs}((cfid, c, i, t)) &= Type((c, i, t)) \end{aligned}$$

Now we define the combination of configuration steps. We have already mentioned that configuration links can deal with redundancies and contradicting configuration steps. The next definition states how configuration steps are combined and in which way contradictions between them are resolved.

Definition 6.2.2 (Combination of Configuration Steps)**.** Given two configuration steps $cs, cs' \in \mathcal{CS}_{FM}$ of a feature model FM. Then the combination of them is given by the partial function $+_{\mathcal{CS}} : \mathcal{CS}_{FM} \times \mathcal{CS}_{FM} \rightsquigarrow \mathcal{CS}_{FM}$ with

$$(cfid, c, i, t) +_{\mathcal{CS}} (cfid, c', i', t') = (cfid, \hat{c}, \hat{i}, \hat{t})$$

where the following conditions hold.

$$\hat{c} = \begin{cases} c \cap c' & |c \cap c' \neq \emptyset \\ \{0\} & |\text{else} \end{cases} \tag{6.26}$$

$$\hat{i} = i \cup i' \tag{6.27}$$

$$\hat{t} = \begin{cases} Type_{FM}(f(cfid)) & |t \cap t' = \emptyset \wedge isParameterized_{FM}(f(cfid)) \\ t \cap t' & |\text{else} \end{cases} \tag{6.28}$$

The combination of configuration steps is given by a partial function since only configuration steps for the same configured feature identifier can be combined. Condition 6.26 prioritizes an exclude over an include: consider two configuration steps for a configured feature identifier *cfid* – one includes *cfid* (i.e. sets the cardinality to one), one excludes *cfid* (i.e. sets the cardinality to zero). The combination of these configuration steps would exclude *cfid* since the intersection of $\{0\}$ and $\{1\}$ is empty. Instance sets are simply combined by union (condition 6.27). When combining configuration steps for parameterized features, the intersection of the values is created (condition 6.28) if it is not empty. Otherwise the base type is assigned. Note that values can only be assigned for parameterized features (see condition 6.22), i.e. $\hat{t} = t \cap t' = \emptyset$ always holds by condition 6.28 for non-parameterized features.

We allow the following capital-sigma notations.

$$\sum_{i=n}^{m} cs_i = cs_n +_{\mathcal{CS}} cs_{n+1} +_{\mathcal{CS}} ... +_{\mathcal{CS}} cs_m \qquad \text{for } n < m \in \mathbb{N}$$

$$\sum_{cs \in CS} cs = cs_1 +_{\mathcal{CS}} cs_2 +_{\mathcal{CS}} ... +_{\mathcal{CS}} cs_n \qquad \text{for } CS = \{cs_1, cs_2, \ldots, cs_n\} \subseteq \mathcal{CS}$$

For proving correctness and minimality of feature constraint propagation, we need some properties of the combination operation for configuration steps. It is obvious that $+_{\mathcal{CS}}$ is commutative and associative since it is defined with commutative and associative set operations. Some additional, for the mentioned proof required, properties of the combination operation are formulated and proved in the next lemma.

Lemma 6.2.3 (Combination of Multiple Configuration Steps)**.** *Given a set of configuration steps CS_{cfid} of a feature model FM for the configured feature identifier $cfid \in \mathcal{CFIDI}_{FM}$, i.e. $\forall (cfid', c', i', t') \in CS_{cfid} : cfid' = cfid$. Then the following statements hold.*

1. *For every non-empty subset $CS'_{cfid} \subseteq CS_{cfid}$ it holds that*

$$Card_{cs}(\sum_{cs \in CS'_{cfid}} cs) = \{0\} \Longrightarrow Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{0\}$$

2.

$$Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{1\}$$
$$\Longleftrightarrow$$
$$\begin{pmatrix} \forall (cfid, c, i, t) \in CS_{cfid} : 1 \in c \\ \wedge \exists (cfid, c, i, t) \in CS_{cfid} : c = \{1\} \end{pmatrix}$$

3.

$$Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{0\}$$
$$\Longleftrightarrow$$
$$\exists (cfid, c, i, t) \in CS_{cfid} : c = \{0\}$$

4. If $CS_{cfid} \neq \emptyset$ *then*

$$Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{0, 1\}$$
$$\Longleftrightarrow$$
$$\forall (cfid, c, i, t) \in CS_{cfid} : c = \{0, 1\}$$

Proof. For the following proofs, note that $\{0\}$, $\{0, 1\}$ and $\{1\}$ are the only allowed cardinalities for configured feature identifiers of the set $\mathcal{CFIDI}_{FM}$ (see Definition 6.1.4 and Definition 6.2.1).

1. Let $Card_{cs}(\sum_{cs \in CS'_{cfid}} cs) = \{0\}$ hold. Since $+_{\mathcal{CS}}$ is associative and commutative $\sum_{cs \in CS_{cfid}} cs = \sum_{cs \in CS'_{cfid}} cs +_{\mathcal{CS}} \sum_{cs \in CS_{cfid} \setminus CS'_{cfid}} cs$ holds. Because we know that $Card_{cs}(\sum_{cs \in CS'_{cfid}} cs) = \{0\}$, it remains to show that $Card_{cs}((cfid, \{0\}, i, t) +_{\mathcal{CS}} (cfid, c', i', t')) = \{0\}$. This is trivial since $c' \in \mathcal{P}(\{0, 1\}) \setminus \emptyset$.

2. Assume that $Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{1\}$. By Definition 6.2.2, we know that this is the case if and only if the intersection of the cardinality sets of all configuration steps of CS_{cfid} is $\{1\}$. This holds if and only if at least one configuration step $(cfid, c, i, t) \in CS_{cfid}$ with $c = \{1\}$ exists, i.e. $\exists (cfid, c, i, t) \in CS_{cfid} : c = \{1\}$, and all other configuration steps of the set CS_{cfid} have the cardinality $\{0, 1\}$ or $\{1\}$, i.e. $\forall (cfid, c, i, t) \in CS_{cfid} : 1 \in c$.

3. "$\Longrightarrow$". We show the contraposition. Assume $\nexists (cfid, c, i, t) \in CS_{cfid} : c = \{0\}$. This is equivalent to $\forall (cfid, c, i, t) \in CS_{cfid} : c \in S$ with $S = \{\{1\}, \{0, 1\}\}$. Obviously, the intersection of two sets of S cannot be empty and is an element of S itself. When combining of two configuration steps of CS_{cfid}, the new cardinality is therefore always constructed as intersection (see Definition 6.2.2), i.e. it is an element of S itself. Therefore, $Card_{cs}(\sum_{cs \in CS_{cfid}} cs) \in S$, i.e. $Card_{cs}(\sum_{cs \in CS_{cfid}} cs) \neq \{0\}$.

"$\Longleftarrow$". Let $\exists cs = (cfid, c, i, t) \in CS_{cfid} : c = \{0\}$ hold. Then, for all configuration steps $cs' = (cfid, c', i', t') \in \mathcal{CS}_{FM}$ (of the feature model) holds that $Card_{cs}(cs +_{\mathcal{CS}} cs') = \{0\}$: if $0 \in c'$ then $c \cap c' = \{0\}$ and if $0 \notin c'$ then the cardinality of $cs +_{\mathcal{CS}} cs'$ is explicitly defined as $\{0\}$ (see Definition 6.2.2).

4. Assume that $Card_{cs}(\sum_{cs \in CS_{cfid}} cs) = \{0, 1\}$. By Definition 6.2.2, we know that this is the case if and only if the intersection of the cardinality sets of all configuration steps of CS_{cfid} is $\{0, 1\}$. This, in turn, holds if and only if all configuration steps of the set CS_{cfid} have the cardinality $\{0, 1\}$, i.e. $\forall (cfid, c, i, t) \in CS_{cfid} : c = \{0, 1\}$.

□

The next presented concept – the configuration decisions – allows the formulation of (1) complex configuration activities and (2) criteria for the application of the corresponding configuration activities. Criteria of configuration decisions refer to a different feature model than configuration activities. So the concept of configuration decisions is an essential step towards the definition of configuration links.

Definition 6.2.4 (Configuration Decision)**.** Given feature models S and T with $F_S \cap F_T = \emptyset$. A configuration decision $cd_{S \to T}$ is defined as pair $cd_{S \to T} = (\varphi, Steps)$ with

$$\varphi \in Form(\mathcal{CFIDI}_S)$$
$$Steps \subseteq \mathcal{CS}_T$$

where φ is denoted as *criterion* and *Steps* as *effect* or *configuration steps* that will be applied to a configuration if the criterion is fulfilled (see Definition 6.1.11).
The set $\mathcal{CD}_{S \to T}$ contains all configuration decisions from S to T.

It is an important point that the set of configuration steps of a configuration decision is arbitrary. This set can theoretically contain contradicting and redundant configuration steps. In the original formalization [Rei08], the criterion is formalized as set of configurations. This is a different point of view to the criterion but it is equivalent to the formalization given here.

The main concept of this section – the configuration links – can now simply be defined as set of configuration decisions.

Definition 6.2.5 (Configuration Link)**.** A configuration link $CL_{S \to T}$ from feature model S to feature model T is a set of configuration decisions $CL_{S \to T} \subseteq \mathcal{CD}_{S \to T}$.
$\mathcal{CL}_{S \to T}$ denotes the set of all configuration links from S to T.

Note that this definition only covers expanded configuration links (see Definition 5.1.1) since the formalization does not contain abbreviations for selection and deselection statements.

Apart from this definition, there is also a functional view on configuration links. A configuration link can be considered as function between configuration sets of feature models. This function maps a source configuration to a target configuration. We define the application of configuration links in a different manner than [Rei08].

At first, we define the set of applied configuration steps for configured feature identifiers. This set contains all configuration steps that are applied during the application of the configuration link. The motivation for a separate definition of this set is that we need it later for the verification of the propagation.

Definition 6.2.6 (Set of Applied Configuration Steps of a Configuration Link for a Configured Feature Identifier)**.** Given two feature models S and T, a configuration C_S of S, a configuration link $CL_{S\to T}$ from S to T and a configured feature identifier $cfid_T \in \mathcal{CFID}_T$. Then the set of applied configuration steps $\mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T)$ of $CL_{S\to T}$ for $cfid_T$ is given by

$$\mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T) = \left\{ cs \in \mathcal{CS}_T \;\middle|\; \begin{array}{l} (\varphi, Steps) \in CL_{S\to T} \\ \wedge C_S \models \varphi \\ \wedge cs = (cfid_T, c, i, t) \in Steps \end{array} \right\}$$

Equipped with this definition, we are finally ready to define the semantics of configuration links – their application to a configuration. Therefore, we define two auxiliary functions: *Cfg* and $\mathcal{X}$. *Cfg* is defined with respect to a configuration link and describes how a configured feature identifier of the target model is configured in the derived target configuration. It maps the source configuration and the configured feature identifier of the target model to the resulting configuration activity for this configured feature identifier. This function contains the combination of all configuration steps (for individual configured feature identifiers) that have to be applied to a single configuration step, as introduced in Definition 6.2.2. The second auxiliary function $\mathcal{X}$ defines how a complete target configuration is derived from a given source configuration. It maps a configuration link and a source configuration to the resulting target configuration. Note that a configuration is, according to Definition 6.1.7, a partial function. A (partial) function can be seen as a set of tuples.

Definition 6.2.7 (Application of a Configuration Link)**.** Given two feature models S and T, a configuration C_S of S, a configuration link $CL_{S\to T}$ from S to T and the following two auxiliary functions. $Cfg_{CL_{S\to T}} : \mathcal{C}_S \times \mathcal{CFID}_T \rightsquigarrow \mathcal{CA}$ with

$$Cfg_{CL_{S\to T}}(C_S, cfid_T) = (c, i, t) \text{ where}$$
$$(cfid_T, c, i, t) = \sum_{cs \in \mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T)} cs$$

defined for all C_S and $cfid_T$ with $\mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T) \neq \emptyset$ and $\mathcal{X} : \mathcal{CL}_{S\to T} \times \mathcal{C}_S \to \mathcal{C}_T$ with

$$\mathcal{X}(CL_{S\to T}, C_S) = \left\{ (\ cfid_T, Cfg_{CL_{S\to T}}(C_S, cfid_T)\ ) \;\middle|\; \begin{array}{l} cfid_T \in \mathcal{CFID}_T \\ \wedge Cfg_{CL_{S\to T}}(C_S, cfid_T) \\ \text{is defined} \end{array} \right\}$$

Then the application of a configuration link $CL_{S\to T}(C_S)$ is defined by the following.

$$CL_{S\to T}(C_S) = \mathcal{X}(CL_{S\to T}, C_S)$$

We say that $CL_{S \to T}(C_S)$ is the target configuration derived through the application of the configuration link to source configuration C_S. Note that the application of a configuration link has functional behavior (i.e. it is left-total and right-unique), although the auxiliary function $Cfg_{CL_{S \to T}}$ is partial (i.e. not necessarily left-total).

6.3 Formal Definition and Verification of Feature Constraint Propagation

Up to now, the basic idea of feature constraint propagation has been presented in Chapter 4 and the concept has been introduced in detail in Chapter 5. At this point, we present the formal foundation of feature constraint propagation. The formalization relates to the formal definitions of feature models in Section 6.1 and configuration links in Section 6.2.

As already mentioned in Section 5.1, the propagation consists of six steps. First of all the configuration link is expanded. In this phase, the abbreviations for selection ([+]) and deselection ([−]) are eliminated and replaced by equivalent expressions that contain only inclusion and exclusion statements. Since this can also be done straightforwardly for feature constraints, we do not provide corresponding abbreviations for selection and deselection in the formalization. The next two steps are the calculation of the reverse mappings and the transformation of logical formulae. These steps are covered by the formalization. An explicit (formal) construction of the reverse mappings is given in this section and the transformation, which is based on these mappings, is formally defined. In addition, correctness and minimality of the transformation are proven. In the last three steps of the propagation (the minimization of the resulting formulae, the lifting of inclusion to selection statements and the extraction of feature links) the resulting formulae are restructured. This process is essential for practical use of the propagation since resulting constraints of transformations are usually very long and complex and, therefore, not manageable for the user. The restructuring of formulae ensures readability and comprehensibility of resulting constraints. Nevertheless, it is irrelevant for this section. In the formalization, we deal with equivalence classes of constraints. The equivalence class of a constraint is, in this context, the set of propositional logic formulae being logically equivalent to the constraint. From a theoretical point of view, it does not matter which formula of an equivalence class is chosen since all of them are logically equivalent. We do not consider the structure of the formula in this context. Summing up, we formalize and verify the reverse mappings and the transformation in this section and neglect the other steps of propagation for the reasons above.

Before we start with the formalization, we adapt the properties correctness and minimality (cf. Section 1.2) to distinct sets of feature constraints and feature links and to the transformation step of feature constraint propagation. Figure 6.1 illustrates these properties. This depicted situation is equal to Figure 1.3, however, sets of feature constraints and sets of feature links are considered instead of single constraints. In addition, only the transformation step and not the whole propagation is taken into account. Correctness and minimality are defined in the already known way. Correctness means that the satisfaction of all transformed feature con-

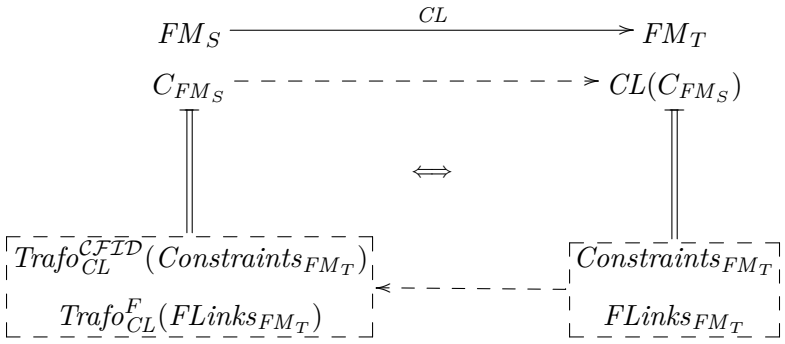


Figure 6.1: Correctness and minimality of the transformation

straints and feature links is a sufficient condition for the satisfaction of all original feature constraints and feature links. Vice versa, minimality means that it is also a necessary condition. Correctness and minimality are defined analogously for the individual reverse mappings.

In order to define the reverse mappings (cf. Section 5.1.2), we need some additional definitions. We start with the set of configuration steps of a configuration decision for a configured feature identifier. The idea of this set is similar to the one for the set of applied configuration steps of a configuration link for a configured feature identifier (see Definition 6.2.6): we collect all configuration steps that point to a certain configured feature identifier. Although, the next definition relates to a single configuration decision (and not the whole configuration link) and the fulfillment of the criterion is not considered.

Definition 6.3.1 (Set of Configuration Steps of a Configuration Decision for a Configured Feature Identifier)**.** Given two feature models S and T, a configuration link $CL_{S\to T}$ from S to T, a configuration decision $cd \in CL_{S\to T}$ and a configured feature identifier $cfid_T \in \mathcal{CFID}_T$. Then the set of configuration steps $\mathcal{CS}_{CL_{S\to T}}(cd, cfid_T)$ for $cfid_T$ of cd is given by

$$\mathcal{CS}_{CL_{S\to T}}(cd, cfid_T) = \left\{ cs \in \mathcal{CS}_T \;\middle|\; \begin{array}{c} cd = (\varphi, Steps) \\ \wedge cs = (cfid_T, c, i, t) \in Steps \end{array} \right\}$$

Apparently, there is a relationship between the set of configuration steps of a configuration decision (see Definition 6.3.1) and the set of configuration steps of a configuration link (see Definition 6.2.6) for a configured feature identifier. We formulate this relationship in the following lemma because we need it for the minimality and correctness proof of the transformation.

Lemma 6.3.2 (Coherency of Sets of Configuration Steps)**.** *Given a configuration link $CL_{S\to T}$ from S to T, a configuration decision $cd = (\varphi, Steps) \in CL_{S\to T}$, a*

configured feature identifier $cfid_T \in \mathcal{CFID}_T$ *and a configuration* $C_S \in \mathcal{C}_S$ *with* $C_S \models \varphi$*. Then the following holds.*

$$\mathcal{CS}_{CL_{S\to T}}(cd, cfid_T) \subseteq \mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T)$$

Proof. For proving the subset relation between the sets, it is sufficient to prove that $cs \in \mathcal{CS}_{CL_{S\to T}}(cd, cfid_T)$ implies $cs \in \mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T)$ for every cs. Given $cs \in \mathcal{CS}_{CL_{S\to T}}(cd, cfid_T)$. Then it holds by Definition 6.3.1 that $cs = (cfid_T, c, i, t) \in Steps$. Since $cd = (\varphi, Steps) \in CL_{S\to T}$ and $C_S \models \varphi$ hold by assumption, it follows directly that $cs \in \mathcal{CS}_{CL_{S\to T}}(C_S, cfid_T)$ by Definition 6.2.6. □

Remember that our first goal in this section is the definition of the three reverse mappings of Section 5.1.2. For this purpose, we need two auxiliary functions: the reverse cardinality mapping and the reverse instance mapping. The former function gets a configured feature identifier and a natural number as input and delivers a subset of the criteria of the underlying configuration link. Every criterion of this subset belongs to a configuration decision setting the cardinality of the given configured feature identifier to the given natural number. The last-mentioned function, the reverse instance mapping, gets a configured feature identifier that points to an instance of a cloned feature as input and delivers all criteria of configuration decisions that create the instance of the cloned feature which the given configured feature identifier points to.

Definition 6.3.3 (Reverse Cardinality Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the reverse cardinality mapping $Rev_{CL_{S\to T}} : \mathcal{CFIDI}_T \times \mathbb{N} \to \mathcal{P}_{fin}(Form(\mathcal{CFIDI}_S))$ is defined by

$$Rev_{CL_{S\to T}}(cfid_T, n) = \left\{ \varphi \in Form(\mathcal{CFIDI}_S) \;\middle|\; \begin{array}{l} cd = (\varphi, Steps) \in CL_{S\to T} \\ \wedge Card_{cs}\left(\sum_{cs \in \mathcal{CS}_{CL_{S\to T}}(cd, cfid_T)} cs \right) = \{n\} \end{array} \right\}$$

In order to identify all configuration decisions setting the cardinality of a configured feature identifier to a given number, we combine all configuration steps (see Definition 6.2.2) of a configuration decision for this configured feature identifier (see Definition 6.3.1). If the cardinality of the resulting sum is a singleton containing the given number, then the configuration decision is important in this case and, consequently, its criterion is contained in the resulting set. The summation of all corresponding configuration steps is necessary and we cannot simply look for single configuration steps setting the cardinality to the given number since contradicting configuration steps can be contained in a configuration decision. This definition resolves contradictions in the individual configuration decisions but it does not consider contradictions between several configuration decisions. These contradictions are considered later. Note that the reverse cardinality mapping is a total function, i.e. it can be constructed for all configured feature identifiers and all natural numbers.

The following definition of the reverse instance mapping is similar to the previous definition of the reverse cardinality mapping.

Definition 6.3.4 (Reverse Instance Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}'_T$ with $\mathcal{CFIDI}'_T = \{cfid \in \mathcal{CFIDI}_T \mid isCloned(f(cfid))\}$. Then the reverse instance mapping $Rev'_{CL_{S\to T}} : \mathcal{CFIDI}'_T \to \mathcal{P}_{fin}(Form(\mathcal{CFIDI}_S))$ is defined by

$$Rev'_{CL_{S\to T}}(cfid_T) = \left\{ \varphi \in Form(\mathcal{CFIDI}_S) \;\middle|\; \begin{array}{l} cd = (\varphi, Steps) \in CL_{S\to T} \\ \wedge i(cfid_T) \in Inst_{cs}\left(\sum_{cs\in\mathcal{CS}_{CL_{S\to T}}(cd,NoInst(cfid_T))} cs\right) \end{array} \right\}$$

Above, we argued that the summation of all configuration steps of configuration decisions for a configured feature identifier is necessary to calculate its resulting cardinality. There are some differences when considering the instance creation since the instance sets are unified by the combination operation (see Definition 6.2.2) in contrast to the cardinality sets. Therefore, it is sufficient for the instance creation if a configuration decision contains one configuration step that creates the corresponding instance. Other configuration steps cannot revoke the instance creation. Nevertheless, the definition of the reverse instance mapping uses the summation. The alternative way (i.e. the definition without the sum) provides no advantages compared to this definition.

The results of both so far introduced mappings are sets of logical formulae. Certainly, we want to deal with single logical formulae. Hence, we define a function that combines all formulae of a formula set to a single formula by constructing their disjunction. In this formalization, we only need the disjunction of formula sets and not their conjunction.

Definition 6.3.5 (Disjunction of Formula Sets)**.** Given an alphabet X. Then $\sqcup : \mathcal{P}_{fin}(Form(X)) \to Form(X)_{/\equiv}$ is defined by

$$\begin{aligned} \sqcup(\emptyset) &= [\bot]_{\equiv} \\ \sqcup(\{\varphi_1, \varphi_2, ..., \varphi_n\}) &= [\varphi_1 \vee \varphi_2 \vee ... \vee \varphi_n]_{\equiv} \qquad \text{for } n \in \mathbb{N}\backslash\{0\} \end{aligned}$$

Note that the codomain of the function above is not, as might be expected, the set of logical formulae but the quotient set with respect to the equivalence relation $\equiv$. This relation denotes the logical equivalence of formulae. The fact that $\equiv$ is an equivalence relation is well-known and trivial to show: it is reflexive, symmetrical and transitive. The quotient set is the set of all equivalence classes. An equivalence class with respect to $\equiv$ contains only equivalent formulae. The construction of equivalence classes is an important step for the definition of the transformation since, from a theoretical point of view, we are not interested in the structure of the formulae but in its semantics only.

Remark 6.3.6. In the following, we allow the application of logical operators to equivalence classes of logical formulae as abbreviation.

$$\begin{aligned} \neg[\varphi]_{\equiv} &= [\neg\varphi]_{\equiv} \\ [\varphi_1]_{\equiv} \otimes [\varphi_2]_{\equiv} &= [\varphi_1 \otimes \varphi_2]_{\equiv} \qquad \text{for } \otimes \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \end{aligned}$$

Finally, we are ready to define the reverse mappings with respect to a configuration link. We already know that we distinguish three kinds of mappings: the reverse inclusion mapping of configured feature identifiers, the reverse selection mapping of configured feature identifiers and the reverse selection mapping of features (cf. Section 5.1.2). The input of the former two mappings is a configured feature identifier of the target feature model – they form the basis for the transformation of feature constraints. The last-mentioned mapping maps features instead of configured feature identifiers – it forms the basis for the transformation of feature links. In contrast to the definition of the mappings in Section 5.1.2, all three mappings deliver equivalence classes of source-side feature constraints. After each introduction of a mapping, its correctness and minimality is shown.

Definition 6.3.7 (Reverse Inclusion Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\rightarrow T}$. Then the reverse inclusion mapping of configured feature identifiers with respect to this configuration link is given by function $RevInc^{\mathcal{CFID}}_{CL_{S\rightarrow T}} : \mathcal{CFIDI}_T \rightarrow Form(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$RevInc^{\mathcal{CFID}}_{CL_{S\rightarrow T}}(cfid_T) = \begin{cases} [\top]_{\equiv} & \text{if } Card_T(f(cfid_T))=\{1\} \\ [\bot]_{\equiv} & \text{if } Card_T(f(cfid_T))=\{0\} \\ \left(\begin{array}{r} \sqcup\,(Rev_{CL_{S\rightarrow T}}(cfid_T, 1)) \\ \wedge\neg \sqcup\,(Rev_{CL_{S\rightarrow T}}(cfid_T, 0)) \end{array}\right) & \text{if } Card_T(f(cfid_T))=\{0,1\} \\ \left(\begin{array}{r} \sqcup\,(Rev_{CL_{S\rightarrow T}}(cfid_T, 1)) \\ \wedge\neg \sqcup\,(Rev_{CL_{S\rightarrow T}}(cfid_T, 0)) \\ \wedge \sqcup\,(Rev'_{CL_{S\rightarrow T}}(cfid_T)) \end{array}\right) & \text{if } isCloned_T(f(cfid_T)) \end{cases}$$

The former two cases of this definition are obvious: configured feature identifiers pointing to mandatory (resp. abstract) features are always included (resp. excluded). The mapping for optional features is more complex. A configured feature identifier pointing to an optional feature is included if at least the criterion of one configuration decision that includes this configured feature identifier (i.e. sets its cardinality to one) is fulfilled and no criterion of a configuration decision that excludes this configured feature identifier (i.e. sets its cardinality to zero) is fulfilled. For cloned features, at least the criterion of one configuration decision that creates the instance has to be additionally fulfilled. We have already mentioned that the creation of an instance cannot be revoked by other configuration decisions. Note that in the last-mentioned two cases the combination of equivalence classes also results in equivalence classes (see Remark 6.3.6). Summing up, the application of the reverse inclusion mapping to a configured feature identifier delivers a set of equivalent formulae whose satisfaction is sufficient for the inclusion of the given configured feature identifier. The

reverse cardinality mapping (presented in Definition 6.3.3 and used in the definition above) resolves, as already mentioned, contradictions between configuration steps in the individual configuration decisions. The reverse inclusion mapping additionally resolves contradictions between different configuration decisions.

Now we show the correctness and minimality of this mapping in the following lemma.

Lemma 6.3.8 (Correctness and Minimality of the Reverse Inclusion Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows Form(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff \tau_{CL_{S\to T}(C_S)}(cfid_T) = \top$$

Proof. Note that we omit the index $S \to T$ of $CL_{S\to T}$. Let C_S be an arbitrary configuration of S, $cfid_T \in \mathcal{CFIDI}_T$ be a configured feature identifier and π be the bijective projection for an arbitrary complete system of representatives R.

We prove this property in two steps. At first we prove the only-if-part and then the if-part.

Only-if-part ("$\Longleftarrow$"). Let $\tau_{CL(C_S)}(cfid_T) = \top$ hold.
We have to show that $C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL}(cfid_T)$. According to the definition of $RevInc^{\mathcal{CFID}}_{CL}$, we distinguish four cases for $cfid_T$ and show the property for every case. It is important that these cases are complete, i.e. every $cfid_T \in \mathcal{CFIDI}_T$ belongs exactly to one case.

Case 1: $Card_T(f(cfid_T)) = \{1\}$.
$C_S \models \top \equiv \pi^{-1}([\top]_\equiv) \equiv \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL}(cfid_T)$ follows directly from Definition 6.3.7.

Case 2: $Card_T(f(cfid_T)) = \{0\}$.
Impossible since $\tau_{CL(C_S)}(cfid_T) = \bot$ by Definition 6.1.8. This is a direct contradiction to the assumption that $\tau_{CL(C_S)}(cfid_T) = \top$.

Case 3: $Card_T(f(cfid_T)) = \{0, 1\}$.
Since $f(cfid_T)$ is optional, it holds that $RevInc^{\mathcal{CFID}}_{CL}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0))$. The resulting formula evaluates to $\top$ if and only if all formulae of $Rev_{CL}(cfid_T, 0)$ evaluate to $\bot$ (case 3.1) and (at least) one formula of $Rev_{CL}(cfid_T, 1)$ evaluates to $\top$ (case 3.2). In the following, we will show that both properties hold.

Subcase 3.1: *To show:* $\forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$.
To prove this statement, we will show the contraposition $\exists \varphi \in Rev_{CL}(cfid_T, 0) : C_S \models \varphi \implies \tau_{CL(C_S)}(cfid_T) = \bot$. Assume that $\exists \varphi \in Rev_{CL}(cfid_T, 0) : C_S \models \varphi$. By definition of Rev_{CL} (see Definition 6.3.3), the existence of a configuration decision $cd = (\varphi, Steps) \in CL$ with $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{0\}$ and $C_S \models \varphi$ follows.

The set $\mathcal{CS}_{CL}(cd, cfid_T)$ contains all configuration steps of cd for $cfid_T$ (see Definition 6.3.1). Since $cd \in CL$ and $C_S \models \varphi$ hold, we can apply Lemma 6.3.2 and obtain the fact that $\mathcal{CS}_{CL}(cd, cfid_T)$ is a subset of the set of applied configuration steps $\mathcal{CS}_{CL}(C_S, cfid_T)$ (see Definition 6.2.6). The fact $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{0\}$ allows the application of Lemma 6.2.3 to $\mathcal{CS}_{CL}(cd, cfid_T)$ and $\mathcal{CS}_{CL}(C_S, cfid_T)$ and it follows that $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{0\}$. Hence $Cfg_{CL}(C_S, cfid_T) = (\{0\}, i, t)$ (see Definition 6.2.7), which means that $cfid_T$ is excluded in the target configuration $\tau_{CL(C_S)}(cfid_T) = \bot$. By contraposition, it follows that $\nexists \varphi \in Rev_{CL}(cfid_T, 0) : C_S \models \varphi$ holds under the assumption, which is equivalent to $\forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$.

Subcase 3.2: *To show:* $\exists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$.
The assumption $\tau_{CL(C_S)}(cfid_T) = \top$ implies that $Card_{CL(C_S)}(cfid_T) = \{1\}$, by Definition 6.1.8. This means that $Card(Cfg_{CL}(C_S, cfid_T)) = \{1\}$ and, moreover, $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{1\}$ (see Definition 6.2.7). Because of Lemma 6.2.3, we know that every configuration step of the set $\mathcal{CS}_{CL}(C_S, cfid_T)$ includes the cardinality 1 (i.e. $\forall cs' \in \mathcal{CS}_{CL}(C_S, cfid_T) : 1 \in Card_{cs}(cs')$) and there exists a configuration step cs in this set which sets $cfid_T$ to inclusion (i.e. $\exists cs \in \mathcal{CS}_{CL}(C_S, cfid_T) : Card_{cs}(cs) = \{1\}$). By Definition 6.2.6, we know that cs is in the set of configuration steps $Steps$ of a configuration decision $cd = (\vartheta, Steps)$ of CL and $C_S \models \vartheta$. We construct the set of configuration steps of this configuration decision cd for $cfid_T$ (see Definition 6.3.1) $\mathcal{CS}_{CL}(cd, cfid_T)$. Obviously, the configuration step cs is an element of this set. Since $C_S \models \vartheta$ and $cd = (\vartheta, Steps) \in CL$ hold by construction, we can apply Lemma 6.3.2 and obtain that $\mathcal{CS}_{CL}(cd, cfid_T)$ is a subset of $\mathcal{CS}_{CL}(C_S, cfid_T)$. It follows directly that $\forall cs' \in \mathcal{CS}_{CL}(cd, cfid_T) : 1 \in Card_{cs}(cs')$. Remember that $cs \in \mathcal{CS}_{CL}(cd, cfid_T)$. The application of Lemma 6.2.3 leads to $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{1\}$. This and the fact that $cd \in CL$ lead to $\vartheta \in Rev_{CL}(cfid_T, 1)$, i.e. $\exists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$ holds under the assumption.

The proven properties $\forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$ of case 3.1 and $\exists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$ of case 3.2 lead directly to $C_S \models \pi^{-1}(\sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0)))$ and, since $RevInc_{CL}^{\mathcal{CFID}}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0))$ by definition, $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ holds for optional features (case 3) under the assumption that $\tau_{CL(C_S)}(cfid_T) = \top$.

Case 4: $isCloned_T(f(cfid_T))$.
In this case, the reverse inclusion mapping is given by $RevInc_{CL}^{\mathcal{CFID}}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0)) \wedge \sqcup(Rev'_{CL}(cfid_T))$. The resulting formula evaluates to $\top$ if and only if all formulae of $Rev_{CL}(cfid_T, 0)$ evaluate to $\bot$ (case 4.1), a formula of $Rev_{CL}(cfid_T, 1)$ evaluates to $\top$ (case 4.2) and a formula of

$Rev'_{CL}(cfid_T)$ evaluates to $\top$ (case 4.3). In the following, we will show that these properties hold.

Subcase 4.1: *To show:* $\forall\varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$.
Analogous to case 3.1. It follows that $\forall\varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$ holds under the assumption.

Subcase 4.2: *To show:* $\exists\varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$.
Analogous to case 3.2. It follows that $\exists\varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$ holds under the assumption.

Subcase 4.3: *To show:* $\exists\varphi \in Rev'_{CL}(cfid_T) : C_S \models \varphi$.
Because of the assumption $\tau_{CL(C_S)}(cfid_T) = \top$, it follows from Definition 6.1.8 that $i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$. This means that the instance of $cfid_T$ is created by the configuration $CL(C_S)$, i.e. $i(cfid_T) \in Inst(Cfg_{CL}(C_S, NoInst(cfid_T)))$ (see Definition 6.2.7). Furthermore, Definition 6.2.7 states that the predicate $i(cfid_T) \in Inst_{cs}(\sum_{cs\in\mathcal{CS}_{CL}(C_S,NoInst(cfid_T))} cs)$ holds. Note that $+_{\mathcal{CS}}$ is defined as union with respect to the instance sets of the configuration steps. Hence, a configuration step $cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))$ with $i(cfid_T) \in Inst_{cs}(cs)$ exists. By Definition 6.2.6, we know that this configuration step is contained in the set of configuration steps *Steps* of a configuration decision $cd = (\vartheta, Steps) \in CL$ and $C_S \models \vartheta$. Next, we construct the set of configuration steps for $NoInst(cfid_T)$ of configuration decision cd (see Definition 6.3.1) $\mathcal{CS}_{CL}(cd, NoInst(cfid_T))$. Obviously, cs is contained in this set. $i(cfid_T) \in Inst_{cs}(\sum_{cs\in\mathcal{CS}_{CL}(cd,NoInst(cfid_T))} cs)$ follows directly (because $cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))$ and $+_{\mathcal{CS}}$ is defined as union with respect to the instance sets). This and the fact that $cd \in CL$ imply that $\vartheta \in Rev'_{CL}(cfid_T)$, i.e. $\exists\varphi \in Rev'_{CL}(cfid_T) : C_S \models \varphi$ holds under the assumption.

The proven properties $\forall\varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$ of case 4.1, $\exists\varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$ of case 4.2 and $\exists\varphi \in Rev'_{CL}(cfid_T) : C_S \models \varphi$ of case 4.3 lead to $C_S \models \pi^{-1}(\sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg\sqcup(Rev_{CL}(cfid_T, 0)) \wedge \sqcup(Rev'_{CL}(cfid_T)))$ and, $RevInc^{\mathcal{CFID}}_{CL}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0)) \wedge \sqcup(Rev'_{CL}(cfid_T))$ (which holds by Definition 6.3.7) implies that $C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL}(cfid_T)$ holds for cloned features (case 4) under the assumption that $\tau_{CL(C_S)}(cfid_T) = \top$.

We showed the property $C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL}(cfid_T) \Longleftarrow \tau_{CL(C_S)}(cfid_T) = \top$ for mandatory features (case 1), abstract features (case 2), optional features (case 3) and instances of cloned features (case 4). It follows that this property holds for every $cfid_T \in \mathcal{CFIDI}_T$.

If-part ("$\Longrightarrow$"). Let $C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL}(cfid_T)$ hold.
We have to show that $\tau_{CL(C_S)}(cfid_T) = \top$. According to the definition of $RevInc^{\mathcal{CFID}}_{CL}$, we distinguish four cases for $cfid_T$ and show the property for every case. It is im-

portant that these cases are complete, i.e. every $cfid_T \in \mathcal{CFIDI}_T$ belongs exactly to one case.

Case 1: $Card_T(f(cfid_T)) = \{1\}$.
$\tau_{CL(C_S)}(cfid_T) = \top$ follows directly from Definition 6.1.8.

Case 2: $Card_T(f(cfid_T)) = \{0\}$.
Impossible since $C_S \not\models \bot \equiv \pi^{-1}([\bot]_{\equiv}) \equiv \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ by Definition 6.3.7. This is a direct contradiction to the assumption that $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

Case 3: $Card_T(f(cfid_T)) = \{0,1\}$.
In this case, we will show the contraposition $\tau_{CL(C_S)}(cfid_T) \neq \top \implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$. Assume that $\tau_{CL(C_S)}(cfid_T) \neq \top$. Then we can distinguish two cases, which we will consider separately: $\tau_{CL(C_S)}(cfid_T) = \bot$ and $\tau_{CL(C_S)}(cfid_T)$ *is not defined*. In the following, we will show that both cases lead to $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$. Note that $RevInc_{CL}^{\mathcal{CFID}}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0))$ since $f(cfid_T)$ is optional, i.e. $\exists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$ and $\forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$.

Subcase 3.1: $\tau_{CL(C_S)}(cfid_T) = \bot$.
This implies that $Card_{CL(C_S)}(cfid_T) = \{0\}$ by Definition 6.1.8. Now we take the configuration link CL into account. The application of Definition 6.2.7 leads to the facts $Card(Cfg_{CL}(C_S, cfid_T)) = \{0\}$ and $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{0\}$. The application of Lemma 6.2.3 states the existence of a configuration step $cs \in \mathcal{CS}_{CL}(C_S, cfid_T)$ with $Card_{cs}(cs) = \{0\}$. Since $\mathcal{CS}_{CL}(C_S, cfid_T)$ is the set of applied configuration steps for $cfid_T$ (see Definition 6.2.6) and $cs \in \mathcal{CS}_{CL}(C_S, cfid_T)$, we know that a configuration decision $cd = (\varphi, Steps) \in CL$ with $C_S \models \varphi$ and $cs \in Steps$ exists. Now we construct the set $\mathcal{CS}_{CL}(cd, cfid_T)$ of configuration steps of cd for $cfid_T$ according to Definition 6.3.1. Obviously, $cs \in \mathcal{CS}_{CL}(cd, cfid_T)$. Through the application of Lemma 6.2.2, we obtain that $Card_{cs}(\mathcal{CS}_{CL}(cd, cfid_T)) = \{0\}$. This and the fact that $cd \in CL$ lead to $\varphi \in Rev_{CL}(cfid_T, 0)$ (see Definition 6.3.3). Since $C_S \models \varphi \in Rev_{CL}(cfid_T, 0)$, it holds that $\neg \forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$, i.e. $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

Subcase 3.2: $\tau_{CL(C_S)}(cfid_T)$ *is not defined.*
This implies that $Card_{CL(C_S)}(cfid_T) = \{0,1\}$ or $Card_{CL(C_S)}(cfid_T)$ *is not defined*. We distinguish these cases and we will show that $\forall cs \in \mathcal{CS}_{CL}(C_S, cfid_T) : Card_{cs}(cs) = \{0,1\}$ holds in both cases. Case A: let $Card_{CL(C_S)}(cfid_T) = \{0,1\}$ hold. We know that $Card(Cfg_{CL}(C_S, cfid_T)) = \{0,1\}$. Consequently, the statement $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{0,1\}$ holds by Definition 6.2.7. The application of Lemma 6.2.3 leads to $\forall cs \in \mathcal{CS}_{CL}(C_S, cfid_T) : Card_{cs}(cs) = \{0,1\}$. Case B: let $Card_{CL(C_S)}(cfid_T)$ *is not defined* hold. We know that $Cfg_{CL}(C_S, cfid_T)$ *is not defined* and therefore $\mathcal{CS}_{CL}(C_S, cfid_T) = \emptyset$ by Definition 6.2.7. Note that an all-quantification over the empty set always evaluates to $\top$. We showed that $\forall cs \in \mathcal{CS}_{CL}(C_S, cfid_T) : Card_{cs}(cs) = \{0,1\}$ holds in both cases A and B, i.e.

it follows from the assumption $\tau_{CL(C_S)}(cfid_T)$ *is not defined*. Let $cd = (\varphi, Steps)$ be an arbitrary configuration decision of CL with $C_S \models \varphi$. Then the set of configuration steps of cd for $cfid_T$ (see Definition 6.3.1) is, by Lemma 6.3.2, a subset of the set of applied configuration steps for $cfid_T$ (see Definition 6.2.6), i.e. $\mathcal{CS}_{CL}(cd, cfid_T) \subseteq \mathcal{CS}_{CL}(C_S, cfid_T)$. Because of the subset relation, $\forall cs \in \mathcal{CS}_{CL}(cd, cfid_T) : Card_{cs}(cs) = \{0, 1\}$ holds. By applying Lemma 6.2.3, it follows that $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{0, 1\}$. Note that this property holds for all configuration decisions $cd = (\varphi, Steps) \in CL$ with $C_S \models \varphi$ (since it was shown for an arbitrary configuration decision cd). This implies that $\nexists cd = (\varphi, Steps) \in CL : C_S \models \varphi \wedge Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{1\}$. Now we apply Definition 6.3.3 to this fact and obtain the fact that $\nexists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$, i.e. $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

The proven properties $\tau_{CL(C_S)}(cfid_T) = \bot \implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ of case 3.1 and $\tau_{CL(C_S)}(cfid_T)$ *is not defined* $\implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ of case 3.2 lead, by contraposition, directly to the fact that $\tau_{CL(C_S)}(cfid_T) = \top$ holds for optional features (case 3) under the assumption $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

Case 4: $isCloned_T(f(cfid_T))$.
In this case, we will show the contraposition $\tau_{CL(C_S)}(cfid_T) \neq \top \implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$. Assume that $\tau_{CL(C_S)}(cfid_T) \neq \top$. Then we can distinguish two cases, which we will consider separately: $\tau_{CL(C_S)}(cfid_T) = \bot$ and $\tau_{CL(C_S)}(cfid_T)$ *is not defined*. In the following, we will show that both cases lead to $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$. Note that $RevInc_{CL}^{\mathcal{CFID}}(cfid_T) = \sqcup(Rev_{CL}(cfid_T, 1)) \wedge \neg \sqcup (Rev_{CL}(cfid_T, 0)) \wedge \sqcup(Rev'_{CL}(cfid_T))$ since $f(cfid_T)$ is cloned. This holds if and only if the following three predicates hold: $\exists \varphi \in Rev_{CL}(cfid_T, 1) : C_S \models \varphi$, $\forall \varphi \in Rev_{CL}(cfid_T, 0) : C_S \not\models \varphi$ and $\exists \varphi \in Rev'_{CL}(cfid_T) : C_S \models \varphi$.

Subcase 4.1: $\tau_{CL(C_S)}(cfid_T) = \bot$.
An instance of a cloned feature is excluded if its cardinality is set to zero $Card_{CL(C_S)} = \{0\}$ or if the instance is not created $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$ (see Definition 6.1.8). The former case is equivalent to the corresponding case for optional features (case 3.1). Therefore, we only have to show the last-mentioned case. Let $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$ hold. This implies, by Definition 6.2.7, that $i(cfid_T) \notin Inst(Cfg_{CL}(C_S, NoInst(cfid_T)))$. By the definition of Cfg_{CL} (Definition 6.2.7), $i(cfid_T) \notin Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))} cs)$ holds. Since $+_{\mathcal{CS}}$ is defined as union with respect to the instance sets (see Definition 6.2.2), it follows that $\nexists cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T)) : i(cfid_T) \in Inst_{cs}(cs)$. Let $cd = (\varphi, Steps)$ be an arbitrary configuration decision of CL with $C_S \models \varphi$. Then the set of configuration steps of cd for $NoInst(cfid_T)$ (see Definition 6.3.1) is, by Lemma 6.3.2, a subset of the set of applied configuration steps for $NoInst(cfid_T)$ (see Definition 6.2.6), i.e. $\mathcal{CS}_{CL}(cd, NoInst(cfid_T)) \subseteq \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))$. Because of the subset relation, $\nexists cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T)) : i(cfid_T) \in Inst_{cs}(cs)$. This and the fact that $+_{\mathcal{CS}}$ is defined as union with respect to the instance sets lead to $i(cfid_T) \notin Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))} cs)$. Note that this property holds for all configuration decisions $cd = (\varphi, Steps) \in CL$ with $C_S \models \varphi$ (since it was shown

for an arbitrary cd). Now we apply Definition 6.3.4 to this fact and obtain that $\nexists \varphi \in Rev'_{CL}(cfid_T) : C_S \models \varphi$, i.e. $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

Subcase 4.2: *$\tau_{CL(C_S)}(cfid_T)$ is not defined.*
Analogous to case 3.2. It follows that $C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

The proven properties $\tau_{CL(C_S)}(cfid_T) = \bot \implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ of case 4.1 and $\tau_{CL(C_S)}(cfid_T)$ *is not defined* $\implies C_S \not\models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$ of case 4.2 lead, by contraposition, directly to the fact that $\tau_{CL(C_S)}(cfid_T) = \top$ holds for cloned features (case 4) under the assumption $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T)$.

We showed the property $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T) \implies \tau_{CL(C_S)}(cfid_T) = \top$ for mandatory features (case 1), abstract features (case 2), optional features (case 3) and instances of cloned features (case 4). It follows that this property holds for every $cfid_T \in \mathcal{CFIDI}_T$.

Since the statement is valid both for the only-if- and the if-part, the equivalence $C_S \models \pi^{-1} \circ RevInc_{CL}^{\mathcal{CFID}}(cfid_T) \iff \tau_{CL(C_S)}(cfid_T) = \top$ is proven. □

In the next definition, we will formally introduce the reverse selection mapping of configured feature identifiers that takes the tree-structure of the target model into account.

Definition 6.3.9 (Reverse Selection Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$. Then the reverse selection mapping of configured feature identifiers with respect to this configuration link is given by function $RevSel_{CL_{S\to T}}^{\mathcal{CFID}} : \mathcal{CFIDI}_T \to Form(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$RevSel_{CL_{S\to T}}^{\mathcal{CFID}}(cfid_T) = \begin{cases} RevInc_{CL_{S\to T}}^{\mathcal{CFID}}(cfid_T) & \text{if } isRoot_T(f(cfid_T)) \\ \left(\begin{array}{c} RevInc_{CL_{S\to T}}^{\mathcal{CFID}}(cfid_T) \\ \wedge RevSel_{CL_{S\to T}}^{\mathcal{CFID}}(Parent(cfid_T)) \end{array}\right) & else \end{cases}$$

This recursive definition is quite intuitive: a configured feature identifier that points to a root feature is selected if it is included and a configured feature identifier that points to a non-root feature is selected if it is included itself and its parent is selected. Note that this definition uses the parent function on instance level and not a parent relation on feature level since we are interested in configured feature identifiers and not in features at this point.

The following lemma shows the correctness and minimality of this mapping.

Lemma 6.3.10 (Correctness and Minimality of the Reverse Selection Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows Form(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ RevSel_{CL_{S\to T}}^{\mathcal{CFID}}(cfid_T) \iff CL_{S\to T}(C_S) \vdash cfid_T$$

Proof. According to Definition 6.1.9, we have to show the following statement.

$$C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff \sigma_{CL_{S\to T}(C_S)}(cfid_T) = \top$$

Proof by mathematical induction.

Basis step. Let $isRoot_T(f(cfid_T))$ hold.

$$\begin{aligned}
&\sigma_{CL_{S\to T}(C_S)}(cfid_T) = \top \\
&\iff \tau_{CL_{S\to T}(C_S)}(cfid_T) = \top && \text{by Definition 6.1.9} \\
&\iff C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Lemma 6.3.8} \\
&\iff C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Definition 6.3.9}
\end{aligned}$$

Inductive step. Let

$$C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \iff \sigma_{CL_{S\to T}(C_S)}(Parent(cfid_T)) = \top$$

hold for the parent of $cfid_T$ (inductive hypothesis).

$$\sigma_{CL_{S\to T}(C_S)}(cfid_T) = \top$$
$$\overset{\text{Definition 6.1.9}}{\iff}$$
$$\tau_{CL_{S\to T}(C_S)}(cfid_T) = \top \wedge \sigma_{CL_{S\to T}(C_S)}(Parent(cfid_T)) = \top$$
$$\overset{\text{inductive hypothesis}}{\iff}$$
$$\tau_{CL_{S\to T}(C_S)}(cfid_T) = \top \wedge C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T))$$
$$\overset{\text{Lemma 6.3.8}}{\iff}$$
$$\begin{aligned}
&C_S \models \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\
\wedge &C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T))
\end{aligned}$$
$$\overset{\text{logic}}{\iff}$$
$$C_S \models \begin{pmatrix} \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \wedge \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \end{pmatrix}$$
$$\overset{\text{Definition 6.3.9}}{\iff}$$
$$C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T)$$

□

According to Section 5.1.2, we now define the reverse selection mapping of features. It is required for the transformation of feature links because they are, in contrast to feature constraints, formulated on feature level and always evaluated with respect to selection.

Definition 6.3.11 (Reverse Selection Mapping of Features w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$. Then the reverse selection mapping of features with respect to this configuration link is given by function $RevSel^{F}_{CL_{S\to T}} : F_T \to Form(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$RevSel^{F}_{CL_{S\to T}}(f_T) = \sqcup\left(\left\{\varphi \in Form(\mathcal{CFIDI}_S) \;\middle|\; \begin{array}{c} cfid_T \in \mathcal{CFIDI}_T \\ \wedge f(cfid_T) = f_T \\ \wedge \varphi \in RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \end{array}\right\}\right)$$

In this definition we consider all configured feature identifiers pointing to the given feature (if this feature is cloned, we take only the configured feature identifiers pointing to an instance of the cloned feature into account), map them by the reverse selection mapping of configured feature identifiers to sets of logical formulae and collect all these formulae in a set. Subsequently, we combine all formulae of the resulting set by disjunction according to Definition 6.3.5. The result is a set of equivalent formulae whose satisfaction is sufficient for the selection of a configured feature identifier pointing to the given feature. Note that this mapping delivers formulae over configured feature identifiers and not over features (of the source feature model), even though its domain is the set of features and not the set of configured feature identifiers (of the target feature model).

Correctness and minimality of this mapping are proven in the following.

Lemma 6.3.12 (Correctness and Minimality of the Reverse Selection Mapping of Features)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$ and a feature $f_T \in F_T$ of feature model T. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows Form(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ RevSel^{F}_{CL_{S\to T}}(f_T)$$
$$\Longleftrightarrow$$
$$\exists cfid_T \in \mathcal{CFIDI}_T : f(cfid_T) = f_T \wedge CL_{S\to T}(C_S) \vdash cfid_T$$

Proof. For the proof of this property we define a set $M(f_T)$ according to the set in Definition 6.3.11.

$$M(f_T) = \left\{\varphi \in Form(\mathcal{CFIDI}_S) \;\middle|\; \begin{array}{c} cfid_T \in \mathcal{CFIDI}_T \\ \wedge f(cfid_T) = f_T \\ \wedge \varphi \in RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \end{array}\right\}$$

$$C_S \models \pi^{-1} \circ RevSel^{F}_{CL_{S\to T}}(f_T)$$
$$\overset{\text{Definition 6.3.11, construction of set } M(f_T) \text{ and } \pi \text{ is a projection}}{\Longleftrightarrow}$$
$$\exists \varphi_T \in M(f_T) : C_S \models \varphi_T$$
$$\overset{\text{Definition of set } M(f_T)}{\Longleftrightarrow}$$
$$\exists \varphi_T \in Form(\mathcal{CFIDI}_S), \exists cfid_T \in \mathcal{CFIDI}_T :$$
$$f(cfid_T) = f_T \wedge \varphi_T \in RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \wedge C_S \models \varphi_T$$
$$\overset{\pi \text{ is a projection}}{\Longleftrightarrow}$$
$$\exists cfid_T \in \mathcal{CFIDI}_T : f(cfid_T) = f_T \wedge C_S \models \pi^{-1} \circ RevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T)$$
$$\overset{\text{Lemma 6.3.10}}{\Longleftrightarrow}$$
$$\exists cfid_T \in \mathcal{CFIDI}_T : f(cfid_T) = f_T \wedge CL_{S\to T}(C_S) \vdash cfid_T$$

□

Equipped with the three introduced reverse mappings, we are finally ready to define the transformation of propositional logic formulae (cf. Section 5.1.3). Actually, there are two transformations: one for formulae over the configured feature identifiers (for the propagation of feature constraints) and one for formulae over the features (for the propagation of feature links). In this context, we see feature links as propositional logic formulae over the features according to Definition 6.1.12. The transformations are defined as functions mapping a target-side feature constraint or feature link to a source-side feature constraint by using the introduced reverse mappings. Through this definition, we can propagate every constraint on instance and feature level that can be expressed by propositional logic. This especially means that the propagation of feature links is not limited to the three established feature link types `needs`, `excludes` and `alternative` but can be used for every imaginable feature link type that can be expressed by propositional logic. Note that the result of a transformation is a concrete formula and not a set of equivalent formulae as in the case of the reverse mappings. We use an arbitrary bijective projection π to identify a concrete formula of a set of equivalent formulae. At this point, it does not matter which of the formulae π chooses since they are logically equivalent.

Definition 6.3.13 (Transformation of Propositional Logic Formulae)**.** Given a complete system of representatives $R \subseteq Form(\mathcal{CFIDI}_S)$ of $\equiv$, the bijective projection $\pi : R \rightleftarrows Form(\mathcal{CFIDI}_S)_{/\equiv}$ defined by $\pi(\varphi) = [\varphi]_\equiv$, a configuration link $CL_{S\to T}$ and a propositional logic formula $\varphi_T \in Form(\mathcal{CFIDI}_T)$ (resp. $\varphi_T \in Form(F_T)$). Then the transformation of this formula with respect to the configuration link is recursively defined as $Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi} : Form(\mathcal{CFIDI}_T) \to Form(\mathcal{CFIDI}_S)$ (resp.

$Trafo^{F}_{CL_{S\to T},\pi} : Form(F_T) \to Form(\mathcal{CFIDI}_S))$ with

$$Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T) = \begin{cases} \pi^{-1} \circ RevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } \varphi_T = cfid_T \in \mathcal{CFIDI}_T \\ \neg\, Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi'_T) & \text{if } \varphi_T = \neg\varphi'_T \text{ and } \varphi'_T \in Form(\mathcal{CFIDI}_T) \\ \left(\begin{array}{c} Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi'_T) \\ \otimes\, Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi''_T) \end{array}\right) & \text{if } \varphi_T = \varphi'_T \otimes \varphi''_T \text{ with } \otimes \in \{\wedge, \vee, \to, \leftrightarrow\} \text{ and } \varphi'_T, \varphi''_T \in Form(\mathcal{CFIDI}_T) \end{cases}$$

$$Trafo^{F}_{CL_{S\to T},\pi}(\varphi_T) = \begin{cases} \pi^{-1} \circ RevSel^{F}_{CL_{S\to T}}(f_T) & \text{if } \varphi_T = f_T \in F_T \\ \neg\, Trafo^{F}_{CL_{S\to T},\pi}(\varphi'_T) & \text{if } \varphi_T = \neg\varphi'_T \text{ and } \varphi'_T \in Form(F_T) \\ \left(\begin{array}{c} Trafo^{F}_{CL_{S\to T},\pi}(\varphi'_T) \\ \otimes\, Trafo^{F}_{CL_{S\to T},\pi}(\varphi''_T) \end{array}\right) & \text{if } \varphi_T = \varphi'_T \otimes \varphi''_T \text{ with } \otimes \in \{\wedge, \vee, \to, \leftrightarrow\} \text{ and } \varphi'_T, \varphi''_T \in Form(F_T) \end{cases}$$

This recursive definition is straightforward: applied to a formula, the transformation substitutes all variables (i.e. configured feature identifiers or features) by their reverse mappings. In the case of feature constraints, we use the reverse inclusion mapping of configured feature identifiers since feature constraints are evaluated with respect to inclusion and formulated on instance level. In the case of feature links, we use the reverse selection mapping of features since they are evaluated with respect to selection and formulated on feature level. In both cases we apply the inverse projection π^{-1} to the mapping's result in order to derive a concrete formula. Note that the introduction of the transformation in Section 5.1.3 additionally considers feature constraints with selection statements and substitutes them by the corresponding reverse selection mapping of configured feature identifiers. In the formalization, there is no abbreviation for the selection and this case can therefore be neglected. A selection statement in a feature constraint can simply be replaced by inclusion statements and the reverse inclusion mappings can be used subsequently.

Now we present the proofs for correctness and minimality of the defined transformations. In these proofs, we use the already proven properties of the mappings.

Theorem 6.3.14 (Correctness and Minimality of the Transformation of Feature Constraints)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$ and a feature constraint $\varphi_T \in Form(\mathcal{CFIDI}_T)$. Then the following holds for every bijective projection $\pi : R \rightleftarrows Form(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models Trafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T) \iff CL_{S\to T}(C_S) \models \varphi_T$$

Proof. Let C_S be an arbitrary configuration of S, $\varphi_T \in \mathit{Form}(\mathcal{CFIDI}_T)$ be an arbitrary constraint and π be the bijective projection for an arbitrary complete system of representatives R. First of all, we apply Definition 6.1.11 to both sides and obtain

$$B^{\mathcal{CFID}^*}_{C_S}(\mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T)) = B^{\mathcal{CFID}^*}_{CL_{S\to T}(C_S)}(\varphi_T).$$

In the next step we apply the coincidence lemma (see Lemma A.2.1 on page 237). As preparation for this step we consider the set of logical formulae $\mathit{Form}(\mathcal{CFIDI}_S \uplus \mathcal{CFIDI}_T)$ and define two functions $\sigma_1, \sigma_2 : \mathcal{CFIDI}_S \uplus \mathcal{CFIDI}_T \to \mathit{Form}(\mathcal{CFIDI}_S \uplus \mathcal{CFIDI}_T)$ with

$$\sigma_1(x) = \begin{cases} \pi^{-1} \circ \mathit{RevInc}^{\mathcal{CFID}}_{CL_{S\to T}}(x) & \mid \text{if } x \in \mathcal{CFIDI}_T \\ \bot & \mid \text{if } x \in \mathcal{CFIDI}_S \end{cases}$$

$$\sigma_2(x) = \begin{cases} x & \mid \text{if } x \in \mathcal{CFIDI}_T \\ \bot & \mid \text{if } x \in \mathcal{CFIDI}_S \end{cases}$$

which define the required substitutions $[\sigma_1]$ and $[\sigma_2]$ for the application of the coincidence lemma. The transformation of logical formulae can now be seen as substitution defined by $\mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T) = \varphi_T[\sigma_1]$ for $\varphi_T \in \mathit{Form}(\mathcal{CFIDI}_T)$ (see Definition 6.3.13). To prove the statement above ($B^{\mathcal{CFID}^*}_{C_S}(\mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T)) = B^{\mathcal{CFID}^*}_{CL_{S\to T}(C_S)}(\varphi_T)$) it is sufficient to show the more general statement

$$B^{\mathcal{CFID}^*}_{C_S}(\varphi[\sigma_1]) = B^{\mathcal{CFID}^*}_{CL_{S\to T}(C_S)}(\varphi[\sigma_2])$$

for $\varphi \in \mathit{Form}(\mathcal{CFIDI}_S \uplus \mathcal{CFIDI}_T)$. We apply the coincidence lemma (see Lemma A.2.1 on page 237) and it follows that it is sufficient to show that

$$B^{\mathcal{CFID}^*}_{C_S}(\sigma_1(\mathit{cfid})) = B^{\mathcal{CFID}^*}_{CL_{S\to T}(C_S)}(\sigma_2(\mathit{cfid}))$$

holds for every $\mathit{cfid} \in \mathcal{CFIDI}_S \uplus \mathcal{CFIDI}_T$. This is trivial for $\mathit{cfid}_S \in \mathcal{CFIDI}_S$.

Let $\mathit{cfid}_T \in \mathcal{CFIDI}_T$ be a configured feature identifier of the target feature model. Then we evaluate $\sigma_1(\mathit{cfid}_T) = \pi^{-1} \circ \mathit{RevInc}^{\mathcal{CFID}}_{CL_{S\to T}}(\mathit{cfid}_T)$, $\sigma_2(\mathit{cfid}_T) = \mathit{cfid}_T$ and $B^{\mathcal{CFID}^*}_{CL_{S\to T}(C_S)}(\mathit{cfid}_T) = B^{\mathcal{CFID}}_{CL_{S\to T}(C_S)}(\mathit{cfid}_T)$ (see Definition 6.1.11), which is equivalent to $\tau_{CL_{S\to T}(C_S)}(\mathit{cfid}_T) = \top$. It remains to show that

$$B^{\mathcal{CFID}^*}_{C_S}(\pi^{-1} \circ \mathit{RevInc}^{\mathcal{CFID}}_{CL_{S\to T}}(\mathit{cfid}_T)) \iff \tau_{CL_{S\to T}(C_S)}(\mathit{cfid}_T) = \top,$$

which can be converted into

$$C_S \models \pi^{-1} \circ \mathit{RevInc}^{\mathcal{CFID}}_{CL_{S\to T}}(\mathit{cfid}_T) \iff \tau_{CL_{S\to T}(C_S)}(\mathit{cfid}_T) = \top$$

(according to Definition 6.1.11). This follows directly by applying Lemma 6.3.8. □

Theorem 6.3.15 (Correctness and Minimality of the Transformation of Feature Links)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$ and a constraint $\varphi_T \in \mathit{Form}(F_T)$ over the set of features of T. Then the following holds for every bijective projection $\pi : R \rightleftarrows \mathit{Form}(\mathcal{CFID}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \mathit{Trafo}^{F}_{CL_{S\to T},\pi}(\varphi_T) \iff CL_{S\to T}(C_S) \models \varphi_T$$

Proof. This proof is similar to the proof of Theorem 6.3.14. First of all, we define two functions $\sigma_1, \sigma_2 : \mathcal{CFIDI}_S \uplus F_T \to \mathit{Form}(\mathcal{CFIDI}_S \uplus F_T)$ with

$$\sigma_1(x) = \begin{cases} \pi^{-1} \circ \mathit{RevSel}^F_{CL_{S\to T}}(x) & \mid \text{if } x \in F_T \\ \bot & \mid \text{if } x \in \mathcal{CFIDI}_S \end{cases}$$

$$\sigma_2(x) = \begin{cases} x & \mid \text{if } x \in F_T \\ \bot & \mid \text{if } x \in \mathcal{CFIDI}_S \end{cases}$$

and consider the transformation as substitution $\mathit{Trafo}^F_{CL_{S\to T},\pi}(\varphi_T) = \varphi_T[\sigma_1]$. The application of the coincidence lemma (see Lemma A.2.1 on page 237) leads to the fact that it is sufficient to show the statement

$$B^{\mathcal{CFID}^*}_{C_S}(\sigma_1(x)) = B^{F^*}_{CL_{S\to T}(C_S)}(\sigma_2(x))$$

for $x \in \mathcal{CFIDI}_S \uplus F_T$, which is trivial for $x \in \mathcal{CFIDI}_S$. Let $f_T \in F_T$ be a feature of the target feature model. Then we evaluate σ_1 and σ_2 and obtain by Definition 6.1.11

$$B^{\mathcal{CFID}^*}_{C_S}(\pi^{-1} \circ \mathit{RevSel}^F_{CL_{S\to T}}(f_T))$$
$$\iff$$
$$\exists \mathit{cfid}_T \in \mathcal{CFIDI}_T : f(\mathit{cfid}_T) = f_T \wedge CL_{S\to T}(C_S) \vdash \mathit{cfid}_T$$

for all $f_T \in F_T$, which remains to be shown. This is equivalent to

$$C_S \models \pi^{-1} \circ \mathit{RevSel}^F_{CL_{S\to T}}(f_T)$$
$$\iff$$
$$\exists \mathit{cfid}_T \in \mathcal{CFIDI}_T : f(\mathit{cfid}_T) = f_T \wedge CL_{S\to T}(C_S) \vdash \mathit{cfid}_T$$

(according to Definition 6.1.11), which follows directly by applying Lemma 6.3.12. □

Finally, we can show the correctness and minimality of the transformation of constraint sets, which is quite obvious when we consider the theorems proven above.

Theorem 6.3.16 (Correctness and Minimality of the Transformation of Constraint Sets)**.** *Given two feature models S and T, configuration link $CL_{S\to T}$, configuration $C_S \in \mathcal{C}_S$, a finite set of constraints $\mathit{Constraints}^{\mathcal{CFID}}_T \subseteq \mathit{Form}(\mathcal{CFIDI}_T)$ over the configured feature identifiers of T and a finite set of constraints $\mathit{Constraints}^F_T \subseteq \mathit{Form}(F_T)$ over the features of T. Then the following holds for every bijective projection $\pi : R \rightleftarrows \mathit{Form}(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\mathit{Constraints}^{\mathcal{CFID}}_T) \cup \mathit{Trafo}^F_{CL_{S\to T},\pi}(\mathit{Constraints}^F_T)$$
$$\iff$$
$$CL_{S\to T}(C_S) \models \mathit{Constraints}^{\mathcal{CFID}}_T \cup \mathit{Constraints}^F_T$$

Proof. This property follows from Theorem 6.3.14 and Theorem 6.3.15:

$$
\begin{aligned}
& C_S \models \mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\mathit{Constraints}^{\mathcal{CFID}}_T) \cup \mathit{Trafo}^{F}_{CL_{S\to T},\pi}(\mathit{Constraints}^{F}_T) \\
& \Longleftrightarrow \\
& \forall \varphi^{\mathcal{CFID}}_T \in \mathit{Constraints}^{\mathcal{CFID}}_T : C_S \models \mathit{Trafo}^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi^{\mathcal{CFID}}_T) \\
& \wedge \forall \varphi^{F}_T \in \mathit{Constraints}^{F}_T : C_S \models \mathit{Trafo}^{F}_{CL_{S\to T},\pi}(\varphi^{F}_T) \\
& \overset{\text{Theorem 6.3.14 and Theorem 6.3.15}}{\Longleftrightarrow} \\
& \forall \varphi^{\mathcal{CFID}}_T \in \mathit{Constraints}^{\mathcal{CFID}}_T : CL_{S\to T}(C_S) \models \varphi^{\mathcal{CFID}}_T \\
& \wedge \forall \varphi^{F}_T \in \mathit{Constraints}^{F}_T : CL_{S\to T}(C_S) \models \varphi^{F}_T \\
& \Longleftrightarrow \\
& CL_{S\to T}(C_S) \models \mathit{Constraints}^{\mathcal{CFID}}_T \cup \mathit{Constraints}^{F}_T
\end{aligned}
$$

□

6.4 Formal Definition of Advanced Feature Constraints

This section formally introduces advanced feature constraints, which have already been motivated and introduced in Section 5.2.5. The main difference between the advanced and the basic form of constraints is that advanced feature constraints provide the possibility to differ between exclusion and the unconfigured state of a configured feature identifier. This is only important when dealing with partial configurations. In the case of full configurations, the exclusion of a configured feature identifier is equivalent to the fact that it is not included. In the formalization, we do not use the abbreviations for selection `[+]` and deselection `[-]` since selection and deselection can easily be expressed by combined inclusion or exclusion statements.

In order to distinguish between inclusion, exclusion and the unconfigured state, a three-valued logic (e.g. Kleene logic [Kle52]) seems to be appropriate for the formalization. However, we decided to formalize advanced feature constraints as predicate logic formulae for a special relational signature because of the following reasons.

1. Inclusion, exclusion and the unconfigured state can be seen as predicates over the variables of a constraint. Every of these predicates evaluates to `true` or `false`, according to the state of the feature in a configuration. Consequently, logical connectives have to be defined only for `true` and `false` and do not have to consider a third state, as in three-valued logics.

2. Constraints can always be evaluated to `true` or `false` in the approach of [Rei08]. There is no third state (cf. Section 2.2).

In the following definition we introduce the signature for advanced feature constraints.

Definition 6.4.1 (Relational Signature for Advanced Feature Constraints)**.** The relational signature for advanced feature constraints is given by

$$\begin{aligned}\Sigma_{AFCons} = \ & \textbf{sorts} : \mathit{card} \\ & \textbf{rels} : \quad \mathit{inc} : \langle \mathit{card} \rangle \\ & \qquad\quad \mathit{exc} : \langle \mathit{card} \rangle \\ & \qquad\quad \mathit{unconf} : \langle \mathit{card} \rangle\end{aligned}$$

This signature defines three unary relations on the sort for cardinalities (*card*): one for the inclusion (*inc*), one for the exclusion (*exc*) and one for the unconfigured state (*unconf*). According to this signature, we can now define advanced feature constraints. Let $Form_\Sigma(X)$ be the set of first-order predicate logic formulae without equality (i.e. "=" is no logical symbol) with respect to the logical signature Σ and the family of variables X.

Definition 6.4.2 (Advanced Feature Constraint)**.** Given a feature model FM. An advanced feature constraint φ is a quantifier free predicate logic formula (i.e. a formula that does not contain any quantifiers) without equality with respect to the relational signature Σ_{AFCons} and the set of variables $X_{card} = \mathcal{CFIDI}_{FM}$.
The set $FormA(\mathcal{CFIDI}_{FM})$ of all advanced feature constraints is defined by

$$FormA(\mathcal{CFIDI}_{FM}) = \left\{ \varphi \in Form_{\Sigma_{AFCons}}(X) \;\middle|\; \begin{array}{l} X = (X_{card}) \wedge \\ X_{card} = \mathcal{CFIDI}_{FM} \wedge \\ Bound(\varphi) = \emptyset \end{array} \right\}$$

Now we can formulate feature constraints like $exc(cfid_1) \vee unconf(cfid_2)$ for configured feature identifiers. Formulae like $cfid_1 = cfid_2$ are not valid advanced feature constraints since the equality is no logical symbol in $Form_{\Sigma_{AFCons}}(X)$. For the evaluation of advanced feature constraints we define a structure (for Σ_{AFCons}) in the next step. A structure is, according to [EMGR+01, Definition 19.2.3], an algebra in which all carrier sets are non-empty and all relational symbols are interpreted as relations.

Definition 6.4.3 (Structure for Advanced Feature Constraints)**.** The Σ_{AFCons}–structure for advanced feature models is given by

$$AFCons = (AFCons_{card}, inc_{AFCons}, exc_{AFCons}, unconf_{AFCons})$$

with

$$\begin{aligned} AFCons_{card} &= \mathcal{P}(\{0,1\}) \backslash \emptyset \\ inc_{AFCons} &= \{\{1\}\} \\ exc_{AFCons} &= \{\{0\}\} \\ unconf_{AFCons} &= \{\{0,1\}\} \end{aligned}$$

The carrier set for cardinalities is defined analogously to the cardinality in a configuration (see Definition 6.1.7) in which only inclusion $\{1\}$, exclusion $\{0\}$ and the unconfigured state $\{0,1\}$ are allowed. The definition of the three unary relations – subsets of the allowed cardinalities – is intuitive. With respect to this structure, we can now define the semantics of advanced feature constraints.

Definition 6.4.4 (Semantics of Advanced Feature Constraints)**.** Given a feature model FM with a configuration C. Then the set of variables with respect to FM is given by $X = (X_{card})$ with $X_{card} = \mathcal{CFIDI}_{FM}$ and the assignment of variables $\beta_C = (\beta_{C_{card}})$ with $\beta_{C_{card}} : \mathcal{CFIDI}_{FM} \to AFCons_{card}$ is defined by

$$\beta_{C_{card}}(cfid) = \begin{cases} \{1\} & \text{if } \tau_C(cfid) = \top \\ \{0\} & \text{if } \tau_C(cfid) = \bot \\ \{0,1\} & \text{else} \end{cases}$$

The configuration C satisfies an advanced feature constraint $\varphi \in FormA(\mathcal{CFIDI}_{FM})$, written $C \models \varphi$, if and only if $(AFCons, \beta_C) \models \varphi$, i.e. φ evaluates to $\top$ in the structure $AFCons$ under the assignment β_C. Otherwise C does not satisfy φ, written $C \not\models \varphi$.

Finally, we define feature models, configuration decisions and configuration links with advanced feature constraints.

Definition 6.4.5 (Feature Model / Configuration Decision / Configuration Link with Advanced Feature Constraints)**.** Feature models, configuration decisions and configuration links with advanced feature constraints are defined as their counterparts with basic feature constraints in Definition 6.1.1, Definition 6.2.4 and Definition 6.2.5, in which the set of basic feature constraints $Form(\mathcal{CFIDI})$ is replaced by the set of advanced feature constraints $FormA(\mathcal{CFIDI})$ and the assignment of variables B is replaced by β.

6.5 Formal Definition and Verification of Advanced Feature Constraint Propagation

In this section we formalize the propagation of advanced feature constraints. The structure of this section is analogous to this of Section 6.3: first of all we introduce some auxiliary functions, then we present the reverse mappings and show their correctness and minimality and, subsequently, we define the transformation and show its correctness and minimality.

Analogous to the case of basic feature constraints, we need a reverse cardinality mapping, a reverse instance mapping and the disjunction of formula sets. Since the advanced versions of these three functions are defined analogously to their basic variants, we simply introduce them together in the following definition.

Definition 6.5.1 (Auxiliary Functions for Advanced Feature Constraints)**.** The advanced reverse cardinality mapping

$$ARev_{CL_{S\to T}} : \mathcal{CFID}_T \times \mathbb{N} \to \mathcal{P}_{fin}(FormA(\mathcal{CFIDI}_S)),$$

the advanced reverse instance mapping

$$ARev'_{CL_{S\to T}} : \mathcal{CFID}_T \to \mathcal{P}_{fin}(FormA(\mathcal{CFIDI}_S))$$

and the disjunction of advanced formula sets

$$\sqcup : \mathcal{P}_{fin}(FormA(X)) \rightarrow FormA(X)_{/\equiv}$$

are defined analogously to their counterparts for basic feature constraints $Rev_{CL_{S\rightarrow T}}$ (see Definition 6.3.3), $Rev'_{CL_{S\rightarrow T}}$ (see Definition 6.3.4) and $\sqcup$ (see Definition 6.3.5), in which the set of feature constraints $Form(\mathcal{CFIDI}_S)$ is replaced by the set of advanced feature constraints $FormA(\mathcal{CFIDI}_S)$.

Now we define the advanced reverse inclusion mapping. The idea of this mapping has already been introduced in the case of basic feature constraints (cf. Section 6.3). It is defined analogously to its basic counterpart (see Definition 6.3.7), though it uses the advanced versions of the auxiliary functions (see Definition 6.5.1). Subsequently, we show its correctness and minimality.

Definition 6.5.2 (Advanced Reverse Inclusion Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\rightarrow T}$ with advanced feature constraints. Then the advanced reverse inclusion mapping of configured feature identifiers with respect to this configuration link is given by function $ARevInc^{\mathcal{CFID}}_{CL_{S\rightarrow T}} : \mathcal{CFIDI}_T \rightarrow FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevInc^{\mathcal{CFID}}_{CL_{S\rightarrow T}}(cfid_T) = \begin{cases} [\top]_\equiv & \text{if } Card_T(f(cfid))=\{1\} \\ [\bot]_\equiv & \text{if } Card_T(f(cfid))=\{0\} \\ \left(\begin{array}{c} \sqcup\,(ARev_{CL_{S\rightarrow T}}(cfid_T,1)) \\ \wedge\neg\sqcup\,(ARev_{CL_{S\rightarrow T}}(cfid_T,0)) \end{array}\right) & \text{if } Card_T(f(cfid))=\{0,1\} \\ \left(\begin{array}{c} \sqcup\,(ARev_{CL_{S\rightarrow T}}(cfid_T,1)) \\ \wedge\neg\sqcup\,(ARev_{CL_{S\rightarrow T}}(cfid_T,0)) \\ \wedge\sqcup\,(ARev'_{CL_{S\rightarrow T}}(cfid_T)) \end{array}\right) & \text{if } isCloned_T(f(cfid)) \end{cases}$$

Lemma 6.5.3 (Correctness and Minimality of the Advanced Reverse Inclusion Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\rightarrow T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ ARevInc^{\mathcal{CFID}}_{CL_{S\rightarrow T}}(cfid_T) \Longleftrightarrow \tau_{CL_{S\rightarrow T}(C_S)}(cfid_T) = \top$$

Proof. The proof of this property is analogous to the corresponding proof for basic feature constraints (see proof of Lemma 6.3.8). However, the evaluation function $B^{\mathcal{CFID}^*}_{C_S}(x)$ (see Definition 6.1.11) has to be replaced by the evaluation function for advanced feature constraints $(AFCons, \beta_{C_S}) \models x$ (see Definition 6.4.4) and all reverse mappings $Rev*$ have to be replaced by their advanced versions $ARev*$. □

We have already figured out in Section 5.2.5 that we need some additional reverse mappings when dealing with advanced feature constraints than in the case of basic

feature constraints. We start with the advanced reverse exclusion mapping. This is a function mapping a configured feature identifier of the target feature model to a set of equivalent formulae over the configured feature identifiers of the source feature model. The application of the underlying configuration link to an arbitrary source configuration fulfilling the constraints of this set leads to a target configuration in which the given configured feature identifier is excluded.

Definition 6.5.4 (Advanced Reverse Exclusion Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ with advanced feature constraints. Then the advanced reverse exclusion mapping of configured feature identifiers with respect to this configuration link is given by function $ARevExc^{\mathcal{CFID}}_{CL_{S\to T}} : \mathcal{CFIDI}_T \to FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) = \begin{cases} [\bot]_\equiv & \text{if } Card_T(f(cfid))=\{1\} \\ [\top]_\equiv & \text{if } Card_T(f(cfid))=\{0\} \\ \sqcup(ARev_{CL_{S\to T}}(cfid_T, 0)) & \text{if } Card_T(f(cfid))=\{0,1\} \\ \begin{pmatrix} \sqcup\,(ARev_{CL_{S\to T}}(cfid_T, 0)) \\ \vee\neg\sqcup\,(ARev'_{CL_{S\to T}}(cfid_T)) \end{pmatrix} & \text{if } isCloned_T(f(cfid)) \end{cases}$$

Configured feature identifiers pointing to mandatory features are never excluded, as opposed to those pointing to abstract features – they are always excluded. Configured feature identifiers pointing to optional features are excluded if at least one criterion of a configuration decision that excludes the configured feature identifier (i.e. sets its cardinality to zero) is fulfilled. This condition is sufficient for the exclusion since an exclude is prioritized over an include when combining configuration steps (see Definition 6.2.2). Instances of cloned features are excluded in the same case or if no criterion of a configuration step that creates the instance is fulfilled (see Definition 6.1.8).

Now we prove the correctness and minimality of this mapping.

Lemma 6.5.5 (Correctness and Minimality of the Advanced Reverse Exclusion Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff \tau_{CL_{S\to T}(C_S)}(cfid_T) = \bot$$

Proof. Note that we omit the index $S \to T$ of $CL_{S\to T}$. We prove this property in two steps. At first we prove the only-if-part and then the if-part.

Only-if-part ("$\Longleftarrow$"). Let $\tau_{CL(C_S)}(cfid_T) = \bot$ hold.
We have to show that $C_S \models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL}(cfid_T)$. According to the definition of $ARevExc^{\mathcal{CFID}}_{CL}$, we distinguish four cases for $cfid_T$ and show the property for every

case. It is important that these cases are complete, i.e. every $cfid_T \in \mathcal{CFIDI}_T$ belongs exactly to one case.

Case 1: $Card_T(f(cfid_T)) = \{1\}$.
Impossible since $\tau_{CL(C_S)}(cfid_T) = \top$ by Definition 6.1.8. This is a direct contradiction to the assumption that $\tau_{CL(C_S)}(cfid_T) = \bot$.

Case 2: $Card_T(f(cfid_T)) = \{0\}$.
$C_S \models \top \equiv \pi^{-1}([\top]_{\equiv}) \equiv \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ follows directly from Definition 6.5.4.

Case 3: $Card_T(f(cfid_T)) = \{0, 1\}$.
Since $\tau_{CL(C_S)}(cfid_T) = \bot$, it follows that $Card_{CL(C_S)}(cfid_T) = \{0\}$ (see Definition 6.1.8). This implies that $Card(Cfg_{CL}(C_S, cfid_T)) = \{0\}$ and, furthermore, $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{0\}$ (see Definition 6.2.7). The application of Lemma 6.2.3 leads to the existence of a configuration step $cs \in \mathcal{CS}_{CL}(C_S, cfid_T)$ with $Card_{cs}(cs) = \{0\}$. We know that this configuration step is in the set of configuration steps $cs \in Steps$ of a configuration decision $cd = (\varphi_T, Steps) \in CL$ and that $C_S \models \varphi_T$ because $cs \in \mathcal{CS}_{CL}(C_S, cfid_T)$ (see Definition 6.2.6). Now we construct the set of configuration steps of configuration decision cd for $cfid_T$ (see Definition 6.3.1) $\mathcal{CS}_{CL}(cd, cfid_T)$. Obviously, $cs \in \mathcal{CS}_{CL}(cd, cfid_T)$ holds directly by construction. The reapplication of Lemma 6.2.3 results in the fact that $Card_{cs}(\mathcal{CS}_{CL}(cd, cfid_T)) = \{0\}$. Therefore, $\varphi_T \in ARev_{CL}(cfid_T, 0)$ (see Definition 6.5.1). Since $C_S \models \varphi_T$, it follows, by Definition 6.5.4, that $C_S \models \pi^{-1} \circ \sqcup(ARev_{CL}(cfid_T, 0)) = \pi^{-1} \circ ARevExc(cfid_T)$ holds under the assumption that $\tau_{CL(C_S)}(cfid_T) = \bot$.

Case 4: $isCloned_T(f(cfid_T))$.
Since $\tau_{CL(C_S)}(cfid_T) = \bot$, it follows that $Card_{CL(C_S)}(cfid_T) = \{0\}$ or $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$ (see Definition 6.1.8). We distinguish these cases in the following. It is important to note that $ARevExc_{CL}^{\mathcal{CFID}}(cfid_T) = \sqcup(ARev_{CL}(cfid_T, 0)) \vee \neg \sqcup (ARev'_{CL}(cfid_T))$ holds (see Definition 6.5.4) since $cfid_T$ points to an instance of a cloned feature.

Subcase 4.1: $Card_{CL(C_S)}(cfid_T) = \{0\}$.
The proof of this case is analogous to the proof of case 3. It holds that $C_S \models \pi^{-1} \circ ARevExc(cfid_T)$ under the assumption $Card_{CL(C_S)}(cfid_T) = \{0\}$.

Subcase 4.2: $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$.
We have to show that $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ under the assumption $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$. Note that $\forall \varphi \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi$ is a sufficient condition for $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ (see Definition 6.5.4). We will prove this statement by showing the contraposition $\exists \varphi \in ARev'_{CL}(cfid_T) : C_S \models \varphi \implies i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$. Let $C_S \models \varphi_T$ hold for $\varphi_T \in ARev'_{CL}(cfid_T)$. The definition of $ARev'$ (see Definition 6.5.1) implies the

existence of configuration decision $cd \in CL$ with $cd = (\varphi_T, Steps)$ and $i(cfid_T) \in Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))} cs)$. Since the combination of configuration steps (see Definition 6.2.2) is defined as union with respect to the instance names, we know of the existence of configuration step $cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))$ with $i(cfid_T) \in Inst_{cs}(cs)$. The fact $C_S \models \varphi_T$ allows the application of Lemma 6.3.2 and we obtain $\mathcal{CS}_{CL}(cd, NoInst(cfid_T)) \subseteq \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))$. Hence the statement $cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))$ holds. This leads to the fact $i(cfid_T) \in Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))} cs)$ since instance names of configuration steps are unified. By Definition 6.2.7, we obtain $i(cfid_T) \in Inst(Cfg_{CL}(C_S, NoInst(cfid_T)))$ and $CL(C_S)(NoInst(cfid_T)) = Cfg_{CL}(C_S, NoInst(cfid_T))$. Consequently, the statement $i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$ holds. By contraposition, it follows that $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ holds under the assumption that $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$.

The proven properties $Card_{CL(C_S)}(cfid_T) = \{0\} \implies C_S \models \pi^{-1} \circ ARevExc(cfid_T)$ and $i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T)) \implies C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ of case 4.1 and case 4.2 and the already mentioned fact $\tau_{CL(C_S)}(cfid_T) = \bot \implies Card_{CL(C_S)}(cfid_T) = \{0\} \vee i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T))$ lead to $C_S \models \pi^{-1} \circ ARevExc(cfid_T)$ for instances of cloned features (case 4) under the assumption that $\tau_{CL(C_S)}(cfid_T) = \bot$.

We showed the property $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T) \impliedby \tau_{CL(C_S)}(cfid_T) = \bot$ for mandatory features (case 1), abstract features (case 2), optional features (case 3) and instances of cloned features (case 4). It follows that this property holds for every $cfid_T \in \mathcal{CFIDI}_T$.

If-part ("$\implies$"). Let $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ hold.
We have to show that $\tau_{CL(C_S)}(cfid_T) = \bot$. Analogous to the only-if-part, we distinguish four cases for $cfid_T$ and show the property for every case. It is important that these cases are complete, i.e. every $cfid_T \in \mathcal{CFIDI}_T$ belongs to exactly one case.

Case 1: $Card_T(f(cfid_T)) = \{1\}$.
Impossible since $C_S \not\models \bot \equiv \pi^{-1}([\bot]_{\equiv}) \equiv \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ by Definition 6.5.4. This is a direct contradiction to the assumption that $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$.

Case 2: $Card_T(f(cfid_T)) = \{0\}$.
$\tau_{CL(C_S)}(cfid_T) = \bot$ follows directly from Definition 6.1.8.

Case 3: $Card_T(f(cfid_T)) = \{0, 1\}$.
Let $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ hold. By Definition 6.5.4, we obtain the existence of $\varphi_T \in ARev_{CL}(cfid_T, 0)$ with $C_S \models \varphi_T$. Therefore, we know that configuration decision $cd \in CL$ with $cd = (\varphi_T, Steps)$ and $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, cfid_T)} cs) = \{0\}$ exists (see Definition 6.5.1). Because of Lemma 6.3.2, the fact that $\mathcal{CS}_{CL}(cd, cfid_T)$ is a subset of $\mathcal{CS}_{CL}(C_S, cfid_T)$ (see Definition 6.3.1 and Definition 6.2.6) holds. By applying Lemma 6.2.3, we conclude that $Card_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, cfid_T)} cs) = \{0\}$ and,

by Definition 6.2.7, $Card(Cfg_{CL}(C_S, cfid_T)) = \{0\}$, i.e. $Card(CL(C_S)(cfid_T)) = \{0\}$. Thus, $\tau_{CL(C_S)}(cfid_T) = \bot$ (see Definition 6.1.8).

Case 4: *$isCloned_T(f(cfid_T))$.*
Let $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$ hold. This directly implies that $\exists \varphi_T \in ARev_{CL}(cfid_T, 0) : C_S \models \varphi_T$ or $\forall \varphi_T \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi_T$ by Definition 6.5.4. We have to show that $\tau_{CL(C_S)}(cfid_T) = \bot$ holds in both cases.

Subcase 4.1: $\exists \varphi_T \in ARev_{CL}(cfid_T, 0) : C_S \models \varphi_T$.
The proof is analogous to the proof of case 3 and we follow that $\tau_{CL(C_S)}(cfid_T) = \bot$ under the assumption $\exists \varphi_T \in ARev_{CL}(cfid_T, 0) : C_S \models \varphi_T$.

Subcase 4.2: $\forall \varphi_T \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi_T$.
Proof by contraposition, i.e. we will show that $\tau_{CL(C_S)}(cfid_T) \neq \bot \implies \neg \forall \varphi_T \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi_T \equiv \exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$. Let $\tau_{CL(C_S)}(cfid_T) \neq \bot$ hold. Then we can distinguish the following two cases 4.2.1 and 4.2.2 and we will show that both cases lead to $\exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$.

Subcase 4.2.1: *$\tau_{CL(C_S)}(cfid_T)$ is not defined.*
By Definition 6.1.8, we know that the fact $\neg(Card_{CL(C_S)}(cfid_T) = \{1\} \wedge i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T)))$ and the fact $\neg(Card_{CL(C_S)}(cfid_T) = \{0\} \vee i(cfid_T) \notin Inst_{CL(C_S)}(NoInst(cfid_T)))$ hold. Consequently, the statements $Card_{CL(C_S)}(cfid_T) \neq \{1\}$, $Card_{CL(C_S)}(cfid_T) \neq \{0\}$ and $i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$ hold. We use the last-mentioned statement to follow from Definition 6.2.7 that $i(cfid_T) \in Inst(Cfg_{CL}(C_S, NoInst(cfid_T))) = Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))} cs)$. Since $+_{\mathcal{CS}}$ is defined as union with respect to the instance names (see Definition 6.2.2), a configuration step $cs \in \mathcal{CS}_{CL}(C_S, NoInst(cfid_T))$ with $i(cfid_T) \in Inst_{cs}(cs)$ exists. Definition 6.2.6 states the existence of a configuration decision $cd \in CL$ with $cd = (\varphi_T, Steps)$, $C_S \models \varphi_T$ and $cs \in Steps$. It also holds that $cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))$ by construction (see Definition 6.3.1). Now we use the fact that instance names are unified by $+_{\mathcal{CS}}$ (see Definition 6.2.2) again and follow the statement $i(cfid_T) \in Inst_{cs}(\sum_{cs \in \mathcal{CS}_{CL}(cd, NoInst(cfid_T))} cs)$. By the definition of $ARev'_{CL}$ (see Definition 6.5.1), we obtain that $\varphi_T \in ARev'_{CL}(cfid_T)$. Remember that $C_S \models \varphi_T$ by construction, i.e. the statement $\exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$ is shown under the assumption *$\tau_{CL(C_S)}(cfid_T)$ is not defined.*

Subcase 4.2.2: $\tau_{CL(C_S)}(cfid_T) = \top$.
We know that $Card_{CL(C_S)}(cfid_T) = \{1\}$ and $i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$ hold (see Definition 6.1.8). The further proof of this case is analogous to (a part of) the proof of case 4.2.1. We have already shown there that the statement $i(cfid_T) \in Inst_{CL(C_S)}(NoInst(cfid_T))$ implies that $\exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$. It follows that $\exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$ holds under the assumption $\tau_{CL(C_S)}(cfid_T) = \top$.

The proven statements *$\tau_{CL(C_S)}(cfid_T)$ is not defined* $\implies \exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$ of case 4.2.1 and $\tau_{CL(C_S)}(cfid_T) = \top \implies \exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models$

φ_T of case 4.2.2 lead to $\tau_{CL(C_S)}(cfid_T) \neq \bot \implies \exists \varphi_T \in ARev'_{CL}(cfid_T) : C_S \models \varphi_T$ and, by contraposition, $\forall \varphi_T \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi_T \implies \tau_{CL(C_S)}(cfid_T) = \bot$.

The shown properties $\exists \varphi_T \in ARev_{CL}(cfid_T, 0) : C_S \models \varphi_T \implies \tau_{CL(C_S)}(cfid_T) = \bot$ of case 4.1 and $\forall \varphi_T \in ARev'_{CL}(cfid_T) : C_S \not\models \varphi_T \implies \tau_{CL(C_S)}(cfid_T) = \bot$ of case 4.2 lead to the fact that $\tau_{CL(C_S)}(cfid_T) = \bot$ holds for instances of cloned features (case 4) under the assumption that $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T)$.

We showed the property $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T) \implies \tau_{CL(C_S)}(cfid_T) = \bot$ for mandatory features (case 1), abstract features (case 2), optional features (case 3) and instances of cloned features (case 4). It follows that this property holds for every $cfid_T \in \mathcal{CFIDI}_T$.

Since the statement is valid both for the only-if- and the if-part, the equivalence $C_S \models \pi^{-1} \circ ARevExc_{CL}^{\mathcal{CFID}}(cfid_T) \iff \tau_{CL(C_S)}(cfid_T) = \bot$ is proven.

□

In the next step, we introduce the definition of the advanced reverse unconfigured mapping. This is, as might be expected, a function mapping a configured feature identifier to a set of equivalent formulae such that the application of the configuration link to a configuration that fulfills these constraints leads to a target configuration in which the given configured feature identifier is unconfigured. The definition of a reverse exclusion mapping and a reverse unconfigured mapping for basic feature constraints is not possible since we cannot distinguish between excluded and unconfigured in the criteria of configuration decisions without the use of advanced feature constraints.

Definition 6.5.6 (Advanced Reverse Unconfigured Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S \to T}$ with advanced feature constraints. Then the advanced reverse unconfigured mapping of configured feature identifiers with respect to this configuration link is given by function $ARevUnconf_{CL_{S \to T}}^{\mathcal{CFID}} : \mathcal{CFIDI}_T \to FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevUnconf_{CL_{S \to T}}^{\mathcal{CFID}}(cfid_T) = \neg ARevInc_{CL_{S \to T}}^{\mathcal{CFID}}(cfid_T) \wedge \neg ARevExc_{CL_{S \to T}}^{\mathcal{CFID}}(cfid_T)$$

This definition is quite intuitive: a configured feature identifier (pointing to a non-cloned or an instance of a cloned feature) is unconfigured if it is neither included nor excluded.

We prove correctness and minimality of the advanced reverse unconfigured mapping in the following lemma.

Lemma 6.5.7 (Correctness and Minimality of the Advanced Reverse Unconfigured Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S \to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ ARevUnconf_{CL_{S \to T}}^{\mathcal{CFID}}(cfid_T) \iff \tau_{CL_{S \to T}(C_S)}(cfid_T) \text{ is not defined}$$

Proof.

$$C_S \models \pi^{-1} \circ ARevUnconf^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T)$$
$$\overset{\text{Definition 6.5.6}}{\Longleftrightarrow}$$
$$C_S \models \pi^{-1} \begin{pmatrix} \neg ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \wedge \neg ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \end{pmatrix}$$
$$\overset{\pi \text{ is a projection and logic}}{\Longleftrightarrow}$$
$$\begin{pmatrix} C_S \not\models \pi^{-1} \circ ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \text{and } C_S \not\models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \end{pmatrix}$$
$$\overset{\text{Lemma 6.5.3 and 6.5.5}}{\Longleftrightarrow}$$
$$\begin{pmatrix} \tau_{CL_{S\to T}(C_S)}(cfid_T) \neq \top \\ \text{and } \tau_{CL_{S\to T}(C_S)}(cfid_T) \neq \bot \end{pmatrix}$$
$$\overset{\text{Definition 6.1.8}}{\Longleftrightarrow}$$
$$\tau_{CL_{S\to T}(C_S)}(cfid_T) \textit{ is not defined}$$

□

The next definition introduces the advanced reverse selection mapping of configured feature identifiers. The idea of this mapping is the same as for the reverse selection mapping for basic feature constraints (see Definition 6.3.9): it maps a configured feature identifier to a set of equivalent formulae, whose satisfaction is sufficient for the selection of the given configured feature identifier. It takes the tree-structure of the target feature model into account. Since the selection of configured feature identifiers (see Definition 6.1.9) is defined both for basic feature constraints and advanced feature constraints in the same manner, the following definition of the advanced reverse mapping is very similar to its basic counterpart. After the definition, the lemma that states its correctness and minimality is presented.

Definition 6.5.8 (Advanced Reverse Selection Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ with advanced feature constraints. Then the advanced reverse selection mapping of configured feature identifiers with respect to this configuration link is given by function $ARevSel^{\mathcal{CFID}}_{CL_{S\to T}} : \mathcal{CFIDI}_T \to FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) = \begin{cases} ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } isRoot_T(f(cfid_T)) \\ \begin{pmatrix} ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \wedge ARevSel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \end{pmatrix} & else \end{cases}$$

Lemma 6.5.9 (Correctness and Minimality of the Advanced Reverse Selection Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T*

with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.

$$C_S \models \pi^{-1} \circ ARevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff CL_{S\to T}(C_S) \vdash cfid_T$$

Proof. Analogous to the proof of the corresponding lemma for basic feature constraints (Lemma 6.3.10) since the correctness and minimality also hold for the advanced reverse inclusion mapping of configured feature identifiers (see Lemma 6.5.3) and the reverse selection mapping of configured feature identifiers is defined analogously for basic and advanced feature constraints (see Definition 6.3.9 and Definition 6.5.8). □

We do not need an advanced reverse deselection mapping for the definition of the transformation of advanced feature constraints. This is because we only need the (advanced) reverse selection mapping of configured feature identifiers for the definition of the (advanced) reverse selection mapping of features, which in turn is required for the transformation of feature links. Feature links are expressed by propositional logic formulae over the features (see Definition 6.1.12). This means that we only have to differ between selected and not selected on feature level and do not need the possibility to express deselection. Therefore, we do not need (advanced) deselection mappings – neither for configured feature identifiers nor for features. For the sake of completeness, we show that this mapping can be defined straightforwardly in Section A.3. In addition, we prove its correctness and minimality in this section.

According to the case of basic feature constraints (cf. Section 6.3), we continue with the definition of the advanced reverse selection mapping of features and the lemma that states its correctness and minimality. Once again, the definition of an advanced mapping is very similar to its counterpart for basic feature constraints.

Definition 6.5.10 (Advanced Reverse Selection Mapping of Features w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ with advanced feature constraints. Then the advanced reverse selection mapping of features with respect to this configuration link is given by function $ARevSel^{F}_{CL_{S\to T}} : F_T \to FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevSel^{F}_{CL_{S\to T}}(f_T) = \bigsqcup \left(\left\{ \varphi \in FormA(\mathcal{CFIDI}_S) \;\middle|\; \begin{array}{c} cfid_T \in \mathcal{CFIDI}_T \\ \wedge f(cfid_T) = f_T \\ \wedge \varphi \in ARevSel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \end{array} \right\} \right)$$

Lemma 6.5.11 (Correctness and Minimality of the Advanced Reverse Selection Mapping of Features)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a feature $f_T \in F_T$ of feature model T. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R*

being a complete system of representatives.

$$C_S \models \pi^{-1} \circ ARevSel^F_{CL_{S\to T}}(f_T)$$
$$\Longleftrightarrow$$
$$\exists cfid_T \in \mathcal{CFIDI}_T : f(cfid_T) = f_T \wedge CL_{S\to T}(C_S) \vdash cfid_T$$

Proof. Analogous to the proof of the corresponding lemma for basic feature constraints (Lemma 6.3.12) since the correctness and minimality also hold for the advanced reverse selection mapping of configured feature identifiers (see Lemma 6.5.9) and the reverse selection mapping of features is defined analogously for basic and advanced feature constraints (see Definition 6.3.11 and Definition 6.5.10). □

Now we are ready to define the advanced transformations of logical formulae. The idea is the same as in the case of basic feature constraints (see Definition 6.3.13) and the definition is similar.

Definition 6.5.12 (Advanced Transformation of Logical Formulae)**.** Given a complete system of representatives $R \subseteq FormA(\mathcal{CFIDI}_S)$ of $\equiv$, the bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ defined by $\pi(\varphi) = [\varphi]_\equiv$, a configuration link $CL_{S\to T}$ with advanced feature constraints and a predicate logic formula $\varphi_T \in FormA(\mathcal{CFIDI}_T)$ (resp. a propositional logic formula $\varphi_T \in Form(F_T)$). Then the transformation of this formula with respect to the configuration link is recursively defined as $ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi} : FormA(\mathcal{CFIDI}_T) \to FormA(\mathcal{CFIDI}_S)$ (resp. $ATrafo^{F}_{CL_{S\to T},\pi} : Form(F_T) \to FormA(\mathcal{CFIDI}_S)$) with

$$ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T) = \begin{cases} \pi^{-1} \circ ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } \varphi_T = inc(cfid_T) \text{ and } cfid_T \in \mathcal{CFIDI}_T \\ \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } \varphi_T = exc(cfid_T) \text{ and } cfid_T \in \mathcal{CFIDI}_T \\ \pi^{-1} \circ ARevUnconf^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } \varphi_T = unconf(cfid_T) \text{ and } cfid_T \in \mathcal{CFIDI}_T \\ \neg ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi'_T) & \text{if } \varphi_T = \neg\varphi'_T \text{ and } \varphi'_T \in FormA(\mathcal{CFIDI}_T) \\ \begin{pmatrix} ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi'_T) \\ \otimes ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi''_T) \end{pmatrix} & \text{if } \varphi_T = \varphi'_T \otimes \varphi''_T \text{ with } \otimes \in \{\wedge, \vee, \to, \leftrightarrow\} \text{ and } \varphi'_T, \varphi''_T \in FormA(\mathcal{CFIDI}_T) \end{cases}$$

$$ATrafo^{F}_{CL_{S\to T},\pi}(\varphi_T) = \begin{cases} \pi^{-1} \circ ARevSel^{F}_{CL_{S\to T}}(f_T) & \text{if } \varphi_T = f_T \in F_T \\ \neg ATrafo^{F}_{CL_{S\to T},\pi}(\varphi'_T) & \text{if } \varphi_T = \neg\varphi'_T \text{ and } \varphi'_T \in Form(F_T) \\ \begin{pmatrix} ATrafo^{F}_{CL_{S\to T},\pi}(\varphi'_T) \\ \otimes ATrafo^{F}_{CL_{S\to T},\pi}(\varphi''_T) \end{pmatrix} & \text{if } \varphi_T = \varphi'_T \otimes \varphi''_T \text{ with } \otimes \in \{\wedge, \vee, \to, \leftrightarrow\} \text{ and } \varphi'_T, \varphi''_T \in Form(F_T) \end{cases}$$

In the case of advanced feature constraints, we replace all predicates ($inc(x)$, $exc(x)$ and $unconf(x)$) of a formula by the corresponding reverse mappings. In the case of feature links, the definition is almost the same as its counterpart for basic feature constraints (see Definition 6.3.13). However, we have to apply the advanced version of the reversed selection mapping since criteria of the underlying configuration link contain advanced feature constraints and not basic feature constraints. Note that this transformation is no classical substitution since it replaces predicates and not literals by formulae.

In the following, correctness and minimality of the advanced transformation are shown.

Theorem 6.5.13 (Correctness and Minimality of the Advanced Transformation of Feature Constraints)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and an advanced feature constraint $\varphi_T \in FormA(\mathcal{CFIDI}_T)$. Then the following holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_T) \iff CL_{S\to T}(C_S) \models \varphi_T$$

Proof. We prove this statement by structural induction.

Basis step. We show the property for all atomic formulae. Note that the signature (see Definition 6.4.1) does not contain constants and equality is not a logical symbol. Therefore, we can distinguish three sorts of atoms: $inc(cfid_T)$, $exc(cfid_T)$ and $unconf(cfid_T)$ for $cfid_T \in \mathcal{CFIDI}_T$ (see Definition 6.4.2).

Let $\varphi = inc(cfid_T)$.

$$\begin{aligned} & C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(inc(cfid_T)) \\ \iff\ & C_S \models \pi^{-1} \circ ARevInc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Definition 6.5.12} \\ \iff\ & \tau_{CL_{S\to T}(C_S)}(cfid_T) = \top && \text{by Lemma 6.5.3} \\ \iff\ & CL_{S\to T}(C_S) \models inc(cfid_T) && \text{by Definition 6.4.4} \end{aligned}$$

Let $\varphi = exc(cfid_T)$**.**

$$
\begin{aligned}
& C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(exc(cfid_T)) \\
\iff\ & C_S \models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Definition 6.5.12} \\
\iff\ & \tau_{CL_{S\to T}(C_S)}(cfid_T) = \bot && \text{by Lemma 6.5.5} \\
\iff\ & CL_{S\to T}(C_S) \models exc(cfid_T) && \text{by Definition 6.4.4}
\end{aligned}
$$

Let $\varphi = unconf(cfid_T)$**.**

$$
\begin{aligned}
& C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(unconf(cfid_T)) \\
\iff\ & C_S \models \pi^{-1} \circ ARevUnconf^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Definition 6.5.12} \\
\iff\ & \tau_{CL_{S\to T}(C_S)}(cfid_T) \textit{ is not defined} && \text{by Lemma 6.5.7} \\
\iff\ & CL_{S\to T}(C_S) \models unconf(cfid_T) && \text{by Definition 6.4.4}
\end{aligned}
$$

Inductive step. Let $C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi) \iff CL_{S\to T}(C_S) \models \varphi$ hold for $\varphi = \varphi_1$ and $\varphi = \varphi_2$ (inductive hypothesis) and let φ_T be a non-atomic formula.

Let $\varphi_T = \neg\varphi_1$**.**

$$
\begin{aligned}
& C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\neg\varphi_1) \\
\iff\ & C_S \models \neg ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_1) && \text{by Definition 6.5.12} \\
\iff\ & C_S \not\models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_1) && \text{by logic} \\
\iff\ & CL_{S\to T}(C_S) \not\models \varphi_1 && \text{by inductive hypothesis} \\
\iff\ & CL_{S\to T}(C_S) \models \neg\varphi_1 && \text{by logic}
\end{aligned}
$$

Let $\varphi_T = \varphi_1 \wedge \varphi_2$**.**

$$
\begin{aligned}
& C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_1 \wedge \varphi_2) \\
\iff\ & C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_1) \wedge ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_2) && \text{by Definition 6.5.12} \\
\iff\ & \begin{pmatrix} C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_1) \\ \text{and } C_S \models ATrafo^{\mathcal{CFID}}_{CL_{S\to T},\pi}(\varphi_2) \end{pmatrix} && \text{by logic} \\
\iff\ & \begin{pmatrix} CL_{S\to T}(C_S) \models \varphi_1 \\ \text{and } CL_{S\to T}(C_S) \models \varphi_2 \end{pmatrix} && \text{by inductive hypothesis} \\
\iff\ & CL_{S\to T}(C_S) \models \varphi_1 \wedge \varphi_2 && \text{by logic}
\end{aligned}
$$

The proofs for the other logical connectives $\vee$, $\rightarrow$ and $\leftrightarrow$ are analogous to the last proof. Therefore they are omitted. $\square$

Theorem 6.5.14 (Correctness and Minimality of the Advanced Transformation of Feature Links)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a feature constraint $\varphi_T \in Form(F_T)$ over the set of features. Then the following holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models ATrafo^F_{CL_{S\to T},\pi}(\varphi_T) \iff CL_{S\to T}(C_S) \models \varphi_T$$

Proof. We prove this statement by structural induction.

Basis step. We show the property for all atomic formulae. Note that $\varphi_T \in Form(F_T)$ is a propositional logic formula over the features of T.

Let $\varphi = f_T$ with $f_T \in F_T$.

$$C_S \models ATrafo^F_{CL_{S\to T},\pi}(f_T)$$
$$\overset{\text{by Definition 6.5.12}}{\iff}$$
$$C_S \models \pi^{-1} \circ ARevSel^F_{CL_{S\to T}}(f_T)$$
$$\overset{\text{by Lemma 6.5.11}}{\iff}$$
$$\exists cfid_T \in \mathcal{CFIDI}_T : f(cfid_T) = f_T \wedge CL_{S\to T}(C_S) \vdash cfid_T$$
$$\overset{\text{by Definition 6.1.11}}{\iff}$$
$$CL_{S\to T}(C_S) \models f_T$$

Inductive step. Analogous to the inductive step of the proof of Theorem 6.5.13. □

Correctness and minimality of the transformation of advanced feature constraint sets (cf. Theorem 6.3.16) follow directly from the theorems proven above.

Chapter 7

Feature Constraint Propagation: Implementation and Evaluation

Up to now, the technique of feature constraint propagation has been introduced on a conceptual level (cf. Chapter 5), formalized and verified (cf. Chapter 6). Now the evaluation of the technique is described. We have chosen two case studies as evaluation: an experimental study and an industrial study. The former study consists of automated functional tests and performance tests for a proof of concept and a runtime analysis, whereas the industrial study aims to show the practical applicability of the feature constraint propagation in a realistic example. Both types of evaluation require an implementation of the technique, which is also described in this chapter.

This chapter is organized as follows. Section 7.1 introduces the implementation of feature constraint propagation. Subsequently, the evaluation of the technique is described in the following two sections. Section 7.2 presents the experimental case study and Section 7.3 the industrial case study. Finally, limitations of feature constraint propagation are discussed in Section 7.4.

7.1 Implementation of Feature Constraint Propagation

Since feature constraint propagation requires configuration links and the feature modeling tool CVM implements all required concepts, we decided to extend this tool by feature constraint propagation. Hence, the CVM framework is described before our implementation.

7.1.1 CVM: A Framework for Compositional Variability Management

The *Compositional Variability Management framework* [AJL+10, CVM12] – short CVM – is a feature modeling tool providing the possibility to organize the variability not only in one monolithic feature model but in several hierarchically organized feature models. CVM was initially implemented within the dissertation [Rei08] but it is still evolving. All feature modeling concepts introduced in this thesis including

configuration links are covered by CVM. However, CVM does not allow to define multi feature links. The framework is implemented as Eclipse plug-in [CR08] and is completely integrated in the popular Eclipse UI [Ecl12]. Figure 7.1 shows a screenshot of the example in Figure 2.5 in CVM. There are two views on a model – the tree editor (on the left-hand side in the figure) and the graphical editor (on the right-hand side in the figure) – both providing editing functionalities. The configuration link of the example is only depicted in the tree editor (CL). In addition to the formal definition, CVM provides the possibility to annotate every artifact of a model, which is very important for practical use. The screenshot contains descriptions for all configuration decisions. The bottom of the figure shows a new integrated view of CVM: the configuration preview. If a configuration link is selected in the editor, the configuration preview shows its source feature models on the left-hand side and allows to configure them. On the right-hand side, the target feature models of the configuration link are shown and the effects of the given source configuration are previewed. In the figure we can see that the selection of `USA` and `Comfort` leads, as already mentioned, to an invalid target configuration because of the selection of `front:Wiper.RainSensor` and `Radar`. Configuration links can either be defined by textual input or the configuration preview can be used to edit individual configuration decisions. If a configuration decision is selected, the configuration preview allows graphical editing, as Figure 7.2 shows for the third configuration decision. Configurations of feature models can be created and edited in a new window with an integrated check for validity, as depicted in Figure 7.3. They are saved within the variability model. Configuration links can be applied to source configurations via mouse click. CVM is compatible with partial configurations and incremental configuration (cf. Section 5.2.3). Furthermore, CVM is equipped with an editor for VSL statements (cf. Section A.1), as can be seen in Figure 7.4, and a VSL converter allowing to convert VSL specifications of variability models (feature models with configuration links) into CVM files and back. More detailed information on CVM and its use can be found in [Rei08, Rei09].

CVM itself is implemented in Java and uses the Eclipse plug-in mechanism [CR08] to interact with Eclipse. To be precise, the framework is not developed as a single Eclipse plug-in but consists of several bundled Eclipse plug-ins. The core of the framework is represented by a data model that implements all variability modeling concepts formalized in Section 6.1 and Section 6.2. In addition, it contains the implementation of the persistence of variability models in XMI standard. The data model was partly generated with the Eclipse Modeling Framework (EMF) [SBPM09, EMF12]. This framework contains a graphical editor allowing to create data models in a UML-like graphical notation and provides code generating functionalities to generate code for the data model and for simple tree editors. The data model as well as the tree editor of CVM are partly generated by EMF as separate Eclipse plug-ins. The graphical editor of CVM was implemented by using the Graphical Editing Framework (GEF) [CR08, GEF12]. This is a framework providing the technology to create graphical editors and views for a structured data model integrated in the Eclipse UI. Furthermore, CVM contains some additional and optional components in form of separate plug-ins, which are neglected here.

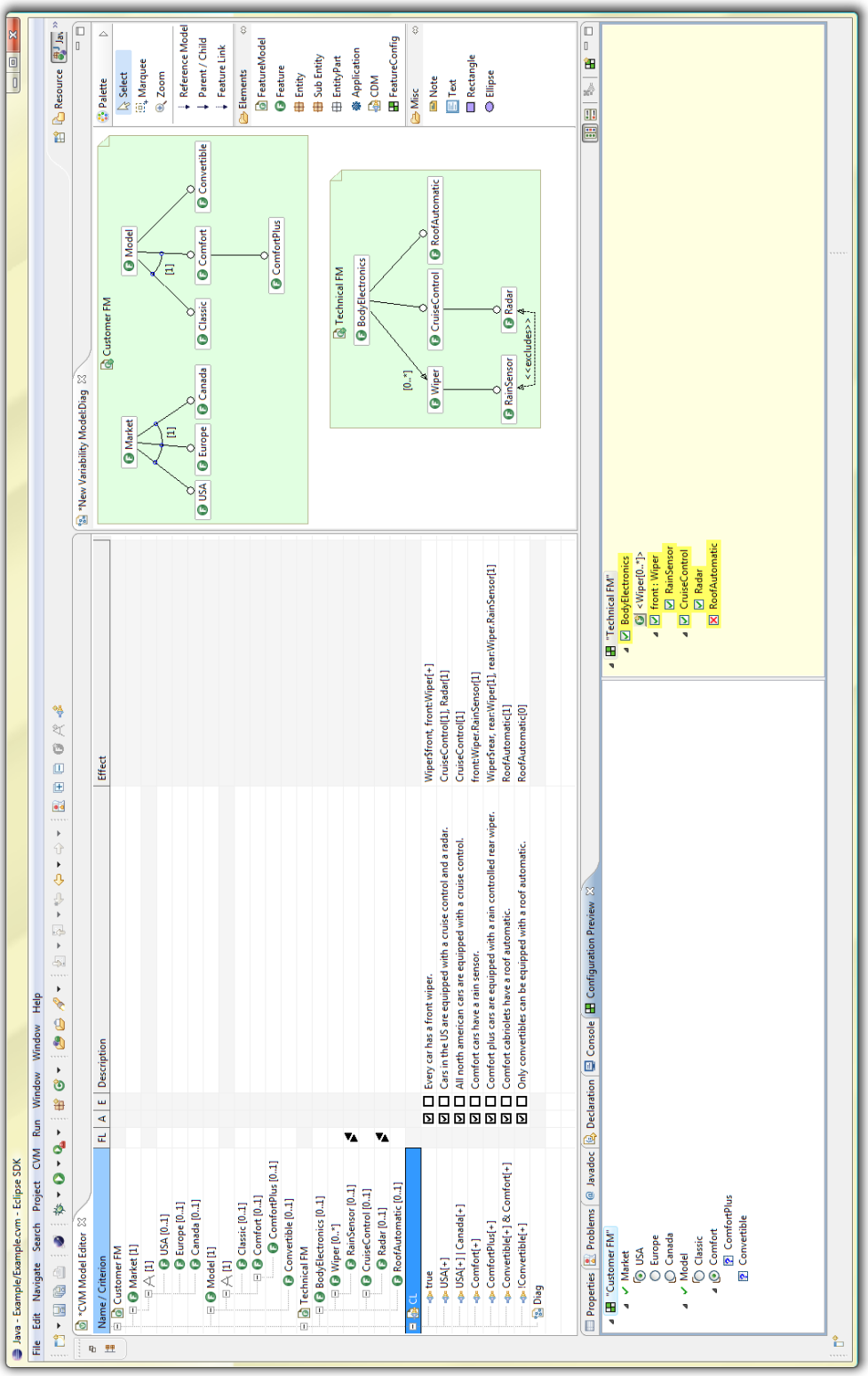


Figure 7.1: CVM screenshot

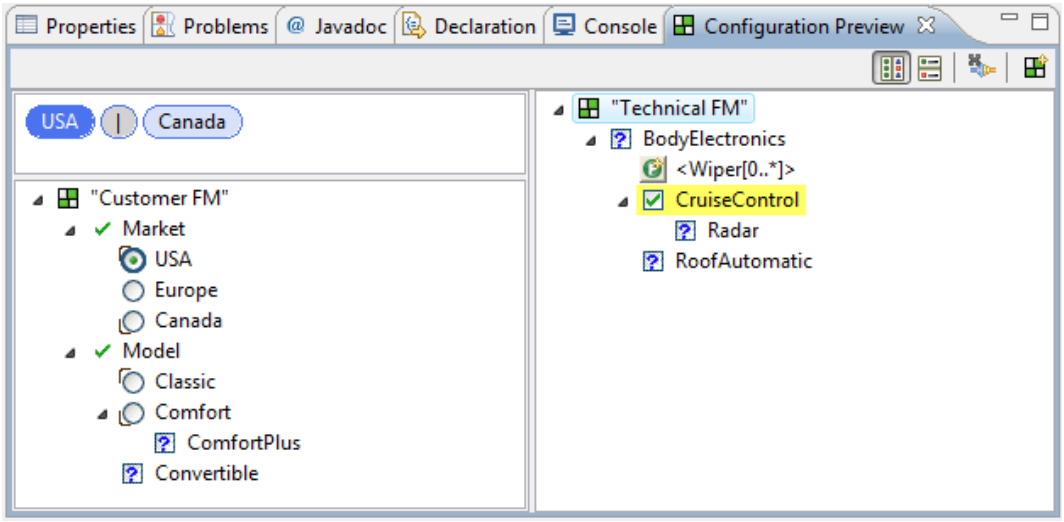


Figure 7.2: CVM screenshot: editing a configuration decision

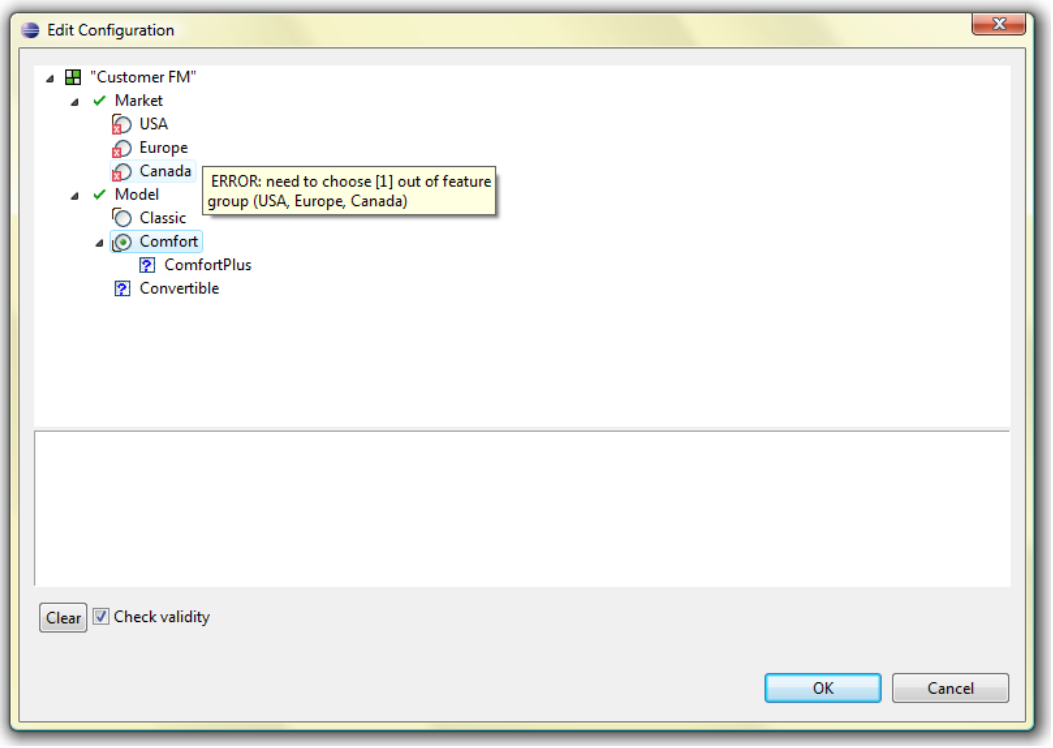


Figure 7.3: CVM screenshot: creating a configuration

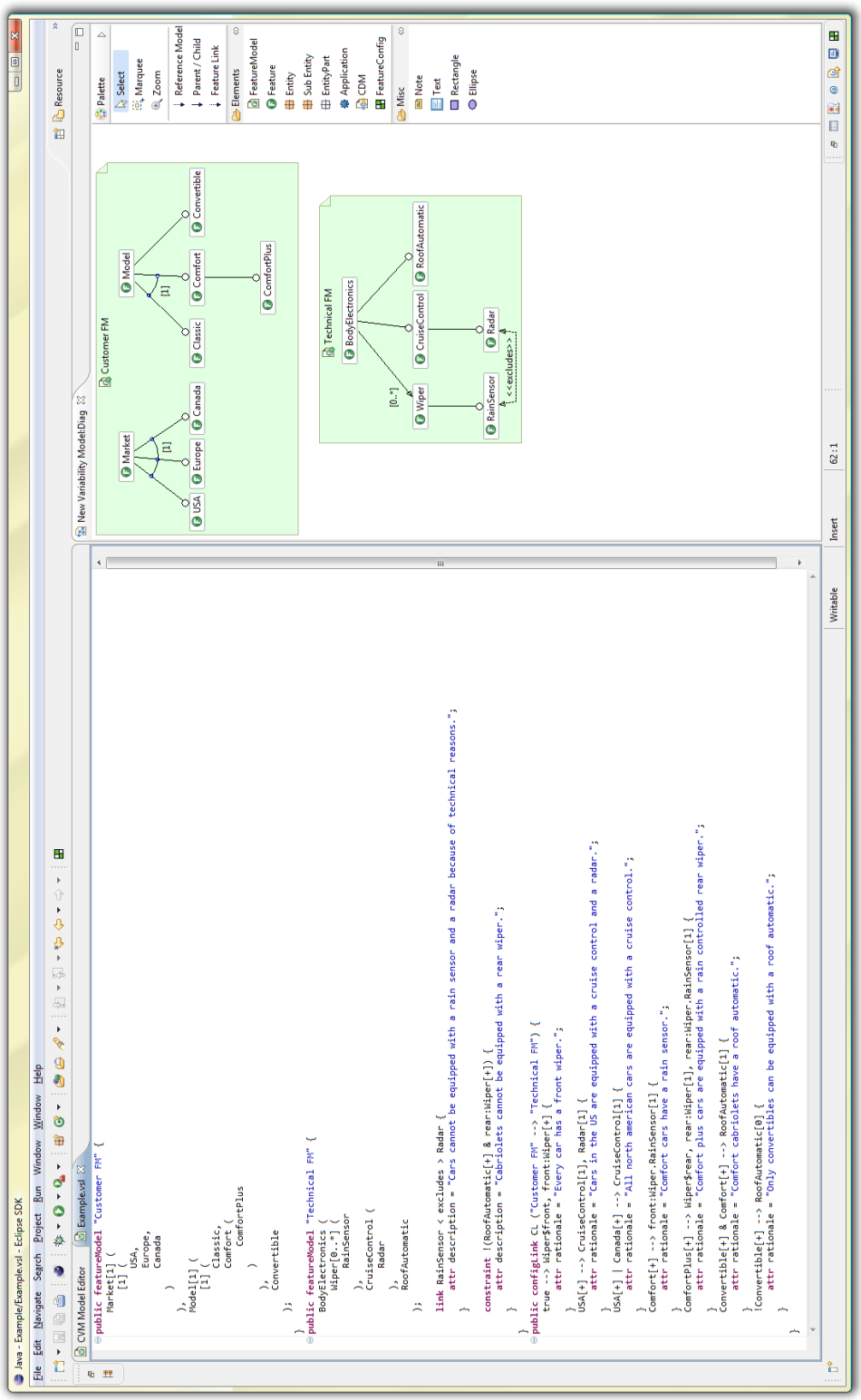


Figure 7.4: CVM screenshot: VSL editor

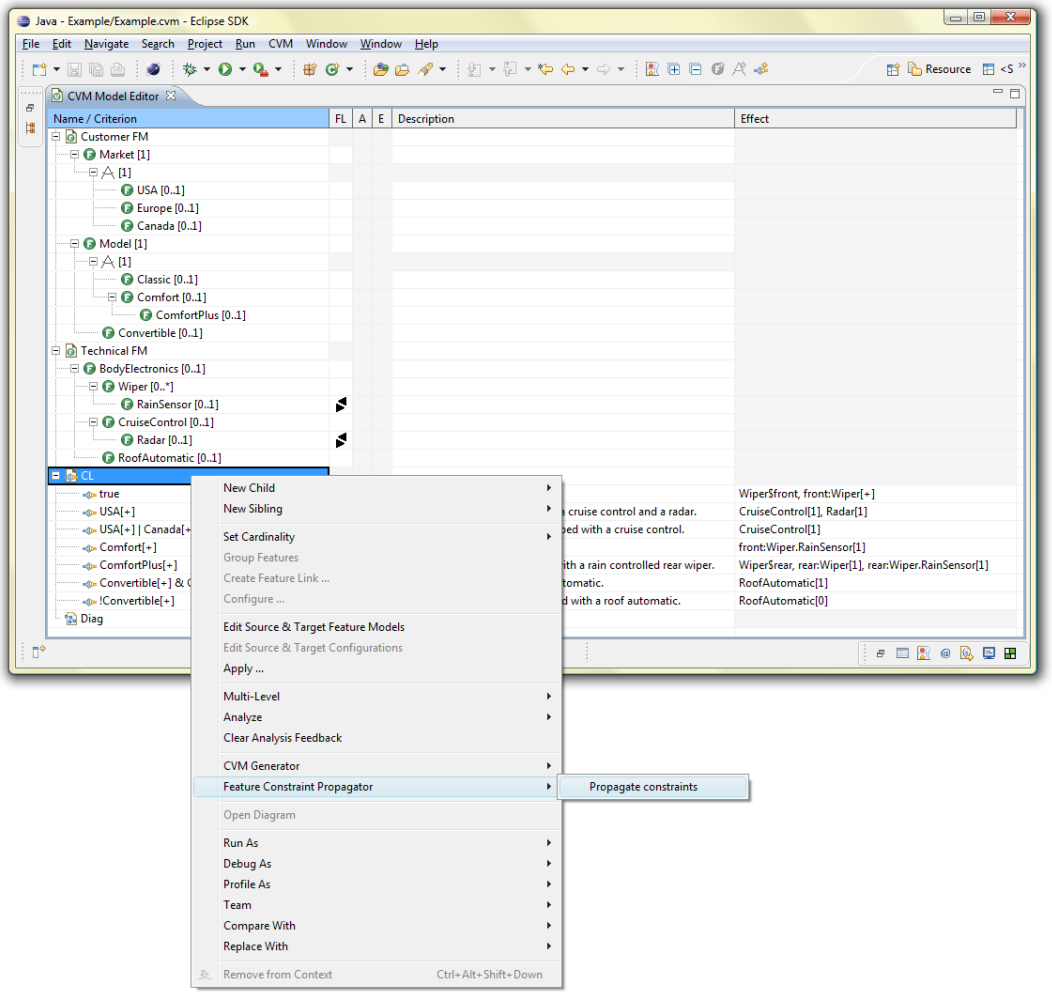


Figure 7.5: CVM screenshot: integrated feature constraint propagator

7.1.2 Extending CVM by Feature Constraint Propagation

Feature constraint propagation is implemented as an own Eclipse plug-in, which complements CVM. After installation, the pop-up menus of configuration links are extended by an additional item, depicted in Figure 7.5, whose selection starts the propagation of all feature constraints and feature links of the target feature model, as described in Section 5.1. Up to now, the implementation covers only the basic version of the propagation and does not comprise the advanced considerations of Section 5.2.

From a technical point of view, feature constraint propagation is not a single Eclipse plug-in but consists of three bundled Eclipse plug-ins: (1) the constraint propagator plug-in, (2) the interface for the logic minimizer and (3) the logic minimizer. All plug-ins are written in Java. The interface and the logic minimizer were separated from the constraint propagator plug-in because the minimizer is ex-

changeable. With this design, the replacement of the minimizer is possible without any changes in the constraint propagator plug-in. CVM is not dependent on feature constraint propagation, which means that feature constraint propagation can be installed or deinstalled without changing CVM itself. Feature constraint propagation requires the CVM core (the data model implementation) and the tree editor of CVM. In principle, only the CVM core is required. The tree editor is only used to connect feature constraint propagation with the UI of CVM. This connection is realized by the use of Eclipse's extension point `org.eclipse.ui.popupMenus`, which allows the integration of a menu item to a specific pop-up menu [CR08]. If this new menu item is chosen, the constraint propagator plug-in gets a reference to the selected configuration link of the data model and the propagation starts in a new Java thread. Its progress is shown in a progressbar and it can be aborted anytime.

In the first step of the propagation, all required information of the data model (source and target models and target-side constraints) is collected in the *constraint propagator*. Then an expanded copy of the configuration link (cf. Section 5.1.1) is created and, on this basis, a *reverse mapper* according to the configuration link is constructed (cf. Section 5.1.2). The reverse mapper class realizes all three reverse mappings of the propagation. The reverse inclusion mapping is initially calculated for all configured feature identifiers of the configuration link and stored in a Java `HashMap` for fast access. The two reverse selection mappings are calculated for individual configured feature identifiers (resp. features) if needed and stored for later access. Afterwards, the transformation (cf. Section 5.1.3) is applied to all feature constraints and feature links of the target model by mapping the variables with the reverse mapper. The resulting constraint is returned in the form of the internal data structure for constraints of CVM. This constraint is now extracted as string and passed via the interface for the logic minimizer to the logic minimizer plug-in.

The logic minimizer plug-in minimizes arbitrary propositional logic formulae to minimal conjunctive or disjunctive normal forms by the ESPRESSO algorithm. It contains the original ESPRESSO implementation of Richard Rudell (University of California, Berkeley) written in C [Esp12]. This implementation requires a special representation of the formula as input – the "Berkeley standard format for the physical description of a programmable logical array (PLA)". In principle, this is a string containing the individual lines of the truth table of the formula. ESPRESSO does not need the whole truth table but only parts of it (e.g. the ON-set – lines evaluating to `true` – is sufficient). In the first step of the minimization process, the input string is parsed. Then the ESPRESSO algorithm is started as external process and the ON-set of the truth table is passed via standard input to this process. Note that the truth table is not created completely in memory but the individual lines are calculated on the fly. The construction of the truth table is realized by an adaption of the open source software component *Truth Table Constructor* [TTC12] under the GNU General Public License and was not implemented completely by the author of this thesis. After the transfer of the data to ESPRESSO is finished, the algorithm calculates the minimal[1] OFF-set (lines of the truth table evaluating to `false`) of the formula and returns it in the Berkeley standard format (see above). In this

[1] It is not guaranteed that the result is really minimal because ESPRESSO is a heuristic algorithm.

context, minimal means that lines which are covered by other lines are neglected. The conjunctive normal form is then extracted out of this set, converted into a data structure for normal forms and returned as output (cf. Section 5.1.4).

The constraint propagator receives the formula in minimal conjunctive normal form, lifts inclusion statements to selection statements (cf. Section 5.1.5) and extracts the feature links (cf. Section 5.1.6). Details on the algorithms have already been described in Section 5.1 and are therefore neglected here.

7.1.3 Variants of Feature Constraint Propagation Implementations

During the implementation, some ideas for possible optimizations and different variants of constraint propagators and reverse mappers came up. This section introduces these variants and illustrates their intentions.

Variants of reverse mappers

- **Basic Reverse Mapper.** This is the basic variant of the reverse mapper described in Section 7.1.2. During the instantiation of this class, the reverse inclusion mapping of all configured feature identifiers included in effects of the expanded configuration link is calculated and stored in a Java `HashMap`. The reverse selection mappings are calculated lazy (i.e. they are calculated and stored for individual configured feature identifiers and features if needed). This mapper implements exactly the reverse mappings presented in Section 5.1.2.

- **Lazy Reverse Mapper.** This mapper calculates all three mappings lazy. The idea of this implementation is to optimize the runtime by calculating only the required mappings of configured feature identifiers and features.

- **Lazy Minimizing Reverse Mapper.** When a propagation is applied, the intermediate results of the propagation (including the reverse mappings) are logged to ensure a manual traceability of individual constraints. The reverse mappings, especially the reverse selection mappings, are often very long formulae containing multiple occurrences of the same variables. This is because of the semantics of selection statements in configuration links and the recursive definition of the reverse selection mapping of configured feature identifiers. In order to simplify the readability and understandability of the log, the idea to minimize the results of the individual reverse selection mappings came up and was implemented in the lazy minimizing reverse mapper. Analogously to the lazy reverse mapper, all three mappings are calculated lazy.

Variants of constraint propagators

- **Basic Constraint Propagator.** This basic variant of the constraint propagator has already been described in Section 7.1.2. It transforms all feature constraints and feature links of a feature model separately, combines the results with conjunctions and restructures the resulting formula afterwards (minimization, lifting of inclusion to selection statements and extraction of feature

links). This corresponds exactly to the concept of the propagation described in Section 5.1.

- **Minimizing Constraint Propagator.** Analogous to the lazy minimizing reverse mapper, this propagator applies the minimization algorithm after the propagation of every individual constraint. This leads to a better readability and understandability of the log. Afterwards, the minimized constraints are combined by conjunction and the same restructuring steps as in the basic constraint propagator are applied.
- **Optimized Minimizing Constraint Propagator.** This optimized variant of the minimizing propagator aborts the propagation as soon as possible. On the one hand, this means that the overall propagation aborts if a constraint equivalent to `false` is propagated from any target-side constraint. In this case, further constraints do not have to be propagated because the conjunction of `false` with an arbitrary formula is always equivalent to `false`. On the other hand, after mapping a start or an end feature of a feature link, it is checked if the resulting constraint already allows to derive the result of the propagation. In this case, the propagator does not calculate the remaining mappings of the other feature and returns the propagation result of the feature link immediately. In the case of `excludes` feature links, the propagation stops if the first feature is mapped to a constraint equivalent to `false` ($\neg(\texttt{false} \wedge x) \equiv \texttt{true}$). In the case of `needs` feature links, the propagation stops if the end feature is mapped to a constraint equivalent to `true` ($x \rightarrow \texttt{true} \equiv \texttt{true}$). Alternatively, the start feature could be mapped first and the calculation could be stopped if this results in a constraint equivalent to `false` ($\texttt{false} \rightarrow x \equiv \texttt{true}$). In the case of `alternative` feature links, the mapping of both features is always necessary. The intention of this propagator is to optimize the runtime by calculating only the required mappings and to stop the propagation as soon as possible. The actual implementation of this propagator uses the logic minimizer to check if a constraint is equivalent to `true` or `false`. To improve the runtime of the propagator, a SAT solver should be used for this activity.

7.2 Experimental Case Study: Automated Testing

Although the correctness of feature constraint propagation has already been shown in Section 6.3, we decided to additionally accomplish an experimental case study to evaluate the technique. This study comprises functional and performance testing (aka *benchmarking*). For this purpose, an automated test framework (described in Section 7.2.1) was implemented and several tests were accomplished (described in Section 7.2.2 and Section 7.2.3).

7.2.1 Automated Test Framework for Feature Constraint Propagation

The automated test framework is, analogously to feature constraint propagation, implemented as Eclipse plug-in and fully integrated into the Eclipse UI. It requires

CVM and the feature constraint propagation plug-in to run and is combined with them via the Eclipse extension point `org.eclipse.ui.popupMenus`. Automated tests can simply be started via the context menu of CVM and all results are logged into a CSV file. This file type is compatible with most spreadsheet programs and can therefore easily be processed and evaluated. All tests base on an integrated model generator. This software component can generate advanced feature models (including cloned features and inheritance) with constraints, configurations and configuration links of a specified size and complexity. The following parameters can be configured.

- generation of feature models
  - *feature count:*
    the number of features in the feature model
  - *probability for optional, mandatory, abstract and cloned features:*
    four values that specify the chance (in percentage terms) of picking a particular cardinality when a feature is generated
  - *number of inheritance relationships:*
    the number of inheritance relationships in the feature model
  - *number of feature links:*
    the number of feature links in the feature model
  - *number of feature constraints:*
    the number of feature constraints in the feature model
  - *length limiter for feature constraints:*
    a percentage value that limits the length of feature constraints (a further description is given below)
  - *probability for inclusion and selection statements in feature constraints:*
    two values that specify the chance (in percentage terms) of generating an inclusion or a selection statement in a constraint
  - *maximal instance count in constraints:*
    the maximal number of instances in constraints

- generation of configurations
  - *configuration size:*
    two values that specify the minimal and the maximal number of configured feature identifiers in a configuration
  - *maximal instance count in configurations:*
    the maximal number of instances per cloned feature in configurations

- generation of configuration links
  - *number of configuration decisions:*
    the number of configuration decisions

- *maximal instance count in configuration decisions:*
  the maximal number of instances per cloned feature in configuration decisions
- *length limiter for criteria of configuration decisions:*
  a percentage value that limits the length of criteria of configuration decisions (a further description is given below)
- *probability for inclusion and selection statements in criteria of configuration decisions:*
  two values that specify the chance (in percentage terms) of generating an inclusion or a selection statement in a criterion of a configuration decision
- *length limiter for effects of configuration decisions:*
  a percentage value that limits the length of effects of configuration decisions (a further description is given below)
- *probability for inclusion, exclusion, selection and deselection statements in effects of configuration decisions:*
  four values that specify the chance (in percentage terms) of generating an inclusion, an exclusion, a selection or a deselection statement in an effect of a configuration decision
- *probability for instance creation statements in effects of configuration decisions:*
  a value that specifies the chance (in percentage terms) of generating an instance creation statement for a cloned feature in an effect of a configuration decision

The algorithm to generate a feature model with n features works as follows. Initially, a new feature model with a root feature is created. Then, the next two steps are repeated n times: an existing feature is picked randomly and a new child of this feature is created. The cardinalities of created features are randomly generated according to the defined probabilities for the individual feature cardinalities. Afterwards, inheritance relationships are added. For this purpose, two features are randomly picked and, if the definition of a inheritance relationship is allowed (see condition 6.2 of Definition 6.1.1), it is added. This action is repeated until the defined number of inheritance relationships is reached. The adding of feature links works analogously. Certainly, the type of a feature link is chosen randomly. Finally, feature constraints are generated iteratively until the defined number is reached. The algorithm for the creation of a feature constraint is described in the following. First of all, an empty constraint is created and the probability for the adding of a statement to this constraint is set to 100%. Then, the following actions are iteratively accomplished. A feature is randomly picked and added to the constraint. It is negated by a 50% chance and its cardinality is set, if the configured feature identifier points to an optional feature, to `[1]` or `[+]` (according to the defined probabilities for inclusion and selection statements in constraints). In the case of a cloned feature, an instance name is generated (`I1`, `I2`, ..., `I<max>`, according to the maximal instance count in constraints) and added before the cardinality is set. Then, the constraint length limiter is subtracted from the actual probability for a new statement. With

a probability of the calculated value, another configured feature identifier is added to the constraint. It is connected by conjunction (50% probability) or disjunction (50% probability). The loop always terminates because the probability for a new statement is decreased in every step. Similar algorithms to create feature models have already been proposed in [TBK09, Seg08]. These algorithms can also create feature groups, which are neglected in our generator since they are irrelevant for the propagation.

The generator for configurations gets a feature model as input and returns a generated configuration for this feature model with the defined number of configured feature identifiers. Iteratively, a configured feature identifier pointing to a random feature is generated. If the feature is cloned, an instance (according to the maximal instance count in configurations) is created and included (50% probability) or excluded (50% probability). Otherwise, if the feature is optional, the configured feature identifier is directly included or excluded in the configuration.

The algorithm that generates configuration links iteratively creates configuration decisions until the determined number of configuration decisions is reached. To generate the criterion of a configuration decision, the same algorithm as for the generation of constraints is used (the parameters for the generation of configuration links are listed above). The length limiter for criteria works analogously to the length limiter for constraints. The generation of the effect of the configuration decision bases, in principle, on the same algorithm. Starting with an empty effect, configuration steps are iteratively created and combined with the effect. After every iteration, the probability for adding a further configuration step is decreased by subtracting the value of the length limiter for effects of configuration decisions (as already described for the generation of feature constraints above). For every configuration step, a configured feature identifier is randomly chosen and the cardinality is set according to the determined probabilities for inclusion, exclusion, selection and deselection statements. In the case of cloned features, an instance name is added according to the maximal instance count. An instance creation statement for an individual instance of a cloned feature is added with the defined probability.

7.2.2 Functional Testing of Feature Constraint Propagation

In order to accomplish functional tests to feature constraint propagation, the generator (cf. Section 7.2.1) is used to iteratively create test cases, each consisting of two generated feature models and a generated configuration link between them. Feature constraint propagation is applied to all generated target-side constraints of a test case and the resulting source-side constraints are added to the source feature model. Afterwards, the generator is used to generate a specified number of source configurations (some of them fulfill the propagated constraints, others do not). Finally, the original CVM implementation is used to apply the configuration link to all generated configurations. A test case is passed if every valid source configuration leads to a valid target configuration and, vice versa, every invalid source configuration results also in an invalid target configuration. This is checked automatically by the CVM implementation. The test framework processes the results. It is able to save the models of every or, alternatively, every failed test CVM files. These files can then

be loaded with CVM. They contain the two feature models, the configuration link, the propagated constraints and all checked source and target configurations. So, the propagation can be repeated, the errors can be analyzed and, after the implementation is fixed, the propagation can be applied one more time to ensure that the error is removed. Since the test framework uses the original CVM implementation to apply the configuration link and to check the correctness of configurations, it does not solely test feature constraint propagation but also CVM. In addition, possible implementation errors in the test evaluation are minimized.

We have conducted over two million tests with feature models ranging from 5 to 20 features (including cloned features and inheritance) and configuration links with four to eight configuration decisions. In every test, one target-side feature constraint and one target-side feature link were propagated. Table 7.1 depicts all accomplished tests with the varied parameters. The fixed parameters of the generator are listed in the following itemization.

- length limiter for feature constraints = 30%
- probability for inclusion statements = 50%,
  selection statements = 50%
  in feature constraints and criteria of configuration decisions
- maximal instance count = 3
  in configurations, constraints and configuration decisions
- length limiter for criteria of configuration decisions = 60%
- length limiter for effects of configuration decisions = 60%
- probability for inclusion statements = 25%,
  exclusion statements = 25%,
  selection statements = 25%,
  deselection statements = 25%
  in effects of configuration decisions
- probability for instance creation statements = 50%
  in effects of configuration decisions

A detailed description of the parameters is given in Section 7.2.1. During the first tests, some errors in the implementation were identified and fixed. After the last error found in the implementation was fixed, over 1.8 million tests without any identified error were accomplished. Although software tests cannot ensure the absence of errors but only show their existence, the conducted tests are an indication for the correctness of the implementation.

Interestingly, the parts of feature constraint propagation that had been formalized before (the reverse mappings and the transformations) did not contain any errors. Most of the errors were located in the inclusion-to-selection algorithm. This algorithm was implemented without a well elaborated concept in the beginning. The replacement rules presented in Section 5.1.5 were formulated later. This indicates the significance of a sound formalization before the implementation is conducted.

Number of test cases (checked models)	Features (per feature model)	Probability for optional, mandatory, abstract and cloned features	Extended features (per feature model)	Configuration decisions (per configuration link)	Configurations (per feature model)	Configuration size	Propagator	Mapper
230,000	5	85,5,3,7	1	4	300	1-5	basic propagator	basic reverse mapper
250,000	7	85,5,3,7	1	4	300	1-7	basic propagator	basic reverse mapper
170,000	8	85,5,3,7	1	6	300	3-8	basic propagator	basic reverse mapper
40,000	12	85,5,3,7	1	7	600	7-14	basic propagator	basic reverse mapper
9,000	20	85,5,3,7	1	8	1000	8-20	basic propagator	basic reverse mapper
200,000	5	100,0,0,0	0	4	300	0-5	basic propagator	lazy reverse mapper
200,000	5	85,5,3,7	1	4	300	1-5	minimizing propagator	basic reverse mapper
150,000	9	85,5,3,7	4	6	600	7-12	minimizing propagator	basic reverse mapper
200,000	5	85,5,3,7	1	4	300	1-5	minimizing propagator	lazy minimizing reverse mapper
30,000	6	70,5,5,20	1	4	500	1-5	minimizing propagator	lazy reverse mapper
100,000	6	70,5,5,20	1	4	500	2-6	minimizing propagator	lazy reverse mapper
270,000	5	0,0,0,100	0	5	300	4-12	optimized minimizing propagator	basic reverse mapper
65,000	8	85,5,3,7	1	4	500	6-10	optimized minimizing propagator	basic reverse mapper
15,000	9	85,5,3,7	4	4	600	7-13	optimized minimizing propagator	basic reverse mapper
1,000	5	85,5,3,7	1	4	300	1-5	optimized minimizing propagator	lazy reverse mapper
4,000	7	85,5,3,7	1	4	500	5-10	optimized minimizing propagator	lazy reverse mapper
30,000	7	85,5,3,7	1	5	800	3-12	optimized minimizing propagator	lazy reverse mapper
65,000	10	85,5,3,7	1	8	1000	8-12	optimized minimizing propagator	lazy reverse mapper
27,000	10	85,5,3,7	1	8	1500	7-12	optimized minimizing propagator	lazy reverse mapper
25,000	15	85,5,3,7	1	7	1200	13-18	optimized minimizing propagator	lazy reverse mapper

Table 7.1: Accomplished functional tests of feature constraint propagation

7.2.3 Performance Testing of Feature Constraint Propagation

The runtimes of the individual steps of feature constraint propagation have already been approximated in Section 5.1. We figured out that the minimization is the crucial step of feature constraint propagation (w.r.t. the runtime). Nevertheless, an empirical approach was also applied to the implementation: performance testing. The main reasons for the accomplishment of performance tests are: (1) to compare the individual implemented variants of mappers and propagators presented in Section 7.1.3, (2) to measure the runtime of feature constraint propagation on a PC with up-to-date hardware and (3) to find out how several parameters (e.g. the model size) influence the runtime.

Benchmarking of feature constraint propagation is not trivial. There are a lot of parameters that can be varied.

- feature count of both feature models
- number of cloned features in both models and their instances in the configuration link
- number of inheritance relationships in both models
- number of configuration decisions
- complexity of criteria of configuration decisions (length, nesting, number of included selection statements)
- complexity of effects of configuration decisions (length, number of included selection statements)
- number of constraints to be propagated
- complexity of constraints to be propagated (length, nesting, number of included selection statements)

Every accomplished performance test (short *benchmark*) consists of several *test series*, themselves consisting of 500 tests with several pairs of propagators and mappers. Each of these tests is organized as follows. The generator is used to compute a model (two feature models and a configuration link between them) and a target-side constraint. Then, several pairs of propagators and mappers are used to propagate the target-side constraint along the configuration link. The runtime of every individual propagation (in milliseconds) is logged. So the runtimes of the individual propagators and mappers can directly be compared. After a test series (i.e. 500 of these tests) is completed, the parameters of the generator are changed and a new test series is started.

The results of all benchmarks are depicted as tables (e.g. Table 7.2). These tables show the best and worst runtime of an individual propagation (*Minimum* / *Maximum*) and the average runtime (arithmetic mean) of all 500 propagations (*Average*) per test series. In addition, the standard deviation (the square root of the variance) for every test series and the overall runtimes of all propagations of all test series for every pair of propagator and mapper (the last row) are shown. The best

and the worst runtime of every row is highlighted in color: the best result in green and the worst result in red.

The next listing shows the values of the fixed parameters of the generator for the following benchmarks.

- probability for optional features = 85%,
  mandatory features = 5%,
  abstract features = 3%,
  cloned features = 7%
- number of inheritance relationships = 1
- number of feature links = 1
- number of feature constraints = 0
- maximal instance count in configuration decisions = 3
- probability for inclusion statements = 50%,
  selection statements = 50%
  in criteria of configuration decisions
- length limiter for criteria of configuration decisions = 60%
- length limiter for effects of configuration decisions = 60%
- probability for inclusion statements = 25%,
  exclusion statements = 25%,
  selection statements = 25%,
  deselection statements = 25%
  in effects of configuration decisions
- probability for instance creation statements = 50%
  in effects of configuration decisions

In order to ensure the comparability of all benchmarks, all measurements are accomplished on the same Windows 7 PC with a AMD Phenom II X4 3.4 GHz processor (quad-core CPU) and 12 GB RAM. Note that the used implementation of ESPRESSO does not support multi-core processors and runs therefore only on one core.

Varying the model size (feature count and number of configuration decisions)

The intention of this first benchmark is to compare all implemented variants of propagators and mappers, to get an overview on the runtime of feature constraint propagation and the scalability of the propagation. The benchmark starts with feature models with a feature count of five (per model) and a configuration link between them with one configuration decision. After every test series, the feature count is

Features / Configuration Decisions		BASIC Propagator, BASIC Mapper	BASIC Propagator, LAZY Mapper	BASIC Propagator, LAZY MINIMIZING Mapper	MINIMIZING Propagator, BASIC Mapper	MINIMIZING Propagator, LAZY Mapper	MINIMIZING Propagator, LAZY MINIMIZING Mapper	OPTIMIZED MINIMIZING Propagator, BASIC Mapper	OPTIMIZED MINIMIZING Propagator, LAZY Mapper	OPTIMIZED MINIMIZING Propagator, LAZY MINIMIZING Mapper
5 / 1	Minimum	48	46	88	120	120	161	46	45	65
	Average	**107**	**90**	**181**	**206**	**204**	**295**	**93**	**90**	**153**
	Maximum	354	386	485	604	464	880	426	346	420
	Standard Deviation	*46*	*42*	*56*	*44*	*43*	*65*	*44*	*41*	*58*
10 / 3	Minimum	50	47	83	120	120	170	48	48	68
	Average	**107**	**92**	**204**	**205**	**212**	**320**	**90**	**97**	**169**
	Maximum	548	393	516	419	470	798	202	243	496
	Standard Deviation	*46*	*42*	*72*	*42*	*40*	*71*	*38*	*41*	*76*
15 / 4	Minimum	50	46	90	120	126	161	49	50	70
	Average	**113**	**94**	**216**	**215**	**214**	**341**	**98**	**99**	**187**
	Maximum	401	379	561	587	1,271	3,303	664	262	788
	Standard Deviation	*48*	*44*	*79*	*51*	*75*	*158*	*50*	*42*	*89*
20 / 6	Minimum	50	49	90	127	122	160	49	46	66
	Average	**113**	**102**	**223**	**214**	**213**	**338**	**102**	**98**	**191**
	Maximum	554	663	631	605	594	937	356	476	536
	Standard Deviation	*56*	*59*	*83*	*49*	*49*	*84*	*45*	*48*	*88*
25 / 7	Minimum	46	46	78	124	124	157	46	46	62
	Average	**122**	**108**	**224**	**214**	**215**	**337**	**103**	**107**	**189**
	Maximum	1,779	1,718	2,233	969	906	2,623	1,638	1,593	2,408
	Standard Deviation	*122*	*132*	*122*	*67*	*68*	*133*	*92*	*94*	*134*
30 / 9	Minimum	46	46	78	124	124	157	46	46	62
	Average	**452**	**454**	**443**	**374**	**384**	**553**	**183**	**191**	**404**
	Maximum	79,312	80,122	101,748	38,611	39,907	101,032	29,079	31,841	100,974
	Standard Deviation	*4,435*	*4,593*	*4,547*	*2,153*	*2,279*	*4,510*	*1,311*	*1,436*	*4,514*
35 / 10	Minimum	46	46	78	109	109	140	46	46	62
	Average	**2,352**	**2,340**	**7,067**	**1,393**	**1,401**	**6,658**	**2,015**	**2,026**	**5,083**
	Maximum	790,901	788,246	2,822,790	395,568	397,951	2,560,910	793,735	793,698	2,400,588
	Standard Deviation	*36,841*	*36,688*	*128,821*	*18,442*	*18,548*	*117,275*	*35,932*	*35,933*	*107,675*
40 / 12	Minimum	46	46	78	109	109	140	46	46	62
	Average	**8,868**	**9,099**	**5,344**	**4,618**	**4,758**	**5,316**	**4,464**	**4,578**	**4,503**
	Maximum	2,590,410	2,621,673	1,090,221	1,300,000	1,313,728	1,012,558	1,304,610	1,313,822	1,007,005
	Standard Deviation	*118,954*	*120,626*	*56,355*	*59,690*	*60,449*	*53,365*	*59,918*	*60,478*	*52,158*
45 / 13	Minimum	46	46	78	109	109	140	46	46	62
	Average	**5,361**	**5,529**	**826**	**2,790**	**2,924**	**936**	**2,495**	**2,620**	**553**
	Maximum	964,381	940,937	116,070	465,618	468,299	114,134	466,786	467,206	117,676
	Standard Deviation	*57,446*	*57,573*	*6,626*	*28,081*	*28,745*	*6,556*	*28,047*	*28,777*	*5,414*
50 / 15	Minimum	46	46	78	109	109	140	46	46	62
	Average	**16,033**	**16,751**	**20,286**	**23,858**	**24,346**	**31,586**	**10,857**	**11,295**	**18,418**
	Maximum	3,048,651	3,044,051	6,050,767	7,314,242	7,334,222	10,336,131	2,906,122	2,912,409	5,854,821
	Standard Deviation	*156,260*	*157,914*	*286,831*	*340,768*	*342,033*	*483,355*	*137,344*	*138,267*	*272,970*
Overall Runtime		*16,508,009*	*17,010,647*	*17,160,129*	*16,649,853*	*17,034,908*	*22,824,463*	*10,053,224*	*10,395,103*	*14,616,492*

green = lowest value in row red = highest value in row

Table 7.2: Benchmark results (all values in milliseconds): correlation between runtime and model size

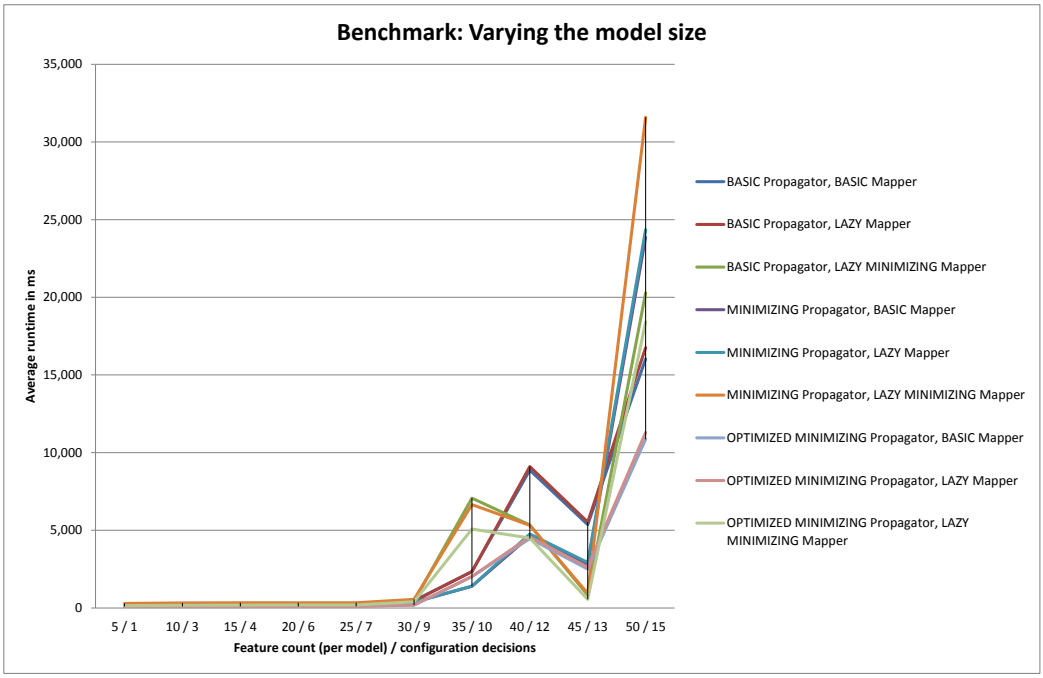


Figure 7.6: Average runtimes of Table 7.2 as diagram

increased by five and the number of configuration decisions is increased relatively to the feature count (30% of the feature count, rounded). All nine combinations of propagators and mappers are included in this benchmark.

The results of the benchmark, which consists of ten test series, are depicted in Table 7.2. The average runtimes of all test series are also depicted as a diagram in Figure 7.6. We draw the following conclusions.

The optimized minimizing propagator is the fastest propagator. The optimized minimizing propagator with the basic mapper achieved the best overall runtime (the last line in the table) of about 167.5 minutes (10.05 million milliseconds) for all 5,000 propagations. The second and the third place in the category "overall runtime" also go to the optimized minimizing propagator with the other mappers. The result that the optimized minimizing propagator is the fastest propagator was expected. The idea of this propagator was runtime optimization by propagating the individual constraints in a determined order and to abort the propagation as soon as possible (cf. Section 7.1.3). Besides the best overall runtime, the optimized minimizing propagator (with all three mappers) also reached most best runtimes of the individual tests (green cells). In addition, it is mentionable that the optimized minimizing propagator never achieved the highest runtime (red cells) in comparison to the other propagators. However, this propagator is not always faster than the other propagators. The best runtimes of some test series were reached by other propagators.

The basic and the lazy reverse mapper have similar runtimes but the lazy minimizing reverse mapper is slower. We expected the lazy reverse

mapper to be faster than the basic reverse mapper because it only calculates the necessary mappings. Interestingly, the basic reverse mapper was the fastest mapper in the category "overall runtime" (with all three propagators). We explain this result by the low number of configuration decisions. The calculation of the complete reverse mapping is therefore very fast. It is accomplished in one method. In the case of the lazy calculation, there are some additional accesses to the data structure to check whether a mapping has already been calculated or not. If a mapping of a feature has not already been calculated, a method to calculate this single mapping is called, in which some help variables are declared and instantiated. All these activities need some time – and they are accomplished for every variable to be mapped. However, the differences between the basic and the lazy reverse mapper are quite small. The worst mapper, regarding the overall runtime, is, as expected, the lazy minimizing reverse mapper (with all three propagators). In the case of the minimizing and the optimized minimizing propagator, the overall runtime of the propagation with this mapper is drastically higher than the one with the other mappers (over 96 minutes for the minimizing propagator[2] and over 70 minutes for the optimized minimizing propagator[3]). The combination of the lazy minimizing reverse mapper with the minimizing propagator produced by far most highest runtimes (red cells). This is because the minimization algorithm is used much more often during a propagation, especially if selection statements are mapped, than during a propagation with a different mapper. Note that the application of the minimization algorithm to an already minimized formula does not necessarily have to be faster than its application to a non-minimized representation of this formula. This leads to a higher runtime of the propagation with the lazy minimized reverse mapper – especially in combination with a propagator that also applies the minimization algorithm to intermediate constraints. By the way, the intention of this mapper was not the optimization of the runtime but a better visualization of the individual mappings (cf. Section 7.1.3).

The minimizing propagator is slower than the other propagators. Looking at the partitioning of the colored cells in the whole table, it becomes apparent at first sight that the minimizing propagator (with all three mappers) reached only in a few tests the best runtime (green cells). However, in many tests it achieved the highest runtime (red cells). This shows that the minimizing propagator is usually slower than the other propagators. However, in some tests (e.g. the test series with 35 features) the minimizing propagator (with the basic as well as the lazy reverse mapper) was much faster than the other propagators and mappers. One possible explanation for this is that the minimization of the intermediate results lead to a propagated formula that can be minimized much faster. The ESPRESSO algorithm contains a lot of optimizations to speed up the minimization. To analyze this behavior, the individual propagation steps, the structures of the resulting formulae and the ESPRESSO algorithm have to be investigated in detail.

The minimal runtimes of the accomplished propagations are independent of the model size. Let us take a closer look at the individual runtimes. The minimal runtimes of propagations are, for all model sizes, almost the same. This is because minimal runtimes appear if the propagated constraints are very short or

[2] $22,824,463 \text{ ms} - 17,034,908 \text{ ms} = 5,789,555 \text{ ms} \approx 96 \text{ min}$

[3] $14,616,492 \text{ ms} - 10,395,103 \text{ ms} = 4,221,389 \text{ ms} \approx 70 \text{ min}$

even trivial (`true` or `false`), e.g. if the start feature of a `needs` feature link cannot get selected through the application of the configuration link. The minimal runtime of propagations increases with the count of minimizations. This is because every minimization starts ESPRESSO in a new process and waits for it to terminate, which takes some time – no matter if the constraint is trivial or very complex. Minimal runtimes of the individual combinations of propagators and mappers are listed in the following.

- basic and optimized minimizing propagator with basic or lazy reverse mapper: 45 to 50 milliseconds
- optimized minimizing propagator with lazy minimizing reverse mapper: 62 to 70 milliseconds
- basic propagator with lazy minimizing reverse mapper: 78 to 90 milliseconds
- minimizing propagator with all three reverse mappers: over 100 milliseconds

Note that the runtime of the propagation is in general also dependent on the number of configuration decisions since some iterations over all configuration decisions of the configuration link are conducted (cf. Section 5.1.2). However, the runtime of this step is negligible in comparison to the overall runtime (cf. Section 5.1) – especially if the number of configuration decisions is low.

The average and maximal runtimes increase with the model size. Theoretically, a larger model leads to higher average and maximal runtimes. However, the diagrammatic representation (Figure 7.6) shows that the average runtimes are not monotonically increasing. This also holds for the maximal runtimes and is discussed later. Nevertheless, increasing average and maximal runtimes with the increasing model size are obvious. In the case of the optimized minimizing propagator with the basic reverse mapper, the average runtime starts at 93 milliseconds (for five features and one configuration decision) and increases to 183 milliseconds (for 30 features and 9 configuration decisions). Then it drastically increases to over ten seconds (for 50 features and 15 configuration links). This drastic increase of runtime for larger models (here for models with more than 30 features and 9 configuration links) is similar for the other propagators and mappers.

The runtimes are highly dependent on the model structure. We have already mentioned that the average runtimes are not monotonically increasing, as the diagram (Figure 7.6) shows. The peaks result from single propagations with a very long runtime. This can also be seen in the table: the maximal runtimes of all model sizes are drastically higher than the average runtimes. This is illustrated by the standard deviation (for all propagators and mappers). This value increases with the model size.

- for models with 25 or less features and 7 or less configuration decisions, the standard deviation is under 200 milliseconds

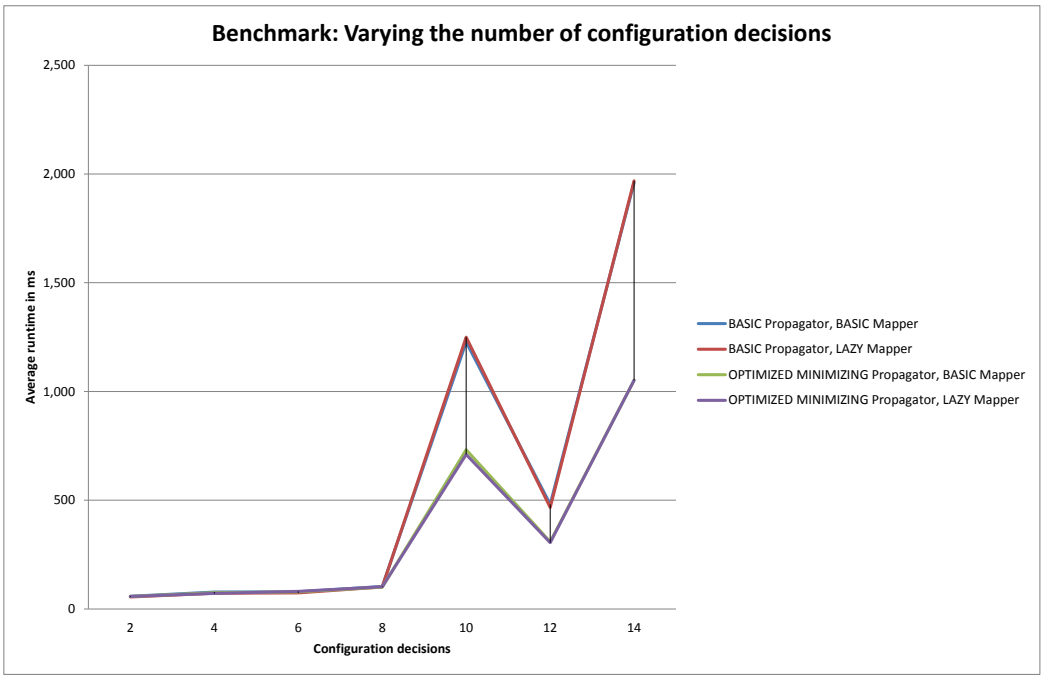


Figure 7.7: Average runtimes of Table 7.3 as diagram

- for models with 30 features and 9 configuration decisions, the standard deviation drastically increases to over one second
- for models with 50 features and 15 configuration decisions, the standard deviation is higher than 137 seconds (a very high value for an average runtime of about ten seconds)

This shows that the runtime of a propagation is not only dependent on the model size but mainly dependent on the model structure. However, larger models produce a much higher runtime variation and a much higher maximal runtime. The increase of these values is not monotonic. The runtimes of the propagations vary so extreme that the propagation with the highest runtime took about one third of the overall runtime of all $5,000$ accomplished propagations for the optimized minimizing propagator. In the case of the other propagators, this value is comparably high. We analyzed the models that produced the maximal runtimes and figured out that in all cases the minimization process is the crucial part of the process. This result was expected because of the results of the runtime approximation of every propagation step (cf. Section 5.1).

Varying the number of configuration decisions

In this benchmark the number of configuration decisions is increased while the number of features remains unchanged at ten. The intention is to investigate the dependency between the runtime and the number of configuration decisions. The

Features / Configuration Decisions		BASIC Propagator, BASIC Mapper	BASIC Propagator, LAZY Mapper	OPTIMIZED MINIMIZING Propagator, BASIC Mapper	OPTIMIZED MINIMIZING Propagator, LAZY Mapper
10 / 2	Minimum	46	46	46	46
	Average	**58**	**55**	**57**	**57**
	Maximum	267	157	156	156
	Standard Deviation	*16*	*11*	*11*	*14*
10 / 4	Minimum	46	46	46	46
	Average	**78**	**73**	**75**	**71**
	Maximum	250	219	173	173
	Standard Deviation	*38*	*33*	*35*	*31*
10 / 6	Minimum	46	46	46	46
	Average	**79**	**74**	**79**	**81**
	Maximum	282	188	296	282
	Standard Deviation	*37*	*33*	*34*	*39*
10 / 8	Minimum	46	46	46	46
	Average	**104**	**102**	**100**	**102**
	Maximum	5,758	6,631	3,107	3,355
	Standard Deviation	*316*	*353*	*164*	*185*
10 / 10	Minimum	46	46	46	46
	Average	**1,227**	**1,249**	**732**	**710**
	Maximum	568,280	580,734	318,172	308,001
	Standard Deviation	*25,410*	*25,967*	*14,225*	*13,770*
10 / 12	Minimum	46	46	46	46
	Average	**481**	**467**	**309**	**305**
	Maximum	164,147	159,404	80,217	77,535
	Standard Deviation	*7,397*	*7,182*	*3,708*	*3,599*
10 / 14	Minimum	46	46	47	47
	Average	**1,960**	**1,969**	**1,051**	**1,052**
	Maximum	878,643	878,724	447,320	446,625
	Standard Deviation	*39,333*	*39,338*	*20,023*	*19,993*
Overall Runtime		*1,991,380*	*1,992,155*	*1,200,796*	*1,187,800*

green = lowest value in row red = highest value in row

Table 7.3: Benchmark results (all values in milliseconds): correlation between runtime and the number of configuration decisions

minimizing propagator and the lazy minimizing reverse mapper are not included in this test since the first benchmark showed that they are generally slower than the other propagators and mappers. Table 7.3 shows the results of this benchmark. The average runtimes of all test series are also depicted as a diagram in Figure 7.7. The test results are similar to the results of the previous benchmark. We draw the following conclusions.

The minimal runtimes of the accomplished propagations are independent of the number of configuration decisions. We have already discussed above that the number of configuration decisions affects the minimal runtimes since some iterations over all configuration decisions are conducted during the propagation. However, at this low number of configuration decisions all minimal runtimes are almost the same.

The average and maximal runtimes increase with the number of configuration decisions. Although the source and the target feature model only consist of ten features each, the average and maximal runtimes of the propagations increase very strong with the number of configuration decisions. Again, the diagram (Figure 7.7) points out that the increase (of the average runtimes) is not monotonic.

The following conclusions are already known from the previous benchmark but are confirmed by this benchmark.

The runtimes are highly dependent on the model structure. The maximal runtime of a propagation along a configuration link with 14 configuration decisions is higher than seven minutes (420,000 ms) for the optimized minimizing propagator and higher than 14 minutes (840,000 ms) for the basic propagator, although the average runtime for the same count of configuration decisions is one to two seconds (depending on the used propagator). As expected, the reason for this long runtime was the minimization of a complex formula. More configuration decisions can lead to more complex formulae (at a constant complexity of criteria of configuration decisions) and this can slow down the minimization process.

The optimized minimizing propagator is faster than the basic propagator. This has already been discussed above.

The basic and the lazy reverse mapper have similar runtimes. This has also already been discussed above.

Varying the feature count

In this third benchmark the dependency of the feature count and the runtimes of feature constraint propagation is analyzed experimentally. Therefore, the feature count of the models is increased in every iteration while the number of configuration decisions remains unchanged at ten. Table 7.4 shows the results of the benchmark. Again, the average runtimes of all propagations are also depicted as a diagram in Figure 7.8 and in Figure 7.9 with a different y-axis scaling. We draw the following conclusions.

The minimal runtimes of the accomplished propagations are independent of the feature count. This result is not surprising and has already been discussed above.

Features / Configuration Decisions		BASIC Propagator, BASIC Mapper	BASIC Propagator, LAZY Mapper	OPTIMIZED MINIMIZING Propagator, BASIC Mapper	OPTIMIZED MINIMIZING Propagator, LAZY Mapper
5 / 10	Minimum	50	50	50	50
	Average	**64**	**64**	**75**	**78**
	Maximum	381	521	300	690
	Standard Deviation	*30*	*36*	*24*	*48*
10 / 10	Minimum	50	45	48	50
	Average	**99**	**163**	**104**	**108**
	Maximum	3,012	31,354	2,931	1,581
	Standard Deviation	*166*	*1,408*	*151*	*123*
15 / 10	Minimum	47	49	46	47
	Average	**122**	**118**	**114**	**111**
	Maximum	5,943	6,194	2,871	3,164
	Standard Deviation	*293*	*308*	*155*	*166*
20 / 10	Minimum	50	50	50	50
	Average	**16,570**	**16,402**	**16,008**	**16,002**
	Maximum	7,860,983	7,758,187	7,744,196	7,750,850
	Standard Deviation	*351,819*	*347,257*	*346,410*	*346,694*
25 / 10	Minimum	50	50	50	50
	Average	**92,637**	**91,896**	**102,881**	**99,254**
	Maximum	46,199,724	45,827,833	51,359,500	49,544,828
	Standard Deviation	*2,066,104*	*2,049,472*	*2,296,859*	*2,215,705*
30 / 10	Minimum	50	50	50	50
	Average	**2,190**	**2,228**	**1,779**	**1,793**
	Maximum	724,370	725,882	724,667	725,481
	Standard Deviation	*33,269*	*33,402*	*32,568*	*32,622*
35 / 10	Minimum	50	50	48	50
	Average	**3,199**	**2,922**	**1,446**	**1,462**
	Maximum	1,437,753	1,295,304	646,209	648,861
	Standard Deviation	*64,319*	*57,957*	*28,904*	*29,024*
Overall Runtime		*57,440,464*	*56,896,148*	*61,202,772*	*59,404,489*
		green = lowest value in row		red = highest value in row	

Table 7.4: Benchmark results (all values in milliseconds): correlation between runtime and feature count

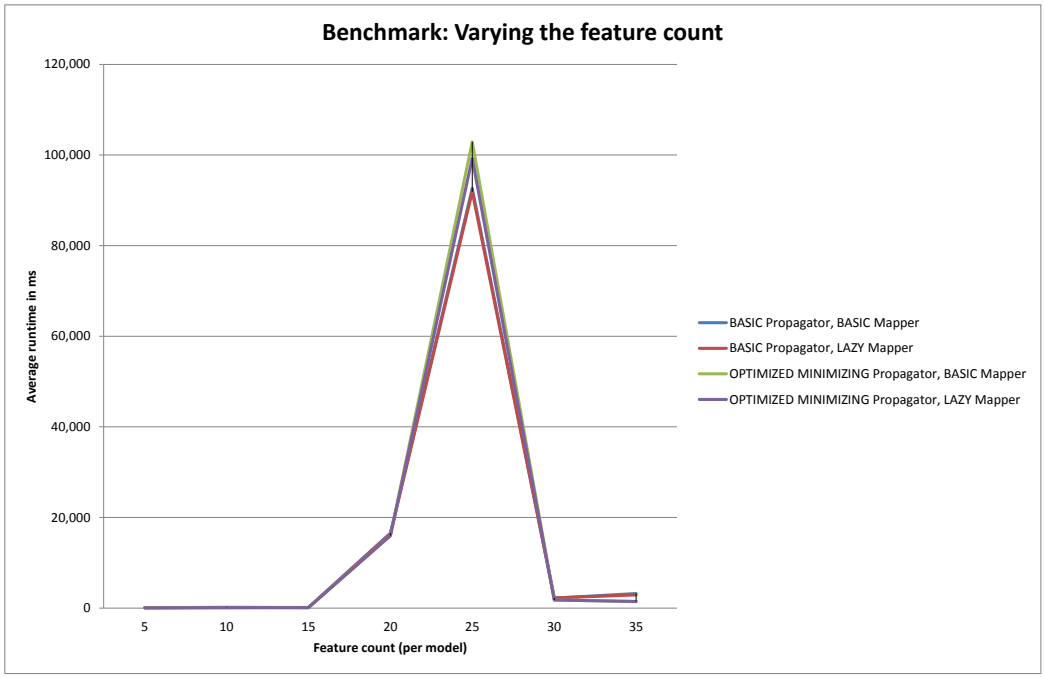


Figure 7.8: Average runtimes of Table 7.4 as diagram

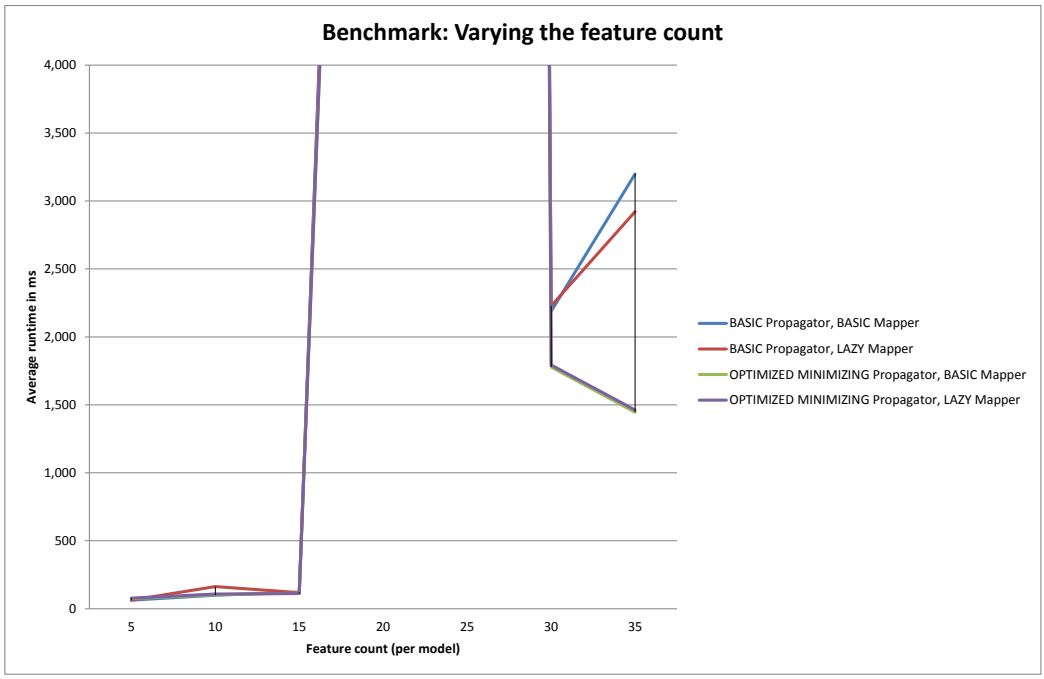


Figure 7.9: Average runtimes of Table 7.4 as diagram (alternative y-axis scaling)

The average and maximal runtimes increase with the feature count. Besides the huge peak at 25 features in the diagram, which is discussed later, the average and maximal runtimes increase with the feature count. However, since this is not obvious in Figure 7.8, the same diagram with a different y-axis scaling is also depicted in Figure 7.9 in order to visualize the increase of the average runtimes with the feature count.

The optimized minimizing propagator is in most cases – but not always – faster than the basic propagator. We mentioned that the optimized minimizing propagator is the fastest propagator. This benchmark shows that this is not always the case. The average runtimes of four of seven test series and the maximal runtimes of five of seven test series of the optimized minimizing propagator are lower than the corresponding runtimes of the basic propagator (both mappers are taken into account). The lowest runtime of every test series is colored green in the table. Although the optimized minimizing propagator is faster in most test series, the overall runtime of the basic propagator is in this benchmark lower than the overall runtime of the optimized minimizing propagator. Interestingly, the reason for this higher overall runtime of the optimized minimizing propagator is only the above-mentioned single model with a runtime of over 12 hours (43, 200, 000 ms). The optimized minimizing propagator needed over an hour (3, 600, 000 ms) longer to process this model than the basic propagator (this value varies dependent on the used mapper)[4]. The reason for this is, one more time, the minimization. In the case of the optimized minimizing propagator, more minimizations are sometimes applied (if the propagation cannot be aborted) than in the case of the basic propagator, as mentioned in Section 7.1.3. This leads to higher runtimes for the propagation in some cases.

The following conclusions are already known from the previous benchmarks but are confirmed by this benchmark.

The runtimes are highly dependent on the model structure. The peak at 25 features in the diagram with the average runtimes (Figure 7.8) results from a single propagation that took over 12 hours (43, 200, 000 ms). It is mentionable that this single propagation took over 80% of the overall runtime (for all propagators and mappers) of all 3, 500 accomplished propagations. This shows that the variation of the runtimes is extreme. The standard deviation is greater than 33 minutes (1, 980, 000 ms) for models with 25 features, whereas the average runtime is under two minutes (120, 000 ms). This holds for all propagators and mappers. The analysis of the model with the highest runtime showed that the transformation of the target-side constraints lead to a very complex source formulae with a lot of variables, such that the minimization took a lot of time. This shows one more time that the minimization is the crucial part of feature constraint propagation. Larger feature models can – but do not have to – lead to more variables in the resulting formulae and, consequently, to a slower minimization. Especially selection statements in criteria can produce long formulae through the expansion of the configuration link in the first step of the propagation (cf. Section 5.1.1).

[4] $51,359,500 \text{ ms} - 46,199,724 \text{ ms} = 5,159,776 \text{ ms} \approx 86 \text{ min}$ with the basic reverse mapper and $49,544,828 \text{ ms} - 45,827,833 \text{ ms} = 3,716,995 \text{ ms} \approx 62 \text{ min}$ with the lazy reverse mapper

The basic and the lazy reverse mapper have similar runtimes. This has already been discussed above.

Summary of Conclusions

Although all conclusions have already been mentioned and discussed, most important facts are summarized in the following listing. A repeated discussion is not given here.

- the crucial part of feature constraint propagation (w.r.t. the runtime) is the minimization
- the standard deviation of the runtime increases drastically with the model size
- the minimal runtime of feature constraint propagation is (almost) independent of the model size
- the average runtime and the maximal runtime of feature constraint propagation increase with the model size
- runtimes of the propagators in most cases:
  optimized minimizing propagator $<$ basic propagator $<$ minimizing propagator
- the reverse mappers have only a relatively small influence on the overall runtime of feature constraint propagation
- the basic and the lazy reverse mapper have similar runtimes but the lazy minimizing reverse mapper is in most cases much slower

Discussion

Above, in the beginning of Section 7.2.3, we stated that benchmarking of feature constraint propagation is a challenging task and presented a list of parameters that could affect the runtime of the propagation. The accomplished benchmarks took only some of these parameters into account: the feature count and the number of configuration decisions. We decided to omit further benchmarks since we identified the logical minimization as crucial part of the technique. The variation of the other parameters surely also affects the runtime of feature constraint propagation. Most of these parameters even have a direct impact on the formulae to be minimized.

- more selection statements (in criteria of configuration links) can lead, through the expansion step, to formulae with more literals
- a higher instance count in (in effects of configuration links) can lead to more complex formulae when feature links are propagated (because all instances of the features have to be considered)
- a lower length limiter for criteria of configuration links leads to longer criteria and, therefore, to possibly more complex formulae

- a lower length limiter for effects of configuration links leads to longer effects and can therefore lead to the occurrence of every configured feature identifier in multiple effects and this can result in more complex formulae

- a higher probability for selection statements in effects of configuration decisions can imply the same effect

Note that longer formulae do not necessarily lead to a higher runtime of the minimization. However, the structure of a formula and the number of literals have a drastic influence on the runtime of the minimization. ESPRESSO uses a lot of optimizations to speed up the minimization. Even small modifications of a model can lead to extreme differences of the runtime. Hence, all above-mentioned parameters should have similar impacts on the runtime of feature constraint propagation. The standard deviation grows very strong with the maximal complexity of the formulae to be minimized.

Every test series consists of 500 tests. In every test, a model (two feature models and a configuration link between them) is generated and the propagation is accomplished with all propagators and mappers. Especially if large models are generated, there are countless possibilities to organize the features, define the configuration link and to specify constraints. We think that, if more tests per test series are accomplished, the peaks (local maximums) in the diagrams can be reduced and the curve approximates to monotonicity. However, complete elimination of the peaks is most likely not achievable with random generation of models, even if the number of tests is very high.

7.3 Industrial Case Study: Daimler Tuner

During the work on this thesis, we presented our research to some employees of Daimler AG [Dai12]. They were very interested in configuration links and the technique of feature constraint propagation in order to express several kinds of variability in different models and to relate them. We, in turn, were interested in an industrial case study for feature constraint propagation. A cooperation was born. They provided us access to a variability specification and we used our approach to restructure and analyze the information.

This section describes the industrial case study, which was conducted in cooperation with Daimler AG. Within the case study the technique of feature constraint propagation is applied to a realistic model. The main goals of the study are, on the one hand, showing the applicability of feature constraint propagation and, on the other hand, illustrating the usefulness of feature constraint propagation when dealing with configuration links in practice. The case study does not aim to fortify the correctness of the implementation.

Note that all models of the case study are property of Daimler AG and cannot be depicted or described in detail in this thesis. For this reason, concrete features or model parts are not mentioned.

7.3.1 Background and Current State

Point of origin of the case study is an existing feature model of Daimler's audio and video tuner. The tuner is part of Daimler's infotainment system, called **COMAND** (Cockpit Management and Data System), and is included in a lot of different Mercedes-Benz cars. Depending on the technical equipment of a car, its tuner has to provide different functionalities. For example, if the infotainment system shall offer the possibility to watch TV, the tuner has to provide the functionality to interpret video signals. So the tuner can be seen as a product line. The feature model of the tuner was created by the *Requirements Management* team at Daimler AG and describes all commonalities and variabilities of this product line – including country specific audio / video standards and broadcast services, different conditional access systems and numerous system and user functions. It contains many dependencies between features that have to be fulfilled by configurations (i.e. concrete tuners). For example, some digital television systems need a special conditional access method. Features of the tuner model describe system and user functions as well as hardware architecture.

The existing tuner feature model was created with the common product line engineering and feature modeling tool pure::variants of pure-systems [Pur12]. It contains 122 features, 101 relations (feature links are called relations in pure::variants) and 7 (feature) constraints. The maximum depth of the feature model is four and it has a variation count of $4.66 * 10^{28}$ (calculated by pure::variants; relations and constraints are ignored). The model does neither contain cloned or parameterized features nor feature inheritance.

7.3.2 Objectives

Within the case study, features describing the hardware architecture of the tuner model (in the following simply called *architectural features*) shall be separated from features describing system and user functions (in the following simply called *functional features*). This leads to a separate *architecture model* and a *functional model*, as depicted in Figure 7.10. The extraction of the architectural features is not trivial because there are a lot of dependencies between them and functional features. The concept of configuration links shall be used to connect both models and ensure the compliance of all dependencies. This means that every hardware architecture (i.e. source configuration) leads, through the application of the configuration link, to a preconfiguration of system and user functions (i.e. target configuration) that respects all constraints between architectural features and functional features of the original model (i.e. all for the architecture required system and user functions are selected and all forbidden functions are deselected). Figure 7.10 illustrates this idea: the architectural feature `ArchF3` and the functional feature `FuncF2` may not be selected both in the original model. Therefore, the new configuration link contains a configuration decision that deselects `FuncF2` in the functional model if the architectural feature `ArchF3` is selected. We say that the configuration link *ensures* the constraint «`ArchF3 excludes FuncF2`». Naturally, resulting configurations of the functional model may be manually refined in the sense that unconfigured features can be selected or deselected. However, features that are configured by the config-

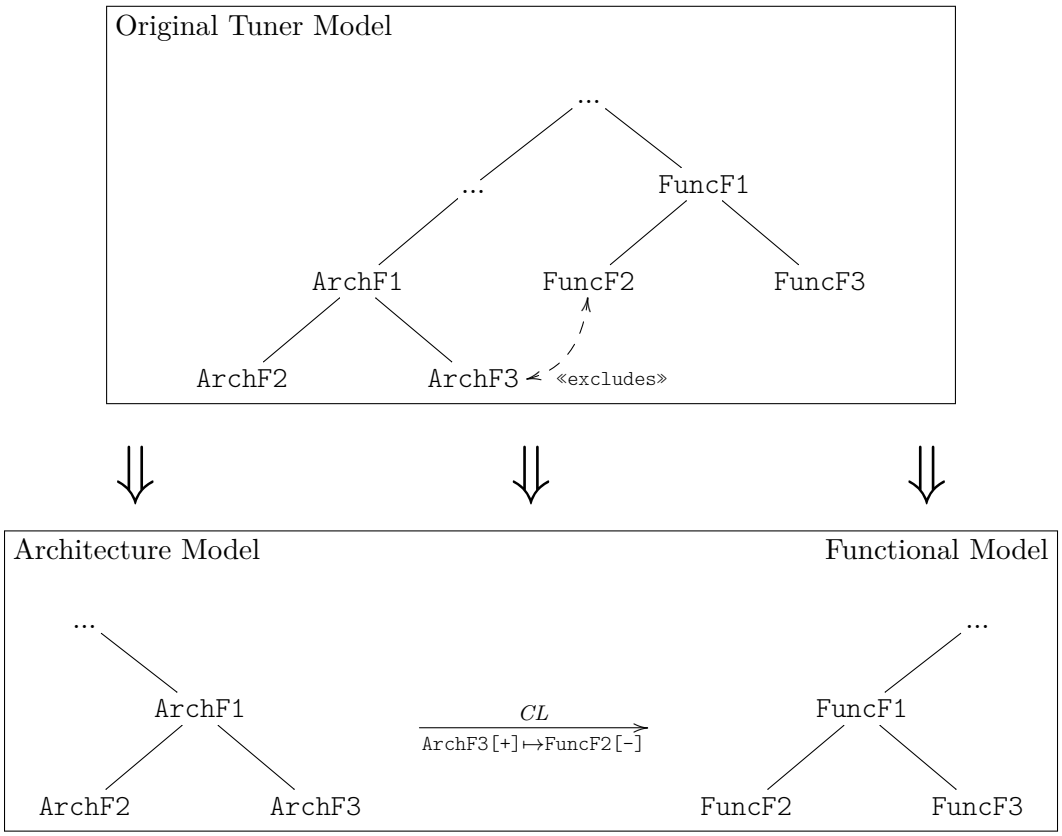


Figure 7.10: Extraction of an architecture model

uration link may not be changed. After all architectural features are extracted and the configuration link is defined, the technique of feature constraint propagation shall be applied to identify derived constraints in the architecture model. These derived constraints shall be analyzed to expose and fix errors in the new variability specification.

Although the use of the architecture model as target model seems to be more intuitive, its use as source model and the definition of the configuration link ensuring the given constraints between both models were motivated by practice: a given hardware architecture of a car allows only a subset of available functionalities. Note that this case study does not aim to show a typical application area for configuration links (such case studies were conducted in [Rei08]) but to evaluate the technique of feature constraint propagation in practice.

7.3.3 Transfer of the Pure Variants Model to CVM

The first step in conducting the case study was the transfer of the pure::variants feature model to CVM. This step was straightforward because both tools are similar. Nevertheless, there are some differences between relations in pure::variants and feature links in CVM. Relations in pure::variants can have more than one end feature, whereas feature links in CVM always have exactly one end feature (cf. Section 7.1.1).

Furthermore, there are some other relation types in pure::variants than feature link types in CVM. Another, admittedly technical, difference between pure::variants and CVM is that both tools use a different syntax for feature constraints.

Table 7.5 shows the differences of constraints in pure::variants and CVM. The left column shows the constraints in the intuitive pure::variants syntax and the right column shows the equivalent feature links or feature constraints in VSL (cf. Section A.1). This table is not complete but contains all constraints that were used within the case study. Note that this table is only valid if cloned features and inheritance are not used. This is because the sets of features on feature level and on instance level are isomorphic in this case.

Lines 1 to 4 of Table 7.5 illustrate differences between the syntax of feature constraints in pure::variants and CVM. Allowed logical connectives in CVM (resp. VSL) are `!` for negation ($\neg$), `&` for conjunction ($\wedge$) and `|` for disjunction ($\vee$). These three logical connectives are functionally complete. `Requires` relations are the counterpart to `needs` feature links. They state that the existence of the start feature implies the existence of at least one end feature. The `RequiresAll` variant implies the existence of all end features. The relations in lines 5, 7 and 8 are simply expressed by `needs` feature links in CVM. The case that a feature requires at least one feature of a given set (see line 6) has to be formulated as feature constraint in CVM: either `A` is not selected or at least one feature of `B`, `C`, ... is selected. `Conflicts` relations correspond to `excludes` feature links (lines 9 to 12). Analogous to `Requires` relations, there are two variants: `Conflicts` and `ConflictsAny`. The former relation states that not all features can be selected together. The last-mentioned relation states that the start feature cannot be selected together with any end feature. If `Conflict` or `ConflictsAny` relations have exactly one end feature, they are translated into equivalent feature links in CVM (lines 9 and 11). Otherwise, they are translated into equivalent feature constraints (lines 10 and 12). Finally, there are `Provides` relations, as depicted in lines 13 to 15. They correspond to "inverse" `needs` feature links. If a `Provides` relation has more than one end feature, it is translated into a feature constraint. Line 15 presents a special case: if there are multiple `Provides` relations to a feature, these relations are not connected via conjunctions but via disjunctions. This means, in line 15, if feature `C` is selected, at least one feature of `A`, `B`, ... has to be selected – not necessarily all. The idea behind this semantics is that a `Provides` relation is the "inverse" relation to `Requires` (cf. line 6). Because of this semantics, multiple `Provides` relations to one feature have to be formulated as one constraint in CVM and cannot be modeled separately.

The resulting model in CVM contains 122 features, 56 feature links and 28 feature constraints. It is equivalent to the initial pure::variants feature model. Admittedly we did not prove this equivalence, but it follows from the equivalence of the individual constraints, which can be comprehended by Table 7.5.

7.3.4 Extracting the Architecture Model

The next step of the case study was the extraction of an architecture model. All architectural features were identified in the tuner model and transferred into a separate feature model, which leads to an architecture model and a functional model (cf. Fig-

№	**pure::variants**	**CVM**
	Constraint	Feature Constraint
1	`NOT A`	`!A[+]`
2	`A AND B`	`A[+] & B[+]`
3	`A OR B`	`A[+] | B[+]`
4	`A requires B`	`!A[+] | B[+]`
	Relation(s)	Feature Link(s) / Constraint
5	`A Requires B`	`«A needs B»`
6	`A Requires B, C, ...`	`!A[+] | (B[+] | C[+] | ...)`
7	`A RequiresAll B`	`«A needs B»`

8	`A RequiresAll B, C, ...`	`«A needs B»,`
		`«A needs C»,`
		`«A needs ...»`

9	`A Conflicts B`	`«A excludes B»`
10	`A Conflicts B, C, ...`	`!(A[+] & B[+] & C[+] & ...)`
11	`A ConflictsAny B`	`«A excludes B»`

12	`A ConflictsAny B, C, ...`	`!(A[+] & B[+])`
		`& !(A[+] & C[+])`
		`& !(A[+] & ...)`

13	`A Provides B`	`«B needs A»`
14	`A Provides B, C, ...`	`!(B[+] | C[+] | ...) | A[+]`
15	`A Provides C, B Provides C, ...` `Provides C`	`!C[+] | (A[+] | B[+] | ...)`

Table 7.5: Transfer of constraints from pure::variants to CVM

ure 7.10). At the beginning of the case study, there has already been a premature architecture model with some additional features that was completed within this step. Dependencies (in form of feature links) between architectural features were also transferred into the architecture model. Dependencies between both models have been deliberately omitted so far and were considered later on.

Next, a configuration link from the architecture model to the functional model was defined. Configuration decisions ensuring the satisfaction of dependencies between architectural features and functional features were formulated and added to the configuration link. This means that all dependencies of the original model between features that are now in the same model are defined as feature links or feature constraints. In contrast, all dependencies of the original model between features that are now in different models are ensured by configuration decisions. Table 7.6 shows how relations were translated into configuration decisions. Assume that features to the left of relation names are contained in the source model (i.e. the architecture

model) and features to the right of relation names are contained in the target model (i.e. the functional model). This table is not complete but contains only cases that appeared within the case study.

`Requires` relations are logical implications. This means, in lines 1, 3 and 4 of Table 7.6, the selection of `A` has the effect that `B` gets (resp. `B`, `C` and ... get) selected by the configuration link. The definition of a configuration decision that ensures the `Requires` relation of line 2 is not possible since the selection of `A` implies only the selection of (at least) one feature of the right hand side – and not the selection of all these features. The effect of the corresponding configuration decision would be nondeterministic and, therefore, not well-formed. Indeed, selecting one arbitrary feature of the right hand side would ensure the given constraint, however, this would actually ensure a stronger constraint, i.e. not every valid configuration of the original model could be created. Therefore, this idea is not feasible. The condition that two or more features are in conflict is expressed by `Conflicts` relations. If a feature `A` is selected, then `B` gets (resp. `B`, `C` and ... get) deselected by the configuration link, as defined in lines 5, 7 and 8. The case of line 6 produces a similar problem to the case of line 2. If feature `A` is selected, it is not clear which features (`B`, `C` or ...) shall be deselected. In order to ensure `Provides` relations, contrapositions of implications are used. This means, in lines 9 and 10, if feature `A` is not selected, then feature `B` gets (resp. `B`, `C` and ... get) deselected by the configuration link. Line 11 expresses the special case for multiple `Provides` relations, which has already been described in Section 7.3.3 (cf. line 15 of Table 7.5). If all features `A`, `B` and ... are not selected, then `C` gets deselected by the configuration link.

Note that all occurring contradictions between configuration decisions have to be handled particularly in order to ensure satisfaction of all constraints when translating constraints into a configuration link according to Table 7.6. This is because an exclusion has precedence over an inclusion in configuration links. Consider the following small example: given relations `A Requires B` and `C Conflicts B`. According to Table 7.6, these constraints lead to a configuration link with configuration decisions `A[+]` $\mapsto$ `B[+]` and `C[+]` $\mapsto$ `B[-]`. Since precedence of exclude over include, the selection of `A` and `C` leads to the deselection of `B` and this violates the original constraint `A Requires B`. To solve this particular problem, a feature link «`A excludes C`» can be added. This additional feature link can be derived by investigating contradictions between resulting configuration decisions. Note that feature constraint propagation can be used to analyze under which conditions a target-side feature gets selected by the configuration link. In this example, the propagation of the artificially created constraint `B[+]` should deliver `A[+]` but results in `A[+]` $\wedge \neg$`C[+]`.

Results of this step were an architecture model, which contains only architectural features, a functional model, which contains only features that describe system and user functions, and a configuration link from the former model to the last-mentioned model (cf. Figure 7.10). This configuration link ensures all constraints of the original model. Table 7.7 shows some metrics of all created models. *Original Tuner Model in CVM* denotes the tuner model in CVM with all architectural and functional features (described in Section 7.3.3). The other three models in this table depict the extracted architecture model, the functional model and the configuration link between them (cf. Figure 7.10). Note that the architecture model and the functional model together

№	pure::variants	CVM
1	`A Requires B`	Criterion: `A[+]` Effect: `B[+]`
2	`A Requires B, C, ...`	*not possible*
3	`A RequiresAll B`	Criterion: `A[+]` Effect: `B[+]`
4	`A RequiresAll B, C, ...`	Criterion: `A[+]` Effect: `B[+], C[+], ...`
5	`A Conflicts B`	Criterion: `A[+]` Effect: `B[-]`
6	`A Conflicts B, C, ...`	*not possible*
7	`A ConflictsAny B`	Criterion: `A[+]` Effect: `B[-]`
8	`A ConflictsAny B, C, ...`	Criterion: `A[+]` Effect: `B[-], C[-], ...`
9	`A Provides B`	Criterion: `!A[+]` Effect: `B[-]`
10	`A Provides B, C, ...`	Criterion: `!A[+]` Effect: `B[-], C[-], ...`
11	`A Provides C, B Provides C, ...` `Provides C`	Criterion: `!A[+] & !B[+] & ...` Effect: `C[-]`

Table 7.6: Transfer of pure::variants relations to configuration decisions

contain more features than the original feature model since the architecture model was enriched by additional architecture information during the case study, which was not contained in the original feature model.

	Original Tuner Model in CVM	Architecture Model	Functional Model	Configuration Link
Features	122	84	92	-
Feature Links	56	5	27	-
Feature Constraints	28	0	3	-
Configuration Decisions	-	-	-	38

Table 7.7: Models of the industrial case study

7.3.5 Application of Feature Constraint Propagation

After all models were created, the technique of feature constraint propagation was applied to all constraints of the functional model. It delivered, after a runtime of four seconds, three propagated feature links and one propagated feature constraint.

These results were analyzed subsequently. For this purpose, the log of the propagation was consulted. It shows every target-side constraint (feature link or feature constraint) and its propagation result separately. So, the identification of target-side constraints that cause an individual propagated source-side constraint is possible. In addition, the log shows all reverse mappings of involved configured feature identifiers and features for deeper analyses. With this information we can identify all configuration decisions that affect the propagated constraints. The complete analysis of the four propagated constraints took about 20 minutes.

The propagated feature constraint in the architecture model was that a concrete reception technology has to be deselected, i.e. the corresponding feature is *dead*. This means that this reception technology (in the following described by feature `ReceptionTechnology`) cannot be used in any tuners. Obviously, this constraint indicates an error in the new variability specification. If `ReceptionTechnology` is selected, the configuration link selects a feature of the functional model that displays some of the received data (in the following called `DisplayFeature`). However, there is a different feature in the functional model that provides data for the display (in the following called `ProvideFeature`). This feature is not selected automatically – by no configuration decision. A `needs` feature link from `DisplayFeature` to `ProvideFeature` leads to the propagated constraint because the selection of `ReceptionTechnology` in the architecture model results, through the application of the configuration link, in the selection of `DisplayFeature` in the functional model. This violates the mentioned feature link. To solve this problem, a simple configuration decision was added to the configuration link: the selection of `ReceptionTechnology` implies the selection of `ProvideFeature`. With this configuration decision the technique of feature constraint propagation was applied again and the propagated constraint disappeared.

At this point, the question arises why the original model does not contain a constraint that corresponds to the added configuration decision. To answer this question consider the following constraints of the original model (in pure::variants syntax) `ReceptionTechnology requires DisplayFeature` and `DisplayFeature requires ProvideFeature`. Since the implication is transitive, the constraint that `ReceptionTechnology requires ProvideFeature` is derived by these constraints. This means that it is included in the original model implicitly. During the extraction of the architecture model and the definition of the configuration link, all explicitly stated constraints of the model were respected – not implicit ones. Therefore, this constraint had not been originally formulated as configuration decision but was identified by feature constraint propagation and added later.

Finally, the three propagated feature links were analyzed. Analogous to the propagated feature constraint, these feature links follow from implicit constraints of the original model. Again, the technique of feature constraint propagation provides the possibility to identify a missing configuration decision. After this configuration decision had been added to the configuration link, feature constraint propagation was applied again. The result was `true`, i.e. there are no implicit constraints. In other terms: every valid configuration of the architecture model leads to a valid configuration of the functional model. With this result the case study was finished.

7.3.6 Conclusions of the Industrial Case Study

Based on the achieved results of the case study, several important conclusions, which state the applicability and usefulness of feature constraint propagation in practice, can be drawn. Conclusions are separated into conclusions regarding the technique of feature constraint propagation and conclusions regarding the area of application and the concept of configuration links. Former conclusions are presented first since they are most important for this thesis.

1. **Feature constraint propagation identifies errors in variability specifications.** The results of the case study attest that the application of feature constraint propagation can reveal errors in the variability specification and helps to correct them. Without this technique these errors can only be identified manually, i.e. through the application of the configuration link to a concrete source configuration and the investigation of the resulting target configuration. Indeed, this manual strategy allows to identify single errors (with great effort), but the identification of all errors is not feasible for models with the size of the tuner. This is caused by the fact that the number of configurations is extremely large (remember the variation count of $4.66 * 10^{28}$ of the tuner model) and not all configurations can be checked manually.

2. **Runtime of feature constraint propagation is appropriate.** One important aspect of the implementation of feature constraint propagation was runtime optimization. This is, as already mentioned, so important since transformed formulae can become very long and complex, even if models are comparatively small. The application of feature constraint propagation to the realistic model of the case study, which took four seconds, shows that the runtime of the propagation is appropriate for the use with realistic models (of a comparable complexity). Section 7.2.3 contains more information about performance of feature constraint propagation, however, all models of these tests are, in contrast to the model of the case study, fictitious.

3. **Restructuring of logical formulae is adequate.** After the transformation of the target-side constraints, the resulting source-side constraint was very long, complex and incomprehensible for the user. The restructuring steps of feature constraint propagation (minimization, lifting of inclusion to selection statements and extraction of feature links) converted the result into one atomic constraint and three `needs` feature links. This representation of the constraints was rated to be an adequate and clearly understandable form. It confirms that the restructuring algorithms can deal with a realistic model in a suitable manner.

4. **Tracing of constraints possible (can be improved).** When using feature constraint propagation in practice, the origin of a propagated constraint plays a very important role. The user normally wants to know "where a propagated constraint comes from". The analysis of the propagated constraints was performed manually with the help of the propagation's log. A duration of about 20 minutes for the detailed analysis of the origins of four constraints in a

model of this size is moderate in the opinion of the author. It became apparent that all required information for the determination of the origins of the individual constraints is available and logged by feature constraint propagation. However, the user has to be very familiar with the details of the technique in order to understand and interpret the log correctly. Because of this reason, the development and embedding of a tracing approach is reasonable for the practical application of feature constraint propagation. This approach should automate the analysis of the available data by filtering unessential information and presenting essential information to the user in a graphical and clearly understandable form. The inclusion of a good tracing approach would surely simplify analysis of the propagated constraints and increase its speed, especially for users which are not familiar with all details of feature constraint propagation.

The following conclusions refer to the area of application and the concept of configuration links.

5. **The use of configuration links to separate hardware architecture from system and user functions is useful in practice.** The basic idea of configuration links is to derive (partial pre-) configurations of the target feature model from given source configurations. The application of this idea to the case study is that a concrete architecture leads to a partial preselection of system and user functions. Features that are forbidden by the architecture are deselected in this preconfiguration (e.g. if there is no receiver for video signals, functions regarding all video contents are not available). In addition, features that are required by the architecture are selected (e.g. the reception of some digital data content requires special sorting algorithms). This is useful since the effort for configuring a concrete tuner is decreased and constraints between architecture and functions are fulfilled automatically.

6. **The use of configuration links reduces the complexity of feature models.** Certainly, the complexity of a given product line cannot be reduced but the concept of configuration links allows to capture the information in several smaller feature models instead of in one monolithic feature model. This means that the complexity of the individual feature models decreases when using configuration links because every feature model describes only the variability of a subordinate product line or an abstract view on the product line. Within the case study, the architecture model and the functional model are less complex than the original tuner model, although, the architecture model was enriched by additional information, which was not contained in the original tuner model. However, not only the number of features of the individual models decreases but also the number of constraints, as Table 7.7 depicts. The original tuner feature model contains 84 constraints (feature links and feature constraints), whereas the architecture and the functional model contain only 35 constraints (feature links and feature constraints) together. Naturally, the complexity of the new variability representation of the overall product line (the architecture model, the configuration link and the functional model) is analo-

gous to the complexity of the original variability representation (the original tuner feature model). All constraints that do not appear in the new feature models are formulated as configuration decisions in the configuration link, as already described in Section 7.3.4. This conclusion corresponds to the intention of hierarchical organization with configuration links [Rei08].

7. **Configuration links can ensure some – but not all – constraints between feature models.** Within the case study, a configuration link was defined according to given dependencies between hardware architecture and functions. For every existing constraint, a configuration decision was formulated that ensures its satisfaction. Occurring contradictions between configuration decisions have to be handled particularly and can lead to additional constraints in feature models as mentioned above. Note that not every constraint can be ensured by a configuration decision, as Table 7.6 illustrates. Constraints that cannot be ensured are constraints with a disjunction as effect. The reason for this fact is caused by the functional behavior of configuration links and is consistent to its basic idea.

The presented conclusions show the practical applicability of feature constraint propagation: it was possible to straightforwardly apply the concept to a realistic example and the propagation delivered short and understandable constraints in an appropriate runtime. The fact that feature constraint propagation helped to identify errors in the variability specification is a first indication for its practical usefulness. In addition, the results of the case study were presented to employees of different departments of Daimler AG who were knowledgeable about the original variability representation of the tuner. The engineers confirmed that the separation of two feature models, the configuration link and feature constraint propagation significantly improved the overall maintainability, readability and understandability. This is another indication for the usefulness.

Threats to Validity

Of course, this case study does not provide strong empirical evidence of applicability and usefulness, due to the small sample and because no quantifiable measures were applied. In addition, the author himself – and not an independent subject – restructured the variability representation, applied feature constraint propagation and analyzed the resulting constraints. In order to provide a strong empirical evidence of applicability and usefulness, the accomplishment of numerous case studies with different independent subjects and their empirical analysis is necessary. However, the study provides good indication that these two objectives are met.

7.4 Limitations of Feature Constraint Propagation

In principle, feature constraint propagation is able to propagate arbitrary constraints that can be expressed by propositional logic along arbitrary configuration links. It is fully compatible with advanced feature modeling concepts, as cloned features,

feature inheritance and parameterized features, and reveals all implicit effects of the constraints of lower-level models within the higher-level feature model. Identifying these implicit effects is, from a conceptional point of view, always useful. The derived constraints can then be added to the source model if they are acceptable or, alternatively, the models (a feature model or the configuration link) can be adapted such that the implicit constraints disappear. We cannot imagine a use-case in which it is useful to define a configuration link that derives invalid target configurations from valid source configurations. Nevertheless, there are some limitations of the technique that should not be overlooked.

The experimental study revealed that the runtime of the technique can constitute a problem concerning its practical use if the resulting formulae are very complex. In the benchmark tests one single propagation took over 12 hours, although the models were not very large (cf. Table 7.4). The reason for this high runtime was that the resulting constraint was hard to minimize. We have already discussed that the minimization is the crucial step (w.r.t. the runtime) of feature constraint propagation. Since the problem of logic minimization is very complex (cf. Section 5.1.4) and we already use a heuristic algorithm, there is no possibility to strongly improve the runtime of this step with algorithmic methods. On the other hand, the industrial study shows that the propagation can deal with much larger models in an appropriate runtime. We think that propagated constraints in realistic models are, in general, not as complex as in our generated fictitious models. This is primarily because configuration links (and especially the criteria of their configuration decisions) in realistic models are not as complicated as in our fictitious models. Within the accomplished industrial study most configuration decisions had atomic criteria. However, these are assumptions and further industrial studies are required to verify (or decline) them.

Although the use of feature constraint propagation is, from a conceptional point of view, useful for every configuration link, we think that reasonable and significant constraints cannot always be propagated in practice. If the source and the target feature model are “wide apart” (i.e. there is a huge difference between their levels of abstraction), there are most likely no reasonable and significant derived constraints, which are really added to the higher-level model. For example, constraints of a very low-level feature model do surely not affect marketing decisions in a realistic scenario. Abstract feature models are in general much smaller than specific feature models. However, even in this setting a configuration link can bind most parts of the variability described by the lower-level model. This means that the variability of the specific model is encapsulated and the interface for its variability – the abstract feature model – provides only a few accessible configurations. For example, a small customer-oriented feature model that provides only a few products is related to a technical feature model with hundreds of features via a configuration link that configures all features of the technical feature model (cf. the example in Figure 1.2). The configuration link is then very complex and contains a lot of configuration decisions. Consequently, implicit effects can easily arise, e.g. because of the absence of a configuration decision or an error in a configuration decision. In this setting, feature constraint propagation can indeed be used to verify the absence of implicit effects and to reveal errors in the variability specification, but we think that it will surely

not reveal reasonable and significant implicit constraints, which are really added to the higher-level model. Further industrial studies are required to investigate which conditions the models have to fulfill so that feature constraint propagation can reveal reasonable and significant constraints.

Up to now, we have not investigated how organizational issues (e.g. if different companies or organizational units are participated) affect the feasibility of feature constraint propagation. Therefore, we cannot estimate which limitations for the technique apply if the individual feature models are developed by different companies or organizational units. This requires more industrial case studies.

Chapter 8

Conclusion

This chapter gives a summary of the thesis (Section 8.1) and its contributions (Section 8.2). Subsequently, a discussion on feature constraint propagation is presented (Section 8.3). The thesis concludes with an outlook for further research (Section 8.4).

8.1 Summary

This thesis introduced the technique of feature constraint propagation, which provides a new way to deal with implicit effects arising from constraints in related variability models. The primary idea of the technique results from the three categories of contradictions in configuration links (cf. Section 2.4). [Rei08] states that it lies in the responsibility of the tool to support the engineers in finding and dealing with these contradictions. This is exactly what feature constraint propagation does. The new idea is to make constraints of lower-lever variability models accessible in higher-level models. In particular, constraints of lower-level models are transformed into constraints of higher-level models that exactly express their "meanings" within the taxonomies of the higher-level models. The adding of the propagated constraints to higher-level models makes those self-contained. This tackles some shortcomings of the constraint solving approaches in combination with configuration links. However, feature constraint propagation does not aim to replace constraint solving approaches but to complement them. After a constraint was propagated, well-known feature model analyses basing on constraint solving can be used locally.

Feature constraint propagation is tailored to the approach of hierarchical organization with configuration links introduced in Chapter 2 and tackles the arising challenge of contradictions in derived configurations. Related work of this thesis (Chapter 3) are especially feature mapping approaches and approaches for managing complex variability. In addition, approaches that use constraint propagation techniques in a different manner and automated feature model analysis approaches are related to this thesis. Chapter 4 contains a detailed problem analysis and describes the basic idea of feature constraint propagation. Special attention is paid to advanced concepts, especially cloned features and inheritance, since these concepts provide a new layer of complexity in feature models, configuration links and also fea-

ture constraint propagation. As result of this chapter, a set of requirements for the technique is revealed. Then, the technique is rigorously introduced on a conceptual level in Chapter 5, in which the requirements are revisited and it is shown that they are all met. The technique consists of six consecutive steps. Every step is introduced in detail and applied to an example for the purpose of illustration. Furthermore, a runtime approximation for all steps is given. In the next part of this chapter, some advanced considerations on feature constraint propagation are presented, as for example the combination with feature model analyses, the compatibility with incremental configuration and the compatibility with advanced feature constraints. One more mentionable advanced consideration of this chapter is the introduction of forward feature constraint propagation, which allows to propagate constraints from a higher-level to lower-levels. Subsequently, feature constraint propagation is compared to alternative approaches. In Chapter 6 the existing formalization of feature models and configuration links [Rei08] is revisited and revised. Feature constraint propagation is formalized and embedded into this formalization. In addition, the essential properties correctness and minimality of the technique are proven. The evaluation of feature constraint propagation is presented in Chapter 7. For this purpose, the prototypical implementation of the technique and the underlying framework are presented at first. To be precise, several implementations of feature constraint propagation are introduced. They differ only in details but have very different runtimes. Two ways of evaluation were applied: an experimental case study and an industrial case study. In the experimental case study, an automated test framework was developed and automated tests were conducted and documented. These tests comprise functional tests as well as performance tests. The functional tests provide a strong indication that the implementation of feature constraint propagation as well as the underlying framework were implemented correctly. In addition, they underline that also the non-formalized parts of the technique are correct. The performance tests of feature constraint propagation compare the runtimes of the different implementations and the different steps of the technique and show how the runtime is related to the model size and different parameters. The described industrial case study was conducted in cooperation with Daimler AG. Within the study, feature constraint propagation was applied to a realistic model. The results show that feature constraint propagation is applicable in practice and indicate its usefulness.

8.2 The Contributions of this Thesis (Refined)

We have already mentioned the main contributions of this thesis in Section 1.3. At this point, we want to refine these contributions and provide several other contributions, which are scattered across the text. Each contribution is listed in the following with a reference to the part of the thesis where it is described in detail.

- presentation of a new way to deal with constraints in hierarchically organized product lines (Section 1.2)
- a definition of the properties correctness and minimality in the context of feature constraint propagation (Section 1.2 and Section 6.3)

- a problem analysis for feature constraint propagation that introduces advanced concepts step-by-step (Chapter 4)
- a rigorous definition of feature constraint propagation on a conceptual level (Section 5.1)
- a runtime approximation for every step of feature constraint propagation (Section 5.1)
- several advanced considerations on feature constraint propagation (Section 5.2)
  - combination with automated analysis approaches (Section 5.2.1)
  - propagation along configuration links with multiple sources and targets (Section 5.2.2)
  - compatibility with incremental configuration (Section 5.2.3)
  - propagation of multiple constraints together (Section 5.2.4)
  - compatibility with advanced feature constraints (Section 5.2.5)
  - definition of forward propagation on a conceptual level (Section 5.2.6)
- comparison of feature constraint propagation with alternative approaches (Section 5.3)
- revision of the existing formalizations of feature models, configurations and configuration links (Section 6.1 and Section 6.2)
- formal definitions of (basic and advanced) feature constraints, feature links and the underlying semantics (Section 6.1 and Section 6.4)
- formalization of (basic and advanced) feature constraint propagation (Section 6.3 and Section 6.5)
- verification of correctness and minimality of (basic and advanced) feature constraint propagation (Section 6.3 and Section 6.5)
- several implementations of feature constraint propagation (Section 7.1)
- implementation of an automated test framework for feature constraint propagation (Section 7.2.1)
- functional testing of feature constraint propagation (Section 7.2.2)
- performance testing of feature constraint propagation (Section 7.2.3)
- an industrial case study at Daimler AG with feature constraint propagation (Section 7.3)

8.3 Discussion

After the contributions of this thesis were presented in the previous section, we discuss the technique of feature constraint propagation with respect to some criteria.

Theoretical Foundation. At first, we defined feature constraint propagation on a pure conceptual level, independent of any technology or concrete implementation. It is very important that a new technique is clearly defined before it is implemented. Although the conceptual definition is very detailed, we decided to formalize feature constraint propagation to provide a rigorous theoretical foundation. In general, a formalization serves to avoid ambiguities, provides a formal semantics and allows to compare a technique with other existing techniques. In addition, it helps implementing a technique. Because of these reasons, we attached great importance to a sound formalization of feature constraint propagation. Our formalization requires a rigorous formalization of the basic concepts, as advanced feature models, the exact semantics of feature constraints and feature links, configuration links and their application. Consequently, the formalization is embedded into the existing formal framework of [Rei08], which is revised in this thesis. Besides the already mentioned practical reasons for a formalization, the formalization of feature constraint propagation constitutes a research result on its own.

Correctness. Before we discuss the correctness of feature constraint propagation, we have to clearly define what we mean with "correctness". In the context of this thesis, we address the correctness of (1) the technique with respect to the requirements, i.e. the technique meets all requirements, (2) the technique with respect to the defined properties correctness and minimality, i.e. the technique is correct and minimal, and (3) the implementation, i.e. it exactly implements the defined technique. In this thesis we clearly formulated requirements for feature constraint propagation and argued that the developed technique meets them (№ 1 above). We formally defined the properties correctness and minimality and proved that feature constraint propagation fulfills them (№ 2 above). In particular, we showed that the transformation of feature constraint propagation is correct and minimal and argued that the other steps, i.e. the expansion of the configuration link and the restructuring steps of the resulting constraint, are also correct in the way that they do not change the semantics of the configuration link or the constraint. Admittedly, this proof does not give evidence that the implementation is correct. This is the goal of the functional tests within the experimental case study, which serve to reveal implementation errors. These tests address the implementation of feature constraint propagation as well as the implementation of CVM. They also comprise the non-formalized parts of feature constraint propagation. The results of the functional tests indicate that the implementation is correct (№ 3 above). However, tests can never prove the absence of errors but only their appearance, in contrast to a formal verification. The first functional tests of feature constraint propagation revealed some errors in the implementation. Interestingly, all detected errors were located in the non-formalized parts of the implementation. This indicates that a formalization helps to implement a concept and to avoid technical errors. Summarizing, all three mentioned types of correctness are addressed in this thesis.

Practical Applicability. The practical applicability of feature constraint propagation is shown by means of the industrial case study at Daimler AG. It was possible to straightforwardly apply the technique to a realistic example. The runtime of the propagation and the restructuring of the resulting constraints were appropriate. In addition, the accomplished experimental case study also contains performance tests showing the scalability of the technique and evaluating the runtimes of the different implementations. The performance tests also revealed that the runtime of the propagation can constitute a problem if the constraint to be minimized is very complex. This can be a limitation for the applicability of the technique in connection with large feature models and configuration links.

Usefulness. From a conceptional point of view, the technique is very useful since it makes implicit effects explicit. The implicit constraints can be added to the higher-level feature models or the models (the feature models, the constraints or the configuration links) can be revised such that the implicit constraints disappear. The absence of implicit constraints can be verified by feature constraint propagation. If the result of a propagation is `true`, there are no implicit constraints. Then, the source feature model can be configured without hesitation because all configurations lead, through the application of the configuration link, to valid target configurations. We cannot imagine a use-case in which it is useful that a valid source configuration leads to an invalid target configuration. In addition, feature constraint propagation allows to validate properties defined by configuration links and to check whether changes in a lower-level feature model affect higher-level feature models. From a practical point of view, the significance and the presentation of the propagated constraints are essential for the usefulness of the technique. Not every propagated constraint is an acceptable constraint, which is really added to the higher-level model. We think that the meaning and the significance of propagated constraints are highly context-sensitive. Sometimes a propagated constraint indicates an error in the variability specification, sometimes it is a "real" derived constraint, which should be added to the model. We think that it is not possible to define generic criteria for models that allow to estimate the significance of propagated constraints. Consequently, the engineers have to be involved and they have to analyze the propagated constraints. If the constraints are easy to understand, their analysis is faster and less error-prone. The accomplished industrial case study at Daimler AG provides some indications for the practical usefulness of the technique, but it does not prove it.

Finally, we want to point out that all methods of verification and evaluation applied in this thesis turned out to be very sensible and useful since every method follows a different goal. Figure 8.1 illustrates the applied methods: (1) a formal verification, (2) an experimental case study and (3) an industrial case study. Although all three methods address the same technique, they follow different goals, as already discussed above. The formal verification proves that the concept is defined correctly and fulfills the desired properties. However, it does not give evidence that the implementation is correct. This is addressed by the experimental case study, which indicates that the implementation is correct and evaluates its scalability. The

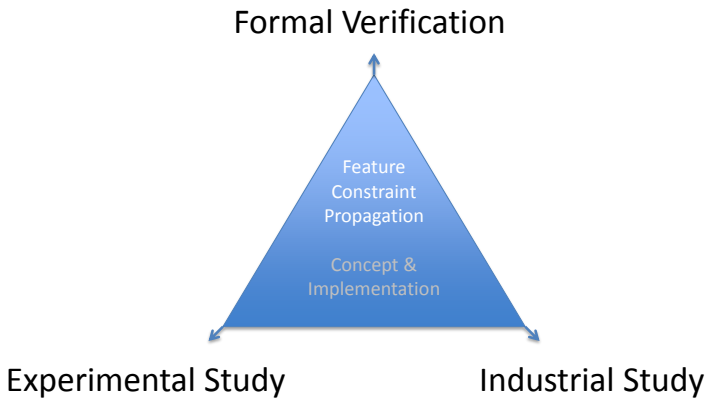


Figure 8.1: Three applied methods of evaluation and verification

industrial case study shows the practical applicability and indicates the practical usefulness of the technique.

8.4 Future Work

The research conducted during the preparation of this thesis arises the following future work.

Investigation of the practical usefulness of feature constraint propagation. The case study in this thesis provides a first indication for the practical usefulness of feature constraint propagation. However, practical usefulness of the technique should be further investigated. Under which conditions are propagated constraints reasonable and significant? We think that feature constraint propagation can always be used to identify errors or undesired effects, however, it is not always appropriate to really add the constraints to the feature models, as in the example from the beginning (Figure 1.2). Criteria for the usefulness of feature constraint propagation and the treatment of propagated constraints should be defined. Moreover, it should be investigated how organizational issues (e.g. if different companies or organizational units are participated) affect these criteria. What are the consequences if, for example, some artifact lines are part of the internal development of separate legal entities? Further industrial case studies with independent subjects are required to investigate the mentioned issues.

Analysis of arising methodological implications. Although feature constraint propagation is not a new variability modeling approach but a technique to complement an existing approach, it is imaginable that its use affects the development methodology. The methodological point of view is not considered in this thesis. However, it is an important aspect for the practical use of the technique and should be investigated. This is a challenging task and requires several comparable industrial case studies.

Development of a tracing approach for feature constraint propagation. The accomplished industrial case study at Daimler AG shows that the identification of the propagated constraints' origin is very important for practical usefulness. At

the moment, this activity has to be accomplished manually with the help of the propagation's log, which can be a time-consuming and error-prone task. In addition, it requires the user to be very familiar with the details of feature constraint propagation. Consequently, the development of an automated tracing approach for the propagated constraints would be very useful.

Exhaustive evaluation of the postprocessing process. Up to now, we have mainly investigated the correctness and minimality of the propagated constraints. The quality of the postprocessing steps (step 4-6) was only rated manually by means of many examples. In addition, the industrial case study indicated that the postprocessing is appropriate. Nevertheless, further studies of the these steps should be accomplished.

Definition of all steps and implementation of advanced feature constraint propagation. Although the reverse mappings and the transformation of advanced feature constraints are formalized and verified (cf. Section 6.5), a rigorous definition of the remaining propagation steps is lacking (cf. Section 5.2.5). In addition, advanced feature constraint propagation should be implemented and tested.

Allowing constraints over data types or the number of instances of cloned features. Up to now, constraints can only address the presence or absence of features. One possible extension is to allow constraints over data types (e.g. "the value of a parameterized feature `F` is greater than 42"). Furthermore, constraints addressing the number of instances of cloned features (e.g. "cloned features `F` and `G` have at most three selected instances together") are imaginable. If such constraints shall be allowed, their effects to feature constraint propagation have to be investigated. Can they be propagated? Does the propagation remain correct and minimal? Can all parts of the definition of valid configurations (Definition 6.1.14) be covered?

Complete definition, formalization, verification, evaluation and implementation of forward feature constraint propagation. The concept of forward propagation is only briefly introduced in this thesis (cf. Section 5.2.6). Up to now, it is not compatible with advanced feature modeling concepts, it is not formalized, its properties are not proven and it is not implemented and evaluated. Thus, a lot of further research is required to thoroughly investigate forward feature constraint propagation.

Further analysis on the mentioned advanced considerations. At this point, we do not want to sum up all advanced considerations presented in Section 5.2. However, all these considerations are only briefly described and not investigated in detail. Further research is required to analyze them in-depth.

Appendix A

Miscellaneous

A.1 The Variability Specification Language (VSL)

Although VSL can also be used to define feature models, we only use it to define configurations, constraints and configuration links. All syntactical structures of VSL used in this thesis are summarized in this section. A detailed introduction to VSL can be found in [Rei09].

Table A.1 shows and explains all statements referring to single features. It is important to note that most of these statements can be used in configurations, in constraints, in criteria of configuration links and in effects of configuration links. According to their appearance, they have different meanings as the table shows.

In order to uniquely address features on instance level, the common dot notation is used. A colon is used to address concrete instances of cloned features, e.g. the statement `i:C.F` addresses the child `F` of the instance `i` of the cloned feature `C`. Naturally, this feature can be included, excluded, etc. as explained in Table A.1. VSL provides three functionally complete logical connectives to define feature constraints and criteria of configuration links:

VSL	classical logic	meaning
!	$\neg$	negation
&	$\wedge$	conjunction
|	$\vee$	disjunction

A.2 Coincidence Lemma for Propositional Logic

Since the use of the coincidence lemma for propositional logic [EMGR$^+$01, Lemma 16.2.6] is one of the main ideas of the accomplished correctness and minimality proof of the transformation of feature constraint propagation, it is given here.

statement	**name**	**meaning in configurations**	**meaning in constraints / criteria of configuration links**	**meaning in effects of configuration links**
`F[1]`	inclusion statement for feature `F`	`F` is included	`F` is included	`F` gets included[a]
`F[+]`	selection statement for feature `F`	`F` is selected	`F` is selected	`F` gets selected[b]
`F[0]`	exclusion statement for feature `F`	`F` is excluded	`F` is excluded *(only allowed within advanced feature constraints)*	`F` gets excluded
`F[-]`	deselection statement for feature `F`	`F` is deselected	`F` is deselected *(only allowed within advanced feature constraints)*	`F` gets excluded (!)
`C$i`	instance creation statement for instance `i` of cloned feature `C`	`i` is created	*not allowed*	`i` gets created
`P=<value>`	assignment for parameterized feature `P`	`P` has the value `<value>`	*not allowed*	value `<value>` gets assigned to `P`

Table A.1: Available statements in VSL and their meanings (excerpt)

[a] This does not hold for cloned features. In order to include an instance of a cloned feature, an additional instance creation statement is required (cf. Section 2.2).

[b] This does not hold for cloned features and their successors. In order to select a (successor of a) cloned feature, an additional instance creation statement for the corresponding instance of the cloned feature is required.

Lemma A.2.1 (Coincidence Lemma for Propositional Logic)**.** *Let P be a set of literals, $\varphi \in Form(P)$ a propositional logic formula, $\sigma_1, \sigma_2 : P \to Form(P)$ two functions which define the substitutions $[\sigma_1]$ and $[\sigma_2]$ and $B_1, B_2 : P \to \{\top, \bot\}$ two assignments with $B_1^*(\sigma_1(p)) = B_2^*(\sigma_2(p))$ for all literals p of φ. Then it holds that $B_1^*(\varphi[\sigma_1]) = B_2^*(\varphi[\sigma_2])$.*

Proof. This lemma can be proven by structural induction (see [EMGR$^+$01]). □

A.3 Advanced Reverse Deselection Mapping

This section introduces the advanced reverse deselection mapping of configured feature identifiers. It is not required in the formalization of advanced feature constraint propagation (cf. Section 6.5) because the formalization does not contain an abbreviation for deselection. However, if advanced feature constraint propagation is defined completely, this mapping should be used to propagate deselection statements in constraints.

Definition A.3.1 (Advanced Reverse Deselection Mapping of Configured Feature Identifiers w.r.t. a Configuration Link)**.** Given configuration link $CL_{S\to T}$ with advanced feature constraints. Then the advanced reverse deselection mapping of configured feature identifiers with respect to this configuration link is given by function $ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}} : \mathcal{CFIDI}_T \to FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with

$$ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) = \begin{cases} ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) & \text{if } isRoot_T(f(cfid_T)) \\ \left(\begin{array}{l} ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \vee ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \end{array}\right) & else \end{cases}$$

Lemma A.3.2 (Correctness and Minimality of the Advanced Reverse Deselection Mapping of Configured Feature Identifiers)**.** *Given two feature models S and T with advanced feature constraints, configuration link $CL_{S\to T}$ with advanced feature constraints, configuration $C_S \in \mathcal{C}_S$ and a configured feature identifier $cfid_T \in \mathcal{CFIDI}_T$. Then the following statement holds for every bijective projection $\pi : R \rightleftarrows FormA(\mathcal{CFIDI}_S)_{/\equiv}$ with R being a complete system of representatives.*

$$C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff CL_{S\to T}(C_S) \nvdash cfid_T$$

Proof. According to Definition 6.1.9, we have to show the following statement.

$$C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \iff \sigma_{CL_{S\to T}(C_S)}(cfid_T) = \bot$$

Proof by mathematical induction.

Basis step. Let $isRoot_T(f(cfid_T))$ hold.

$$
\begin{aligned}
&\sigma_{CL_{S\to T}(C_S)}(cfid_T) = \bot \\
&\iff \tau_{CL_{S\to T}(C_S)}(cfid_T) = \bot && \text{by Definition 6.1.9} \\
&\iff C_S \models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Lemma 6.5.5} \\
&\iff C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) && \text{by Definition A.3.1}
\end{aligned}
$$

Inductive step. Let

$$C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \iff \sigma_{CL_{S\to T}(C_S)}(Parent(cfid_T)) = \bot$$

hold for the parent of $cfid_T$ (inductive hypothesis).

$$\sigma_{CL_{S\to T}(C_S)}(cfid_T) = \bot$$

$$\overset{\text{Definition 6.1.9}}{\iff}$$

$$\tau_{CL_{S\to T}(C_S)}(cfid_T) = \bot \vee \sigma_{CL_{S\to T}(C_S)}(Parent(cfid_T)) = \bot$$

$$\overset{\text{inductive hypothesis}}{\iff}$$

$$\tau_{CL_{S\to T}(C_S)}(cfid_T) = \bot \vee C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T))$$

$$\overset{\text{Lemma 6.5.5}}{\iff}$$

$$C_S \models \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T)$$
$$\vee C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T))$$

$$\overset{\text{logic}}{\iff}$$

$$C_S \models \begin{pmatrix} \pi^{-1} \circ ARevExc^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T) \\ \vee \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(Parent(cfid_T)) \end{pmatrix}$$

$$\overset{\text{Definition A.3.1}}{\iff}$$

$$C_S \models \pi^{-1} \circ ARevDesel^{\mathcal{CFID}}_{CL_{S\to T}}(cfid_T)$$

□

Nomenclature

ADORA	Analysis and Description of Requirements and Architecture
aka	also known as
BDD	binary decision diagram
BeTTy	Benchmarking and Testing on the Analysis of Feature Models
cf.	confer (compare)
Ch.	Chapter
CPU	central processing unit
CSP	constraint satisfaction problem
CSV	comma-separated values
CVM	Compositional Variability Management
DAG	acyclic digraph
Def.	Definition
DyReS	Dynamic Reconfiguration Support
e.g.	exempli gratia (for example)
EMF	Eclipse Modeling Framework
et al.	et aliae / et alii (and others)
etc.	et cetera (and so on)
FAMA	Feature Model Analyser
GEF	Graphical Editing Framework
GHz	Gigahertz
i.e.	id est (that is)
min	minutes

MOF	Meta-Object Facility
ms	milliseconds
№	number
NP	nondeterministic polynomial time
OCL	Object-Constraint Language
P	polynomial time
PC	personal computer
PLA	programmable logic array
QSAT	satisfiability problem of quantified Boolean formulae
RAM	random-access memory
resp.	respectively
SAT	Boolean satisfiability problem
SMT	satisfiability modulo theories
SPLANE	Software Product Line Analysis Engine
SPREBA	Software Product Line Requirements Engineering Based on Aspects
TVL	Text-based Variability Language
U.S. / USA	United States of America
UI	user interface
UML	Unified Modeling Language
VSL	Variability Specification Language
w.r.t.	with respect to
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Index of Terms

List of Figures

List of Tables

List of Algorithms

Bibliography

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 424–427, Washington, DC, USA, 2011. IEEE Computer Society.

[ACLF12] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Separation of Concerns in Feature Modeling: Support and Applications. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD 2012)*, pages 1–12, New York, NY, USA, 2012. ACM.

[AJL+10] Andreas Abele, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, David Servat, Martin Töngren, and Matthias Weber. The CVM Framework – A Prototype Tool for Compositional Variability Management. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010)*, volume 37 of *ICB-Research Report*, pages 101–105. Universität Duisburg-Essen, 2010.

[ALHM+11] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Supporting Consistency Checking between Features and Software Product Line Use Scenarios. In *Proceedings of the 12th International Conference on Top Productivity through Software Reuse (ICSR 2011)*, pages 20–35, Berlin, Heidelberg, 2011. Springer.

[ASM03] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Ontology for Configurable Software Product Families. In *Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI 2003), Configuration Workshop*, Acapulco, Mexico, August 2003.

[ASM04] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In Frank van der Linden, editor, *Proceedings of the*

5th International Workshop on Software Product-Family Engineering (PFE 2003), Revised Papers, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer, 2004.

[Bar01] Roman Barták. Theory and Practice of Constraint Propagation. In *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control (CPDC 2001)*, pages 7–14, 2001.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC 2005)*, pages 7–20. Springer, 2005.

[BCFH10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, pages 159–162. University of Duisburg-Essen, January 2010.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[Bec03] Martin Becker. Towards a General Model of Variability in Product Families. In *Proceedings of the 1st Workshop on Software Variability Management*, Groningen, Netherlands, 2003.

[Bed02] Thomas Bednasch. Konzept zur Implementierung eines konfigurierbaren Metamodells für die Merkmalsmodellierung. Diploma thesis, in German, Fachhochschule Kaiserslautern, 2002.

[Ben07] David Benavides. *On the Automated Analysis of Software Product Lines using Feature Models: A Framework for developing Automated Tool Support*. Dissertation, University of Seville, Spain, 2007.

[Bes06] Christian Bessière. Constraint Propagation. Technical report, University of Montpellier, 2006.

[BLP04] Stan Bühne, Kim Lauenroth, and Klaus Pohl. Why is it not Sufficient to Model Requirements Variability with Feature Models. In Mikio Aoyama, Frank Houdek, and Takashi Shigematsu, editors, *Proceedings of the International Workshop on Automotive Requirements Engineering*, Los Alamitos, September 2004. Nanzan University, Nagoya, Japan, IEEE Computer Society Press.

[BLPW04] Stan Bühne, Kim Lauenroth, Klaus Pohl, and Matthias Weber. Modeling Features for Multi-Criteria Product-Lines in the Automotive Industry. In *Proceedings of the Workshop on Software Engineering for Automotive Systems (SEAS 2004) at the 26th International*

Conference on Software Engineering (ICSE 2004), pages 9–16, May 2004.

[BRCTS06] David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. A Survey on the Automated Analyses of Feature Models. In José Cristóbal Riquelme Santos and Pere Botella, editors, *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006)*, pages 367–376, 2006.

[BSRC09] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models: A Detailed Literature Review. Technical Report ISA-09-TR-04, ISA research group, 2009. Available at `http://www.isa.us.es/` (March 2013).

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6), 2010.

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[BSTRC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In Klaus Pohl, Patrick Heymans, Kyo Chul Kang, and Andreas Metzger, editors, *Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VaMoS 2007)*, volume 2007-01 of *Lero Technical Report*, pages 129–134, 2007.

[BTRC05a] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005)*, pages 491–503, Berlin, Heidelberg, 2005. Springer.

[BTRC05b] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005)*, pages 677–682, 2005.

[BU11] David Buchfuhrer and Christopher Umans. The Complexity of Boolean Formula Minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011.

[Bur93] Neil Burkhard. Reuse-Driven Software Processes Guidebook. Version 02.00.03. Technical Report SPC-92019, Software Productivity Consortium, November 1993.

[CAB11] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, April 2011.

[CBA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, pages 156–172, London, UK, 2002. Springer.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012)*, pages 173–182, New York, NY, USA, 2012. ACM.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration Using Feature Models. In Robert L. Nord, editor, *Proceedings of the 3rd International Software Product Line Conference (SPLC 2004)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process: Improvement and Practice*, volume 10, pages 7–29, 2005.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *Proceedings of the Theory and Practice of Software (ETAPS 2008), 11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008)*, pages 16–30, Berlin, Heidelberg, 2008. Springer.

[CK05] Krzysztof Czarnecki and Chang H. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report, October 2005.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 211–220, New York, NY, USA, 2006. ACM.

[CR08] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison-Wesley, 3rd edition, 2008.

[CVM12] CVM Homepage, October 2012. `http://www.cvm-framework.org/`.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[Dai12] Daimler AG Homepage, June 2012. `http://www.daimler.com`.

[DGR11] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.

[Ecl12] Eclipse Foundation Homepage, October 2012. `http://www.eclipse.org/`.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.

[EMF12] EMF Homepage, October 2012. `http://www.eclipse.org/modeling/emf/`.

[EMGR+01] Hartmut Ehrig, Bernd Mahr, Martin Große-Rhode, Felix Cornelius, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Lehrbuch. Springer, 2001. In German.

[EPAH08] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Knowledge Based Method to Validate Feature Models. In Steffen Thiel and Klaus Pohl, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Second Volume (Workshops)*, pages 217–225. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[Esp12] ESPRESSO Implementation from Berkeley University of California, October 2012. http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm.

[Fab08] Fabian Lücke. Propagierung von Abhängigkeiten zwischen Featuremodellen. Diploma thesis, in German, Technische Universität Berlin, 2008.

[Fea13] FeatureMapper Homepage, May 2013. http://featuremapper.org/.

[FK93] Kenneth D. Forbus and Johan De Kleer. *Building Problem Solvers*. MIT Press, Cambridge, MA, USA, 1993.

[GBPG13] Gerd Gröner, Marko Bošković, Fernando Silva Parreiras, and Dragan Gašević. Modeling and validation of business process families. *Information Systems*, 38(5):709–726, 2013.

[GBR+00] Martin Glinz, Stefan Berner, Johannes Ryser, Stefan Joos, Nancy Schett, Reto Schmid, Yong Xia, and Robert Bosch Gmbh. The ADORA Approach to Object-Oriented Modeling of Software. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE 2001)*, Lecture Notes in Computer Science, pages 76–92. Springer, 2000.

[GEF12] GEF Homepage, October 2012. http://www.eclipse.org/gef/.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Has13] Haskell Homepage, February 2013. http://www.haskell.org/.

[HCH09] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal Modelling of Feature Configuration Workflows. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 221–230, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[Hei09] Florian Heidenreich. Towards Systematic Ensuring Well-Formedness of Software Product Lines. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD 2009)*, pages 69–74, New York, NY, USA, 2009. ACM.

[Hem08] Adithya Hemakumar. Finding Contradictions in Feature Models. In Steffen Thiel and Klaus Pohl, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Second Volume (Workshops)*, pages 183–190. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[HHS+11] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling (SoSyM)*, pages 1–23, 2011.

[HHSD10] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, and Dirk Deridder. Towards Multi-view Feature-Based Configuration. In *Proceedings of the 16th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2010)*, pages 106–112. Springer, 2010.

[HKM11] Peter Höfner, Ridha Khedri, and Bernhard Möller. An Algebra of Product Families. *Software and Systems Modeling*, 10:161–182, May 2011.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 943–944, New York, NY, USA, May 2008. ACM.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of the 15th International Conference on Software Product Lines (SPLC 2011)*, pages 150–159. IEEE, 2011.

[HST+08] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Yves Bontemps, Raimundas Matulevicius, and Andreas Classen. Evaluating Formal Properties of Feature Diagram Languages. *IET Software*, 2(3):281–302, 2008.

[HT08] Herman Hartmann and Tim Trew. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society.

[HTH13] Arnaud Hubaux, Thein Than Tun, and Patrick Heymans. Separation of Concerns in Feature Diagram Languages: A Systematic Survey (to appear). *ACM Computing Surveys*, 2013.

[HTM09] Herman Hartmann, Tim Trew, and Aart Matsinger. Supplier Independent Feature Modelling. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 191–200, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[Hub12] Arnaud Hubaux. *Feature-based Configuration: Collaborative, Dependable, and Controlled*. Dissertation, University of Namur, Belgium, 2012.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[JK07] Mikoláš Janota and Joseph Kiniry. Reasoning about Feature Models in Higher-Order Logic. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 13–22, Washington, DC, USA, September 2007. IEEE Computer Society.

[Joo00] Stefan Joos. *Adora-L – Eine Modellierungssprache zur Spezifikation von Software-Anforderungen*. Dissertation, 2000. In German.

[Kar53] Maurice Karnaugh. The Map Method for Synthesis of Combinational Logic Circuits. *Transactions of the American Institute of Electrical Engineers (AIEE), Part I*, 72(9):593–599, 1953.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, Amsterdam, 1952.

[KRR10] Florian Kammüller, Alexander Rein, and Mark-Oliver Reiser. Feature Link Propagation Across Variability Representations with Isabelle/HOL. In *Proceedings of the Workshop on Product Line Approaches in Software Engineering (PLEASE 2010) at the 32th International Conference on Software Engineering (ICSE 2010)*, pages 48–53. ACM, 2010.

[Kum92] Vipin Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI MAGAZINE*, 13(1):32–44, 1992.

[McC56] Edward J. McCluskey. Minimization of Boolean Functions. *The Bell System Technical Journal*, 35(5):1417–1444, November 1956.

[MCMdO08] Marcílio Mendonça, Donald D. Cowan, William Malyk, and Toacy Cavalcante de Oliveira. Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis. *Journal of Software*, 3(2):69–82, 2008.

[MCO07] Marcílio Mendonça, Donald Cowan, and Toacy Oliveira. A Process-Centric Approach for Coordinating Product Configuration Decisions. In *Proceedings of the 40th Hawaii International International Conference on Systems Science (HICSS-40)*, page 283. IEEE Computer Society, 2007.

[Men09] Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. Dissertation, University of Waterloo, 2009.

[MHGH12] Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. An SMT-based Approach to Automated Configuration. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT 2012)*, pages 107–117, Manchester, UK, 2012.

[MHP+07] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In Alistair Sutcliffe and Pankaj Jalote, editors, *Proceedings of the 15th IEEE International Conference on Requirements Engineering (RE 2007)*, pages 243–253. IEEE, IEEE Comuter Society, October 2007.

[MOF13] MOF Homepage, May 2013. `http://www.omg.org/mof/`.

[MRM+12] Swarup Mohalik, Sethu Ramesh, Jean-Vivien Millo, Shankara Narayanan Krishna, and Ganesh Khandu Narwane. Tracing SPLs Precisely and Efficiently. In *Proceedings of the 16th International Software Product Line Conference (SPLC 2012), Volume 1*, pages 186–195, New York, NY, USA, 2012. ACM.

[MSA09] Mike Mannion, Juha Savolainen, and Timo Asikainen. Viewpoint-Oriented Variability Modeling. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), Volume 01*, pages 67–72, Washington, DC, USA, 2009. IEEE Computer Society.

[OCL13] OCL Homepage, March 2013. `http://www.omg.org/spec/OCL/`.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9, January 1976.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer New York, Inc., Secaucus, NJ, USA, 2005.

[PCBD10] Carlos Andres Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-Based Composition of Software Architectures. In Muhammad Ali Babar and Ian Gorton, editors, *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2010.

[PLP11] Richard Pohl, Kim Lauenroth, and Klaus Pohl. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 313–322, November 2011.

[Pur12] pure-systems Homepage, June 2012. http://www.pure-systems.com/.

[RA12] Jean-Claude Royer and Hugh Arboleda. *Model-Driven and Software Product Line Engineering*. John Wiley & Sons, 2012.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT 2002)*, June 2002.

[Rei80] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[Rei08] Mark-Oliver Reiser. *Managing Complex Variability in Automotive Software Product Lines with Subscoping and Configuration Links*. Dissertation, Technische Universität Berlin, 2008.

[Rei09] Mark-Oliver Reiser. Core Concepts of the Compositional Variability Management Framework (CVM) – A Practitioner's Guide. Technical Report 2009/16, Technische Universität Berlin, 2009.

[RKW07] Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Matthias Weber. Unified Feature Modeling as a Basis for Managing Complex System Families. In Klaus Pohl, Patrick Heymans, Kyo Chul Kang, and Andreas Metzger, editors, *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2007)*, volume 2007-01 of *Lero Technical Report*, pages 79–86, 2007.

[RKW09] Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Matthias Weber. Compositional Variability – Concepts and Patterns. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS-42)*, pages 1–10. IEEE Computer Society Press, 2009.

[RSTS11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2011)*, pages 11–20, New York, NY, USA, 2011. ACM.

[RSV87] Richard L. Rudell and Alberto Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(5):727–750, September 1987.

[RW05] Mark-Oliver Reiser and Matthias Weber. Using Product Sets to Define Complex Product Decisions. In *Proceedings of the 9th International Software Product Line Conference (SPLC 2005)*, pages 21–32, Berlin, Heidelberg, 2005. Springer.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2nd edition, 2009.

[SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49:717–739, July 2007.

[SDNB04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In Robert L. Nord, editor, *Proceedings of the 3rd International Software Product Line Conference (SPLC 2004)*, volume 3154 of *Lecture Notes in Computer Science*. Springer, 2004.

[Seg08] Sergio Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[SGB+12] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2012)*, pages 63–71, Leipzig, Germany, 2012. ACM.

[SHBRC11] Sergio Segura, Robert M. Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006)*, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic Semantics of Feature Diagrams. *Computer Networks*, 51(2):456–479, 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284, 2004.

[Spi89] J. Michael Spivey. *The Z Notation: A Reference Manual*, 1989.

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of*

the 5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2011), pages 119–126, New York, NY, USA, 2011. ACM.

[STJ09a] Frans Sanen, Eddy Truyen, and Wouter Joosen. Mapping Problem-Space to Solution-Space Features: A Feature Interaction Approach. In Jeremy G. Siek and Bernd Fischer, editors, *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE 2009)*, volume 45, pages 167–176, New York, NY, USA, October 2009. ACM.

[STJ09b] Frans Sanen, Eddy Truyen, and Wouter Joosen. Problem-Solution Feature Interactions as Configuration Knowledge in Distributed Runtime Adaptations. In Masahide Nakamura and Stephan Reiff-Marganiec, editors, *Proceedings of the Feature Interaction Workshop (FIW 2009) at the 10th International Conference on Feature Interactions in Software and Communication Systems (ICFI 2009)*, pages 166–175. IOS Press, 2009.

[Sto12] Reinhard Stoiber. *A New Approach to Product Line Engineering in Model-Based Requirements Engineering.* Dissertation, University of Zurich, 2012.

[STSS13] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. Automated Analysis of Dependent Feature Models. In *Proceedings of the 7th Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2013)*, pages 9:1–9:5, New York, NY, USA, 2013. ACM.

[Tac09] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation.* Dissertation, Saarland University, January 2009.

[TBC+09] Thein Than Tun, Quentin Boucher, Andreas Classen, Arnaud Hubaux, and Patrick Heymans. Relating Requirements and Feature Configurations: A Systematic Approach. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 201–210, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William R. Cook. Safe Composition of Product Lines. In Charles Consel and Julia L. Lawall, editors, *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, pages 95–104. ACM, 2007.

[THSC06] Jean-Christophe Trigaux, Patrick Heymans, Pierre-Yves Schobbens, and Andreas Classen. Comparative Semantics of Feature Diagrams: FFD vs. vDFD. In *Proceedings of the 4th International Workshop on Comparative Evolution in Requirements Engineering (CERE 2006)*, 2006.

[TJSJ08] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. Support for Distributed Adaptations in Aspect-Oriented Middleware. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 120–131, New York, NY, USA, 2008. ACM.

[Tsa95] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[TTC12] Truth Table Constructor Implementation, October 2012. `http://www.brian-borowski.com/Software/Truth/`.

[Tut01] William Thomas Tutte. *Graph Theory*. Cambridge University Press, 2001.

[Uma98] Christopher Umans. The Minimum Equivalent DNF Problem and Shortest Implicants. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 556–563, November 1998.

[UVSv06] Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-vincentelli. Complexity of Two-Level Logic Minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

[Var13] VarMod-PRIME Project Homepage, April 2013. `http://paluno.uni-due.de/en/varmod-prime/`.

[VeA13] VeAnalyzer Homepage, July 2013. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/MultiPLe/modeling.htm`.

[vO04] Rob van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.

[WDSB09] Jules White, Brian Dougherty, Doulas C. Schmidt, and David Benavides. Automated Reasoning for Multi-step Feature Model Configuration Problems. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 11–20, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[WW03] Matthias Weber and Joachim Weisbrod. Requirements Engineering in Automotive Development: Experiences and Challenges. *IEEE Software Magazine*, 20(1):16–24, 2003.

[ZIS] Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. ISO/IEC 13568:2002, International Standard.