TKK Dissertations 132
Espoo 2008

A CONCEPTUAL MODELLING APPROACH TO SOFTWARE VARIABILITY

Doctoral Dissertation

Timo Asikainen

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

TKK Dissertations 132
Espoo 2008

A CONCEPTUAL MODELLING APPROACH TO SOFTWARE VARIABILITY

Doctoral Dissertation

Timo Asikainen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium AS1 at Helsinki University of Technology (Espoo, Finland) on the 1st of August, 2008, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 9210
FI - 02015 TKK
FINLAND
URL: http://www.soberit.tkk.fi/
Tel. +358-9-451 4851
Fax +358-9-451 4958
E-mail: reports@soberit.tkk.fi

ISBN 978-951-22-9484-8
ISBN 978-951-22-9485-5 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)
URL: http://lib.tkk.fi/Diss/2008/isbn9789512294855/

TKK-DISS-2495

Multiprint Oy
Espoo 2008

ABSTRACT OF DOCTORAL DISSERTATION	HELSINKI UNIVERSITY OF TECHNOLOGY P. O. BOX 1000, FI-02015 TKK http://www.tkk.fi/

Author Timo Asikainen

Name of the dissertation

A Conceptual Modelling Approach to Software Variability

Manuscript submitted January 10, 2008	Manuscript revised June 24, 2008
Date of the defence August 1, 2008	
☐ Monograph	☒ Article dissertation (summary + original articles)
Faculty	Faculty of Information and Natural Sciences
Department	Department of Computer Science and Engineering
Field of research	Conceptual modelling
Opponent(s)	Professor Alexander Felfernig
Supervisor	Professor Tomi Männistö
Instructor	Docent Timo Soininen

Abstract

Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context. Increasing amounts of variability are required of software systems. The number of possible variants of a software system may become very large, essentially infinite. Efficient methods for modelling and reasoning about software variability are needed and numerous such languages have been developed. Most of these languages either lack a solid conceptual foundation or a rigorous formal semantics, or both.

In this dissertation, three novel software variability modelling languages, KOALISH, FORFAMEL and KUMBANG, which synthesises KOALISH and FORFAMEL, are developed. The languages are based on concepts found relevant to modelling software variability in scientific literature and practice, namely features and software architecture. They synthesise and clarify the concepts defined in a number of previous languages. Ideas first developed in product configuration research for modelling variability in non-software products are elaborated and integrated into the languages. A formal semantics is given for the languages by translation to weight constraint rule language (WCRL).

One of the goals of this dissertation is to enable the representation of software variability knowledge at different levels of abstraction in a uniform manner, preferably using an existing conceptual modelling language with a formal semantics. Unfortunately, it turns out that no existing language meets these requirements. Consequently, a novel conceptual modelling language, NIVEL, with the necessary capabilities is developed in this dissertation. The modelling concepts of NIVEL are not based on software variability. Consequently, NIVEL can be applied in domains other than software variability and is hence generic and contributes to the theory of conceptual modelling. A formal semantics enabling automated, decidable reasoning is given for NIVEL by translation to WCRL.

NIVEL is used to give an alternative definition of KUMBANG. The alternative definition is more compact and easily understandable than the original one. Major parts of the semantics of KUMBANG are captured by the semantics of NIVEL. The definition of KUMBANG in terms of a generic modelling language also brings software variability modelling closer to other forms of modelling, thus making software variability modelling less of an isolated discipline.

Keywords conceptual modelling, variability, feature modelling, software architecture, metamodelling	
ISBN (printed) 978-951-22-9484-8	ISSN (printed) 1795-2239
ISBN (pdf) 978-951-22-9485-5	ISSN (pdf) 1795-4584
Language English	Number of pages 84 p. + app. 81 p.
Publisher Helsinki University of Technology, Department of Computer Science and Engineering	
Print distribution Helsinki University of Technology, Department of Computer Science and Engineering	
☒ The dissertation can be read at http://lib.tkk.fi/Diss/2008/isbn9789512294855/	

VÄITÖSKIRJAN TIIVISTELMÄ	TEKNILLINEN KORKEAKOULU PL 1000, 02015 TKK http://www.tkk.fi/

Tekijä Timo Asikainen

Väitöskirjan nimi

Ohjelmistojen varioituvuuden käsitteellinen mallintaminen

Käsikirjoituksen päivämäärä 10.1.2008	Korjatun käsikirjoituksen päivämäärä 24.6.2008
Väitöstilaisuuden ajankohta 1.8.2008	
☐ Monografia	☒ Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)

Tiedekunta	Informaatio- ja luonnontieteiden tiedekunta
Laitos	Tietotekniikan laitos
Tutkimusala	Käsitteellinen mallintaminen
Vastaväittäjä(t)	Professori Alexander Felfernig
Työn valvoja	Professori Tomi Männistö
Työn ohjaaja	Dosentti Timo Soininen

Tiivistelmä

Varioituvuudella tarkoitetaan järjestelmän kykyä tulla tehokkaasti laajennetuksi, muutetuksi, mukautetuksi tai konfiguroiduksi tiettyä käyttötarkoitusta varten. Ohjelmistojärjestelmiltä vaaditaan yhä enemmän varioituvuutta, ja järjestelmän mahdollisten varianttien lukumäärä voi olla erittäin suuri, käytännössä ääretön. Varioituvuuden mallintamiseen ja sitä koskevaan päättelyyn tarvitaan tehokkaita menetelmiä. Tätä tarkoitusta varten on kehitetty monia eri kieliä, joista kuitenkin puuttuu selkeä käsitteellinen perusta, täsmällinen formaali semantiikka tai molemmat.

Tässä väitöstyössä kehitetään kolme ohjelmistojen varioituvuuden mallinnuskieltä, KOALISH, FORFAMEL ja KUMBANG, joista KUMBANG yhdistää kaksi edellistä. Kielet perustuvat piirteen ja ohjelmistoarkkitehtuurin käsitteisiin, jotka on aiemmassa tutkimuksessa ja käytännössä havaittu tärkeiksi varioituvuuden mallintamisen kannalta. Kielet yhdistelevät ja selkeyttävät aiemmissa varioituvuuden mallinnuskielissä määriteltyjä käsitteitä sekä sisältävät tuotekonfiguroinnin piirissä syntyneitä ajatuksia. Kielille määritellään formaali semantiikka erään sääntöpohjaisen tiedonesittämiskielen, WCRL:n (Weight Constraint Rule Language), avulla.

Väitöstyön eräänä tavoitteena on mahdollistaa ohjelmistojen varioituvuuteen liittyvän tietämyksen esittäminen eri abstraktiotasoilla yhtenäisesti, mieluiten käyttäen olemassaolevaa käsitteellisen mallintamisen kieltä. Koska mikään olemassa oleva kieli ei täytä näitä vaatimuksia, kehitetään tarkoitukseen sopiva kieli, NIVEL, osana väitöstyötä. NIVEL on yleiskäyttöinen siinä mielessä, että sen sisältämät mallinnuskäsitteet eivät perustu eivätkä käyttökohteet siten rajoitu ohjelmistojen varioituvuuden mallintamiseen. Näin ollen NIVEL edistää käsitteellisen mallinnuksen teoriaa. NIVEL:lle määritellään WCRL:n avulla automaattisen, ratkeavan päättelyn mahdollistava formaali semantiikka.

KUMBANG:lle annetaan vaihtoehtoinen määrittely NIVEL:n avulla. Vaihtoehtoinen määrittely on tiiviimpi ja helppotajuisempi kuin alkuperäinen. Merkittävä osa KUMBANG:n semantiikasta saadaan esitettyä suoraan NIVEL:n semantiikan avulla. Lisäksi annettava vaihtoehtoinen määrittely sitoo ohjelmistojen varioituvuuden mallintaminen osaksi yleistä käsitteellistä mallintamista.

Asiasanat käsitteellinen mallintaminen, varioituvuus, piirremallinnus, ohjelmistoarkkitehtuuri, metamallinnus	
ISBN (painettu) 978-951-22-9484-8	ISSN (painettu) 1795-2239
ISBN (pdf) 978-951-22-9485-5	ISSN (pdf) 1795-4584
Kieli englanti	Sivumäärä 84 s. + liitt. 81 s.
Julkaisija Teknillinen korkeakoulu, Tietotekniikan laitos	
Painetun väitöskirjan jakelu Teknillinen korkeakoulu, Tietotekniikan laitos	
☒ Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2008/isbn9789512294855/	

Acknowledgements

This dissertation was prepared in the Laboratory of Software Business and Engineering, Department of Computer Science and Engineering, Helsinki University of Technology. The graduate assistantships of Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Department of Computer Science and Engineering are gratefully acknowledged, as well as research-project funding from the Academy of Finland, the Finnish Funding Agency for Technology and Innovation (Tekes) and the Technology Industries of Finland Centennial Foundation.

I wish to thank my colleagues in the laboratory for a stimulating working environment. In particular, I am grateful to my supervisors, Docent Timo Soininen and, later, Professor Tomi Männistö, for their advice and constructive criticism. In addition, I extend thanks to Professor Ilkka Niemelä from the Laboratory for Theoretical Computer Science for his guidance, especially that relating to knowledge representation. Finally, I am indebted to the preliminary examiners, Associate Professor Krzysztof Czarnecki and Professor Kai Koskimies, for their insightful comments that helped to improve the dissertation.

Timo Asikainen

Contents

List of Publications

This dissertation consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Timo Asikainen, Timo Soininen, and Tomi Männistö. 2003. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In: Frank van der Linden (editor), 5th International Workshop on Product Family Engineering (PFE-5), volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer.

II Timo Asikainen, Tomi Männistö, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In: Liam O'Brien (editor), 10th International Software Product Line Conference (SPLC 2006), pages 31–40. IEEE Computer Society.

III Timo Asikainen, Tomi Männistö, and Timo Soininen. 2007. Kumbang: A Domain Ontology for Modelling Variability in Software Product Families. Advanced Engineering Informatics 21, no. 1, pages 23–40.

IV Timo Asikainen and Tomi Männistö. NIVEL: A Metamodelling Language with a Formal Semantics. Accepted for publication in Software and Systems Modeling.

The author of this dissertation is the principal author of all these publications. He is responsible for all the research reported and the text in the publications. The other authors (Tomi Männistö and Timo Soininen) have participated in developing the ideas presented in the publications, supervised and given instruction in the research described in them and made suggestions for their improvement.

1 Introduction

The chapter begins with a description of the background of this dissertation. The research problem and detailed research questions are defined next, followed by a description of the research methodology applied in the dissertation. The scope and main contributions are discussed in the two subsequent sections. The chapter is concluded with an outline of the dissertation.

1.1 Background

Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context [114]. There is a growing demand for variability of software and a significant research interest in the topic, as exemplified by the workshops and special issues devoted to it, see, e.g., [125, 26, 77, 24, 98, 83]. Products that incorporate variability are useful for various purposes: they can be used to address multiple user segments, allow price categorisation, support various hardware platforms and operating systems, provide different sets of features for different needs and cover different market areas with different languages, legislation and market structure.

Software product families, or *software product lines*, as they are also called, have become an important means for implementing variability [23, 34]. A software product family is commonly defined as consisting of a common architecture, a set of reusable assets used in systematically producing individual products and the set of products thus produced [23]. Another definition considers a common managed set of features satisfying the specific needs of a market segment as a defining characteristic of a software product family [34].

A software product family may contain thousands of variation points [25, 18] and the number of individual products in a family may be essentially infinite. There may be complex interdependencies between different variation points; finding a product matching a specific set of customer requirements while taking into account the various interdependencies is both error-prone and time-consuming [34, 45]. Consequently, rigorous methods for representing and efficiently reasoning about software variability are needed. A large number of software variability modelling languages have been proposed. An important class of such languages is based on modelling the common and variable *features* of a product family [63, 64, 37, 36, 75, 133, 40, 39, 132, 20, 17]; a number of languages for modelling product family architectures have been reported [130, 44].

However, most, if not all, software variability modelling languages leave room for improvement in conceptual and semantic rigour. That is, the conceptual basis of such languages is not accurately defined through a syntax, either abstract or concrete. Furthermore, the semantics of such languages is described only informally using natural languages or, if formal details are provided, only an outline of the formalisation is presented. Instead, many authors seem to be more concerned with details of the notation or concentrate on developing tool support before the conceptual basis for such tools has been established. This condition severely undermines

the practical applicability of variability modelling languages and methods based on these, prevents systematic reasoning, both automated and other forms, about the languages and makes comparing such languages with each other unnecessarily difficult.

1.2 Research problem and questions

This dissertation strives to contribute to the theory of software variability management by developing novel software variability modelling languages with emphasis on conceptual clarity and rigorous, formal semantics. The research problem addressed is twofold, and further divided into more detailed research questions:

1. What kind of languages are best suited for representing knowledge on software variability?

   (a) What is the conceptual basis of such languages?

   (b) How should the conceptual basis of such languages be defined?

   (c) What is the formal semantics of such languages?

2. What kind of languages are best suited for defining the conceptual basis of the above-mentioned kind of languages?

   (a) What is the conceptual basis of such languages?

   (b) What is the formal semantics of such languages?

A language referred to in the first research problem will be termed a *software variability modelling language* and a language referred to in the second research problem a *metamodelling language*. The definition of a conceptual basis will be termed *abstract syntax*.

Note that the two research problems resemble each other: both problems call for the identification of existing modelling languages, or in the absence of such languages, for their development. This is also reflected in the research questions that are the same for both problems, with the exception that the issue of how to define a conceptual basis for metamodelling languages is not dealt with.

1.3 Methodology

This section describes the research methodology applied in this dissertation. First, the overall approach used in this dissertation is explained. Thereafter, a number of methodological principles applied in implementing the overall approach are discussed.

1.3.1 Overall approach

The research method adopted in this dissertation is the *constructive* one. A constructive research method is motivated by the fact that, in the absence of previous languages adequately addressing the research problem, new languages must be constructed to solve it.

According to Kasanen et al. [65], the main phases of the constructive research process in the context of management accounting research are:

1. Find a practically relevant problem which also has research potential.
2. Obtain a general and comprehensive understanding of the topic.
3. Innovate, i.e., construct a solution idea.
4. Demonstrate that the solution works.
5. Show the theoretical connections and the research contribution of the solution concept.
6. Examine the scope of applicability of the solution.

Another research method that is constructive by nature is *design science*. Hevner et al. propose the following guidelines for design science in information systems research [59]:

1. *Design as an artefact*: Design-science research must produce a viable artefact in the form of a construct, a model, a method, or an instantiation.
2. *Problem relevance*: The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
3. *Design evaluation*: The utility, quality, and efficacy of a design artefact must be rigorously demonstrated via well-executed evaluation methods.
4. *Research contributions*: Effective design-science research must provide clear and verifiable contributions in the areas of the design artefact, design foundations, and/or design methodologies.
5. *Research rigor*: Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artefact.
6. *Design as a search process*: The search for an effective artefact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
7. *Communication of research*: Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Although neither of the above-discussed approaches to constructive research applies, as such, to development of conceptual modelling languages, the approaches do provide a methodological framework: the constructive approach includes *identifying a relevant research problem* (phases 1 and 2 according to Kasanen et al. [65]; guideline 2 according to Hevner et al. [59]), *constructing a solution to the problem using scientific practices* (3 [65]; 1 and 5 [59]) and *evaluating the solution and showing its theoretical contribution* (4, 5 and 6 [65]; 3, 4 and 7 [59]).

In this dissertation, the relevance of the research problem is mainly motivated by previous research: both software variability and metamodelling languages are established research topics, see Section 1.3.2 for details. In addition, managing software variability is of great practical relevance, as discussed in Section 1.1. The scientific practices used to construct the solutions are iterated in Section 1.3.3. Finally, the theoretical contribution of the constructed conceptual modelling languages is demonstrated by motivating the most important design decisions underlying them and comparing them with previous work in Chapter 6.

1.3.2 Synthetisation

The software variability modelling languages developed in this dissertation synthesise, unify and extend previous approaches to software variability. The synthesised languages are based on the notions of feature and software product family architecture. These notions have attracted most research interest as the conceptual bases for modelling variability and hence provide a solid starting point for the software variability modelling languages developed in this dissertation.

In addition, the software variability modelling languages incorporate a number of ideas originating from the product configuration domain [110, 48], where variability has been studied in the domain of non-software, mainly mechanical and electronics, products. More specifically, these ideas help to add conceptual rigour to previous software variability modelling languages.

The metamodelling languages developed in this dissertation synthesise modelling constructs found in previous conceptual modelling languages, such as UML (Unified Modeling Language) [119, 121], ER (entity-relationship) modelling [33] and Telos [87]. In addition, the metamodelling languages incorporate a number of recent ideas, most importantly, strict metamodelling [6], distinction between ontological and linguistic instantiation [9, 69], unified modelling elements [7] and deep instantiation [4, 11] and the role data model [14] dating back to 1977.

1.3.3 Practices

As discussed in Section 1.2, this dissertation emphasises the *abstract syntax* and *formal semantics* of the conceptual modelling languages developed in it. Abstract syntax is favoured over concrete syntax due to the fact that the interest in this dissertation lies in the conceptual basis of the modelling languages; a concrete syntax involves details such as keywords and ordering of language elements irrelevant from this point of view and may draw attention away from the essence [116].

Giving a modelling language a formal semantics brings a number of benefits. First, without a semantics, a language only amounts to a collection of notations. Such a collection may well be useful for communication purposes, but may lead into disputes over the proper usage and interpretation of the notations [46]. Although the semantics of a modelling language would be intuitively clear, it may still not be precise enough to enable rigorous reasoning and implementing model transformations required, e.g., in a model-driven approach to software development.

The importance of a formal semantics is also emphasised by the large number of papers formalising parts of UML, e.g., [46, 78, 21, 131, 71]. Giving a modelling language under development a formal semantics may help detect errors and omissions in its specification: as an example, the authors of Telos report that "ambiguities and inconsistencies were discovered during the process of constructing a formal account of the language" [87]. Finally, it is particularly useful to give a formal semantics to a metamodelling language: the domain-specific languages defined by metamodels expressed in that language are given at least a partial semantics, as demonstrated in Chapter 5.

On the other hand, it is not the case that a language with a formal semantics would in general be unacceptable to users [55]. Instead, such a language can be made accessible through a suitable concrete syntax or supporting tools. This is an approach sometimes referred to as "logic through the backdoor" in artificial intelligence research [107].

This dissertation follows the *knowledge representation hypothesis* [106, 29]. According to the hypothesis, knowledge must be represented explicitly and declaratively using some logic-like language: explicitness implies that knowledge is represented in a direct and unambiguous way; declarativeness means that the semantics of the representation is specified without reference to how the knowledge is applied procedurally.

1.4 Scope

As suggested by the detailed research questions and discussed in Section 1.3.3, the focus of this dissertation lies in conceptual basis and formal semantics of conceptual modelling languages. Consequently, other aspects related to language definition, most importantly concrete syntax, pragmatics and tool support, are given little or no emphasis. Also, approaches to implementing variability similar to, e.g., generative programming [37] and the approaches suggested by Svahnberg et al. [114] and Santos et al. [100], as well as case studies with industrial software product families are outside the scope of this dissertation.

The software variability modelling languages developed in the dissertation are based on the notions of feature and software product family architecture. Conversely, other concepts that could have served as a basis for such languages are left out: for instance, behavioural aspects of software are not addressed. Also, the software variability modelling languages developed do not include issues such as evolution [76, 44], model metrics [132, 19] and staged configuration [38, 40] deemed important by a number of researchers.

As demonstrated in Chapter 5, the metamodelling languages are designed to ensure their suitability for modelling software variability, which may affect their applicability in other domains. On the other hand, the languages do not explicitly commit to concepts stemming from the software variability domain but instead, as discussed in Section 1.3.2, are based on established conceptual modelling languages (UML, ER modelling and Telos) and a number of recent ideas. Still, the languages exclude a number of issues considered important in previous research, such as class-valued attributes, a theory of parts and wholes [15] or a notion of time.

1.5 Contribution

The contribution of this dissertation is organised around three software variability modelling languages (Koalish [I], Forfamel [II] and Kumbang [III]) and a metamodelling language, Nivel [IV]. In the terminology of the constructive research discussed in Section 1.3, these languages play the role of *solution*.

The three software variability modelling languages are based on different concepts and are defined abstract syntax using progressively sophisticated methods. Lessons learnt in developing Koalish [I] were taken into account when developing Forfamel [II]; Kumbang [III] embodies lessons learnt from both Koalish and Forfamel. On the other hand, all three languages have in common that they are given a formal semantics by translation to WCRL (Weight Constraint Rule Language) [105] which enables automated and other forms of reasoning about the languages.

Koalish [I] is an extension of Koala [130, 128, 129], a component model and an architecture-description language developed at Philips Consumer Electronics. The component diagram notation of UML, since version 2.0, is based on concepts similar to those underlying Koala, which further supports the practical relevance of Koala. Koalish extends Koala through a number of variability mechanisms. Most importantly, the definition of the compositional structure of components in Koalish may include a number of possible types for the part and the number of components occurring as a part may be specified using a cardinality; Koala only allows a single possible type of which exactly one instance must always occur as a part. Similarly, it is possible to define two or more alternative types for an interface in Koalish, whereas Koala only allows a single type. In addition, Koalish abstracts a number of variability mechanisms defined in Koala in terms of implementation. The concrete syntax of Koalish is given using the extended Backus–Naur form (EBNF); the abstract syntax of Koalish is defined only implicitly as a by-product

Forfamel [II] is a feature modelling language that builds on the foundation of the first feature modelling language, FODA (Feature-Oriented Domain Analysis) [63] and includes a number of extensions, such as feature cardinalities [36, 32] including or-features [37] and group cardinalities [39] as special cases, and attributes [36, 40, 38]. Unlike previous feature modelling languages, Forfamel is based on a distinction between *feature types* occurring in *feature models* describing software product families and their instances, *features*, occurring in *configurations*, each describing an individual product in the product family. Consequently, Forfamel is closer to

conceptual modelling languages including the notion of instantiation, such as UML and ER modelling. In addition, FORFAMEL strives to add conceptual clarity and semantic rigour to previous feature modelling languages. The abstract syntax for FORFAMEL is given using a UML class diagram.

KUMBANG [III] unifies KOALISH and FORFAMEL and hence architecture- and feature-based variability modelling, respectively, into a single language: the variability of a software product family can be modelled from two points of view simultaneously. In addition, the interrelations between these points of view can be specified using *implementation constraints*, intuitively stating what is required of an architecture to deliver a certain feature. In spite of the possibility of relating the two views using *implementation constraints*, the two views are independent of each other in that they are governed by their internal well-formedness and consistency rules; satisfying the implementation constraints cannot result in either of the views becoming internally inconsistent. An abstract syntax for KUMBANG is defined using a UML profile, which gives insight to the applicability of the UML profile mechanism to language definition.

In the context of this dissertation, the primary purpose of NIVEL [IV] is to serve as a language for defining the abstract syntax of software variability modelling languages. The suitability of NIVEL for this purpose is demonstrated in Chapter 5 where $\text{KUMBANG}_{\text{NIVEL}}$, an alternative definition of KUMBANG using NIVEL, is given. $\text{KUMBANG}_{\text{NIVEL}}$ is more compact and easily understandable than the definition of KUMBANG given in [III]. $\text{KUMBANG}_{\text{NIVEL}}$ also shows that software variability modelling can be seen as an application area for metamodelling languages in addition to, or even instead of, a domain of research of its own.

NIVEL also contributes to the theory of conceptual modelling languages, especially that of metamodelling. NIVEL is based on concepts—class, instantiation, association, generalisation, attribute and value—found in many previous conceptual modelling languages, such UML and ER modelling. These concepts are generalised to enable modelling on any number of levels in a uniform manner. In addition, NIVEL incorporates a number of recent ideas including strict metamodelling [6], distinction between ontological and linguistic instantiation [9], unified modelling elements [7] and deep instantiation [4, 11]. A formal semantics enabling automated and other forms of reasoning is given for NIVEL by translation to WCRL.

1.6 Outline of the dissertation

The remainder of this dissertation is structured as follows. An overview of the literature relevant to this dissertation is provided in Chapter 2. The software variability modelling languages KOALISH, FORFAMEL and KUMBANG are defined and given a formal semantics in Chapter 3, followed by an overview of NIVEL in Chapter 4. Chapter 5 introduces $\text{KUMBANG}_{\text{NIVEL}}$, an alternative definition of KUMBANG given in terms of NIVEL. Next follow a discussion and comparison with previous work (Chapter 6) and an outline for further work (Chapter 7). Some concluding remarks are provided in Chapter 8.

2 Review of the literature

This chapter provides an overview of the literature relevant to this dissertation. First, a brief introduction to software product families is given. Thereafter, the notions of feature and product family architecture and languages for modelling software variability based on them are discussed in Sections 2.2 and 2.3, respectively. Modelling in general and in software engineering in particular are covered in Sections 2.4 and 2.5. Metamodelling is studied in Section 2.6 followed by overviews of a number of metamodelling languages in Section 2.7. Section 2.8 gives an introduction to product configuration domain, where variability of (non-software) products has been studied. The chapter is concluded in Section 2.9 with an introduction to WCRL, a knowledge representation language that will be used to give formal semantics to conceptual modelling languages developed in subsequent chapters.

2.1 Software product family

This section introduces the concept of *software product family*, or *line*, an alternative term used. Software product families are an important means for implementing software variability. There are a number of definitions for the concept [23, 34]. Clements and Northrop define the concept as follows [34]:

> A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Bosch defines the concept somewhat differently [23]:

> A software product line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using the mentioned reusable assets

The two definitions share the notion of a set of reusable or core assets. Also, both definitions include the set of products or systems developed using these assets. The definition by Clements and Northrop identifies satisfying "the specific needs of a particular market segment" as a defining characteristic of a software product family, whereas Bosch's definition emphasises the role of product line (family) architecture.

The activities related to software product family engineering are typically organised into two phases, namely the *development* and *deployment* [23]; other authors use the terms *domain* and *application engineering* to refer to roughly the same phenomena, respectively [37]. During the development process, the software product family architecture and components implementing the common part, i.e., components present in all individual products in the product family, are implemented. The

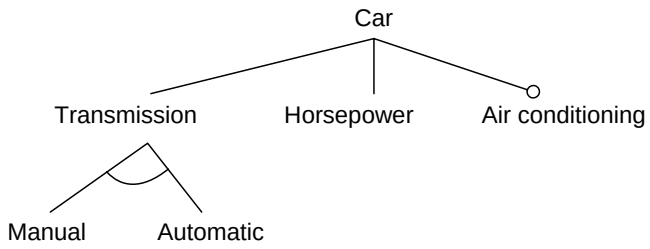

Figure 2.1: A sample FODA model, adapted from [63]

deployment process involves deriving individual products from the product family, based on a set of specific market or customer requirements. The architecture and components from the above-discussed development process form the basis for the deployment process: they are adapted to match the requirements for the individual product being deployed. However, in a typical case, product-specific code must be developed to implement product-specific requirements that are not covered by the assets from the development process.

2.2 Feature modelling

In this section, a brief overview of feature modelling is provided. The section begins with an introduction to FODA [63], the first feature modelling language reported, followed by a discussion on its most important extensions.

In FODA, a feature is defined as an attribute of a system that directly affects end users [63]. Later, the definition of feature has been extended to "a system property that is relevant to some stakeholder" [40].

The variability of a software product family can be represented by organising the common and variable features of the family into a *feature model*. Figure 2.1 illustrates a sample feature model in FODA. Syntactically, a feature model is a tree the root of which is termed the *root feature*, sometimes referred to as *concept*; in the model of Figure 2.1, the root feature is *Car*. The root feature may have other features as its *subfeature* and these may, in turn, have other features as their subfeature, etc.

There are a number of subfeature kinds: a *mandatory* subfeature (*Transmission* and *Horsepower* in the sample model) must be selected whenever its *parent* (*Car*) is selected; an *optional* feature (*Air conditioning*) may, but needs not, be selected whenever its parent is selected. An *alternative* subfeature consists of a set of features of which exactly one must be selected whenever the parent feature is selected; in Figure 2.1, *Manual* and *Automatic* form an alternative subfeature of *Transmission*.

An individual product is described by a *configuration* consisting of a set of features; the product is said to *deliver* the features in the configuration. A *valid configuration* of a feature model consists of a subset of the features in the feature model and obeys the rules defined in the feature model, most importantly the above-described

rules for selecting subfeatures. The task of finding a feature configuration matching a specific set of requirements for an individual product at hand is termed *configuration task*. Note that with the exception of the root feature, all other features in a valid configuration must be subfeatures of some other feature. This requirement will be referred to as the *groundedness property of feature modelling*.

A number of feature modelling languages extend FODA. The most important extensions include directed-acyclic-graph (DAG)-formed feature models, feature cardinalities, attributes and or-features, or more generally, group cardinalities.

In some feature modelling methods, feature models and configurations may take the form of a DAG [64, 99, 36, 41] , i.e., a feature may be a subfeature of more than one feature. This possibility is referred to as *reference attribute* [36] or *feature reference attribute* [41]. It is said that the two or more parents *share* the subfeature.

A *feature cardinality* [36, 32] specifies how many times a subfeature may and must appear in a valid configuration. A mandatory feature corresponds to cardinality 1, and an optional subfeature to cardinality 0..1. Other cardinalities, such as 2..4 (must appear at least two and at most four times), and 1..* (at least once), can be defined.

In some languages, features may have *attributes* [36, 40, 39]. An attribute is a value characterising a feature. There are different variants of attributes: a feature may be restricted to have at most one attribute [40] or any number of attributes may be allowed [36].

An *or*-feature [37] is a subfeature kind similar to an alternative feature, with the difference that at least one of the alternatives must be selected. More generally, a *feature group* consists of two or more features out of which a number specified by the *group cardinality* must be included in each valid configuration. As an example, an alternative feature corresponds to a group cardinality of 1 and an or-feature to a group cardinality of 1..*.

In order to distinguish feature groups from other subfeatures, the term *solitary subfeature* is used to refer to a subfeature that is not a member of a feature group [39] and the terms *feature cardinality* and *group cardinality* when referring to cardinalities in the context of solitary feature and feature group, respectively.

Feature modelling languages have been given formal semantics by translation to various languages, such as propositional logic [75, 17, 133, 42], binary decision diagrams [124, 42], constraint programming [19, 20], higher-order logic [62] and grammars [39, 17].

2.3 Modelling product family architectures

In this section, an overview of the notion of software architecture and languages for representing software architecture is provided. Thereafter, languages for modelling software product family architectures are discussed.

The level of design concerning the overall structure of software systems is commonly referred to as the *software architecture* level of design. This level includes structural issues such as the organisation of a system as a composition of components, the protocols for communication, the assignment of functionality to design elements, etc. [50]. A definition of software architecture is [16]:

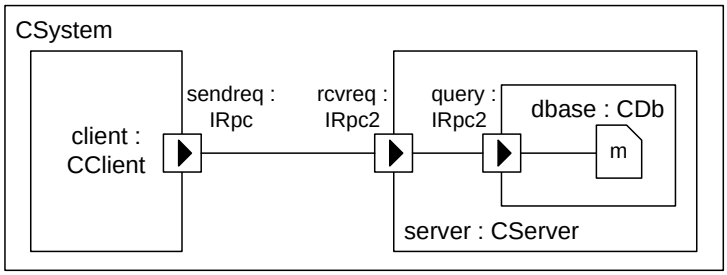

Figure 2.2: The architecture of a client-server system represented using Koala.

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relations among them.

A large number of languages for modelling software architecture, or *architecture-description languages* (ADL) have been reported: Wright [3], Acme [51], C2 [79] and Rapide [73] are examples of ADLs focusing on different aspects of architectural description. ADLs have been surveyed by Medvidovic and Taylor [80]. Most ADLs employ *component* as the basic concepts for describing software architecture; a component is typically defined as a locus of computation. Further, most ADLs include concepts for describing the connection points of components, typically termed *port*, and entities mediating communication between components, termed *connector*.

Most ADLs are intended for describing the architecture of a single system and thus lack concepts for modelling variability. Hence, most of them are not directly applicable to modelling product family architecture. However, there are exceptions to this rule, including Koala [130], and xADL 2.0 [123] that will be discussed next.

2.3.1 Koala

Koala [130, 128, 129] is a component model and an ADL developed at Philips Consumer Electronics in order to enable embedded software to be built more efficiently. Figure 2.2 contains an example of an architecture description in Koala.

A *component* is defined as "an encapsulated piece of software with an explicit interface to its environment" [130]. The interface of a component consists of one or more interfaces. An *interface type* is a "small set of semantically related functions" [130]. A function corresponds to a function signature, e.g., in the C programming language. Each interface of a component is *typed* and identified by a *name*. An interface type a is termed a *subtype* of interface type b if (and only if) the set of functions in a is a superset of those in b; similarly, b is termed a *supertype* of a.

An interface has a *d irection* based on which it is termed either a *required* or a *provided* interface[1]: a required (provided) interface signals that the component

[1]The terms *requires* and *provides interface* are used instead of *required* and *provided interface*, respectively, by van Ommering et al. [130].

requires (provides) the service described by the interface type from (to) its environment.

Both required and provided interfaces are identified by a *name*. In Figure 2.2, the component type *CClient* requires an interface named *sendreq* of type *IRpc*; interfaces of type *IRpc2* are provided by the names *rcvreq* and *query* by the component types *CServer* and *CDb*, respectively.

A component may have other components as its *part* or, in other words, *contain* other components. A part is identified by a *name* that describes the role of the contained component within the containing component, i.e., *whole*. In Figure 2.2, component type *CSystem* contains components of type *CClient* and *CServer* by the names *client* and *server*, again respectively. *CServer* further contains a component of the type *CDb* by the name *dbase*. A part is encapsulated in the sense that it can only be accessed from within the containing component.

The unit of implementation in Koala is *module*. For example in Figure 2.2, the interface *query* is implemented in module *m*.

There may be *connections* between pairs of interfaces. A provided interface in a part may be *connected* to a provided interface in its whole: in Figure 2.2, the interface *rcvreq* is connected to the interface *query*. It is said that the connection is *from rcvreq to query*; the terms *source* and *target* interface are likewise used, respectively. In a similar vein, a required interface in a part may be connected to an interface of its whole. Finally, a required interface may be connected to a provided interface, given that the components with the two interfaces are parts of the same component, as is the case for interfaces *sendreq* and *rcvreq* in Figure 2.2.

A *configuration* is defined as a component that is not part of any other component and has no interfaces. In a valid configuration, the connections must obey a set of rules referred to as the *Koala connection constraints* in this dissertation: each interface that is not implemented in a module must be connected to exactly one other interface according to the above-described rules. Further, in each connection, the target interface must be a subtype of the source interface.

Koala incorporates a number of variability mechanisms. A configuration is characterised by a set of *parameters*. Configurations with different parameters are considered different: the functionality of a system may directly vary based on the parameters. Further, parameter values may affect which interfaces are connected. An interface may be declared *optional*, implying that the component with the interface may, but needs not to, have the interface when instantiated in a configuration.

The semantics for Koala is given in terms of the outputs of tools operating on the language.

2.3.2 xADL 2.0

xADL 2.0 [43, 123] is an infrastructure for rapid development of ADLs based on the Extended Markup Language (XML). Hence, xADL 2.0 is not merely an ADL, but provides facilities for defining customised ADLs. In this dissertation, the focus is set on the conceptual core of xADL 2.0; the extension mechanisms are ignored.

The basic modelling elements of xADL 2.0 include *component type*, *connector type* and *interface type*. A component may be composed of other components. An

interface is a connection point of a component. Interfaces may be *bound* using a connector. The variability modelling elements of xADL 2.0 include *optional element*, *variant type* and *optional variant element*. An optional element has the intuitive semantics that it may be, but is not required to be, included. A variant type pertains to a choice between two or more elements. An optional variant element is both an optional element and a variant type. The control over whether to include an optional element and which variant should be selected is in *Boolean guards*, which are Boolean expressions on a number of variables.

2.4 Modelling

In this dissertation, a definition of *model* by Stachowiak [111] is adopted. According to the definition, a model must possess three features:

1. *Mapping feature*: A model is based on an original.
2. *Reduction feature*: A model only reflects a (relevant) selection of an original's properties.
3. *Pragmatic feature*: A model needs to be usable in place of an original with respect to some purpose.

As suggested by the definition and discussed by Kühne [69, 68], being a model or an original, or *system* [69] or *system under study* [104, 53], alternative terms used, are not intrinsic properties of entities but roles played by them in a binary relationship.

Models are expressed using a *modelling language*. The term *language element* will be used to refer to entities constituting a modelling language. Examples of language elements include *proposition* and *connective* in propositional logic and *entity* and *relationship* in ER modelling. Entities constituting models are referred to as *model elements*.

A distinction between the *abstract* and *concrete syntax* of modelling language is made. The terms are defined in a similar manner as has been done for programming languages [116]: it is customary to use Backus-Naur Form (BNR) in the definition of concrete syntax. Abstract syntax, on the other hand, pertains to the definition of the conceptual basis of a language. Hence, the abstract syntax of a language ignores issues related to represention, such as the choice of particular keywords and delimiters, and the ordering of language elements.

Besides syntax, the *semantics* of a modelling language is likewise of interest. Intuitively, semantics pertains to the meaning of a modelling language. There are several styles of giving a language a semantics [116]. In general, the definition of a semantics includes specifying a *semantic domain* and a *semantic mapping* [56]. The semantic domain may be a mathematical domain, as is the case in *denotional semantics*, or another language, as is the case in *translational semantics* [116]. The semantic mapping is a function relating the syntactic elements of the modelling language with the elements of the semantic domain. The term *formal semantics* is

used if the semantic domain is either a mathematical domain or a language with a formal semantics, e.g., propositional logic.

Finally, in spirit of Stachowiak's pragmatic feature, the potential utility of models and modelling language as a means for expressing models is adopted as an underlying guideline for this dissertation. As phrased by Box and Draper [28, p. 424]: "Essentially, all models are wrong, but some are useful."

2.5 Modelling in software engineering

Modelling plays a prominent role in software engineering. A large number of different approaches to modelling have emerged, each supported by different forms of languages, processes and tools. In this section, a brief overview of the different approaches is provided.

One way of classifying models is to distinguish between *descriptive* and *prescriptive model* [74], or *forward-looking* and *backward-looking model* [53], roughly alternative terms used. A descriptive model "mirrors an existing original" and a prescriptive one is "used as a specification of something to be created" [74]. Just as being a model is not an intrinsic property of an entity, being a descriptive or descriptive model is not an intrinsic property of an entity, but of the relationship between a model and an original. The same entity may play both a prescriptive and descriptive role, possibly at different times; such a model is termed *transient* [74].

Although being prescriptive or descriptive is not a property of a model, let alone of a modelling language or paradigm, some approaches to modelling can be characterised either as prescriptive or descriptive. For example, in the context of object-oriented software development, in *object-oriented analysis* there is "an emphasis on finding and describing the objects—or concepts—in the problem domain whereas in *object-oriented design*, there is "an emphasis on defining software objects and how they collaborate to fulfil the requirements" [72]. Hence, object-oriented analysis can be characterised as descriptive and design as prescriptive.

The *conceptual modelling* initiative defines a level of system description that is "closer to the human conceptualisation of a problem domain" and can "serve as knowledge bases to be used by an expert system and as starting points for a system implementation" [31]. Further, "the descriptions that arise from conceptual modelling activities are intended to be used by humans, not machines" [86]. Hence, conceptual modelling can be characterised as predominantly descriptive.

On the other hand, *model-driven engineering*, or alternatively, *model-driven development* (MDD), is commonly defined as "the systematic use of models as primary engineering artefacts throughout the engineering lifecycle". Model-driven engineering has emerged as an approach supplementary to third-generation programming languages as a means for constructing concrete software [101, 49]. Modelling languages have been envisaged as making the current, third-generation programming languages obsolete in a manner similar to that in which compilers largely removed the need for writing assembler or binary code [8]. The Model Driven Architecture (MDA) [84] initiative of the OMG (Object Management Group) is an example of

an approach to model-driven engineering. To summarise, model-driven engineering represents a predominantly prescriptive modelling approach.

The modelling languages developed in this dissertation are termed *conceptual* in reference to the ideas of prescriptive model and conceptual modelling.

2.6 Metamodelling

This section begins with a discussion of the notion of metaness. The main body of the section consists of an overview of different ideas related to metamodelling.

2.6.1 Metaness

The notion of *metaness* often occurs when modelling is discussed, especially in the software engineering context. In this dissertation, a characterisation of metaness by Kühne [69] is adopted. The characterisation is based on a binary relation R and the relation R^n defined for $n \geq 1$ as:

$$e_1 R^n\, e_2 = \begin{cases} e_1\, R\, e_2 & \text{if } n = 1 \\ \exists\, e : e_1\, R^{n-1}\, e \wedge e\, R\, e_2 & \text{if } n \geq 2 \end{cases}$$

A *relation suitable for building metalevels* is required to satisfy three properties: the *acyclic*, *anti-transitive* and *level-respecting* properties [69]. The level-respecting property implies the anti-transitive property [69] and the acyclic property, see Appendix B of [IV]. Hence, the level-respective property defined as [69]

$$\forall n, m : (\exists\, e_1, e_2 : e_1\, R^n\, e_2 \wedge e_1\, R^m\, e_2) \rightarrow n = m$$

suffices to characterise a relation suitable for building metalevels.

Given a relation R suitable for building metalevels, element e_1 can be characterised as *meta* with respect to e_2 if (and only if) $e_1 R^n e_2$ for $n \geq 2$. For example, if R is the binary relation $modelOf(m, o)$, m is termed a *model of original* o and $modelOf^2(m, o)$ implies that m is a *metamodel* of o. Further, each element m corresponds to a *level*, or *metalevel* or *layer*, alternative terms used. The level corresponding to m is said to be *above* the level corresponding to o whenever $modelOf^p(m, o)$ for $p \geq 1$; conversely, the level corresponding to o is said to be *below* the one corresponding to m. Finally, the terms *top* and *bottom* are used to refer to levels above and below all other levels, respectively.

If R is the $instanceOf(i, t)$ relation, i is said to be an *instance of* t and t a *type* of i. Finally, if $instanceOf^p(i, t)$, i is termed an *instance of order* p *of* t. Instances of order 2 or higher are collectively termed *higher-order instances*.

The above characterisation of metaness is in accordance with a number of definitions of metamodel, such as "a model of models" [84]. As previously suggested by Kühne [69], in this dissertation metamodels are not exclusively regarded as definitions of domain-specific languages, although they can in some circumstances be interpreted as such; other authors have adopted the opposite view [104, 119, 95].

A language suitable for expressing model hierarchies consisting of three or more models will be termed a *metamodelling language*. The more generic term *metamodelling framework* will be used to refer to methodologies suitable for expressing model hierarchies consisting of at least three models that may consist of more than one individual language, e.g., the OMG framework consisting of UML [119, 121], the Meta Object Facility (MOF) [81] and the Object Constraint Language (OCL) [92].

2.6.2 Strict metamodelling

The *strict metamodelling rule* has been formulated as follows [6]:

> In an n-level modeling architecture, $M_0, M_1, \ldots, M_{n-1}$, every element of an M_m-level model must be an instance-of exactly one element of an M_{m+1}-level model, for all $m < n - 1$, and any relationship other than the instance-of relationship between two elements X and Y implies that $level(X) = level(Y)$.

In effect, the rule requires that the instance-of relation may not cross levels in the sense that a (model) element would be an instance of another element two or more levels above. Also, the rule requires an element, except one on the M_0 level, to be an instance of a single element only; that is, multiple classification is not allowed.

2.6.3 Ontological and linguistic instantiation

Many metamodelling frameworks are based on a single form of instantiation. For example, in the OMG framework, each level, except the top, is characterised as an instance of the level next above it and the top level, containing MOF, as an instance of itself. Language elements, such as *Metaclass*, *Class* and *Object*, are on the top level, as illustrated in Figure 2.3 (a).

However, it has been argued that resorting to a single form of instantiation leads to a number of problems, such as *dual classification* [7], also termed *ambiguous classification* [4], and class/object duality [7]. As an example of dual classification, *Collie* in Figure 2.3 (a) is classified as both as an instance of *Class* and *Breed*; as another example, *Lassie* is both an instance of *Object* and *Collie*. Further in Figure 2.3 (a), class/object duality is exemplified in *Collie* that is both an instance of *Breed* and the class of *Lassie*.

As a remedy against problems arising in metamodelling frameworks based on a single form of instantiation, it has been suggested that metamodelling frameworks should distinguish between two forms of instantiation, termed the *logical* and *physical* dimension [7, 10], or alternatively, *ontological* and *linguistic instantiation* [9, 69]. In this dissertation, the latter set of terms is adopted.

As the term *linguistic* suggests, linguistic instantiation pertains to the (abstract syntax of the) language used to *represent* model elements or, in other words, the form model elements take [4, 7]. In terms of model and language elements as defined and discussed in Section 2.4, linguistic instantiation is a relation between model and language elements where the former play the role of instance and the latter that of

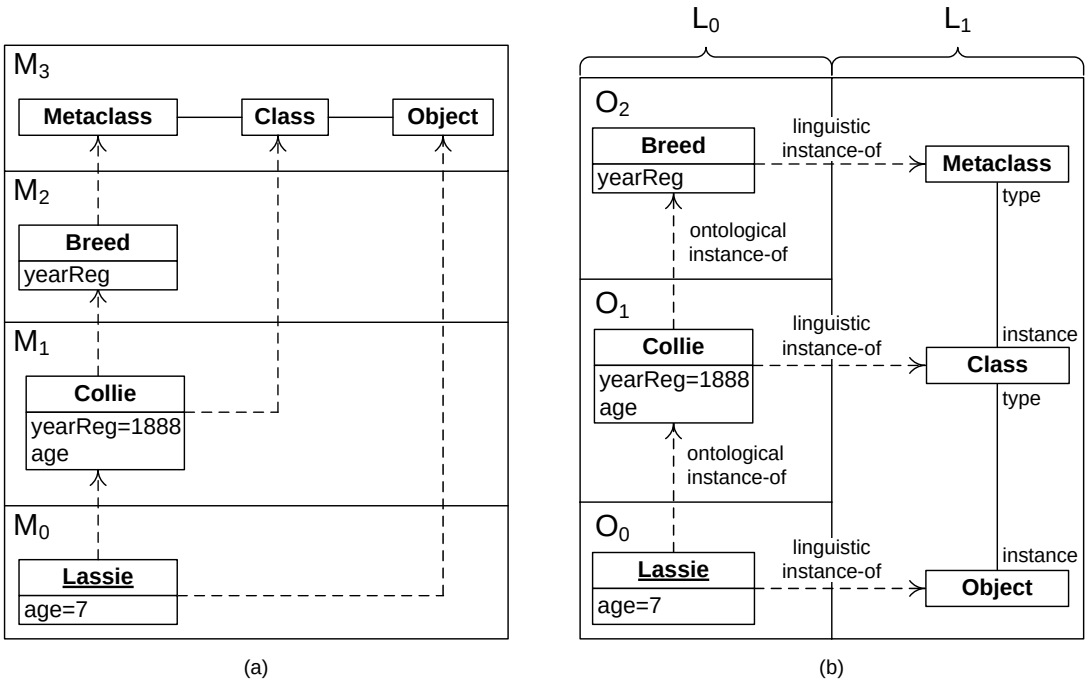

Figure 2.3: (a) A metamodelling framework based on a single form of instantiation. (b) Ontological and linguistic instantiation. Adapted from [69].

class or type. An ontological instantiation relationship relates two models or model elements that are in the same domain of discourse, e.g., biology or retail sales, but on different levels [69].

Figure 2.3 (b) illustrates the distinction between ontological (dashed vertical arrows) and linguistic instantiation (dashed horizontal arrows). In the figure, *Lassie* is an ontological instance of *Collie* and a linguistic instance of *Object*. *Collie*, on the other hand, is an ontological instance of *Breed* and a linguistic instance of *Class*. *Breed* is a linguistic instance of *Metaclass*.

Note that ontological instantiation, either between levels or individual model elements, must be represented using some language element. However, not all conceptual modelling languages provide a construct for this purpose. For example, in UML class diagrams it is not possible to express that a class is an instance of another class, although it is possible to include *InstanceSpecification*s representing the instances of classes in models; similarly in ER modelling, it is not possible to express that an entity or relationship is an instance of another entity or relationship, respectively, although the materialization

2.6.4 Unified modelling elements

In a multilevel modelling framework, model elements on a level are typically represented using the same language element: for example, in Figure 2.3 (b), level O_0 is

associated with *Object*, level O_1 with *Class* and level O_2 with *Metaclass*. Should another ontological level be added, a new language element would need to be introduced, probably termed *Metametaclass* in this case. Alternatively, if it is not possible to add language elements when needed, the maximum number of available ontological levels is determined by the set of language elements at hand [7].

The notion of *unified modelling elements* is based on the observation that model elements on intermediate levels, i.e., other than the top and bottom level, are uniform in the sense that they exhibit both characteristics of classes and objects. Consequently, the model elements can be represented by a single language element, termed the *unified modelling element*[2]. Further, model elements on the top and bottom levels can be viewed as special cases: top-level elements with a degenerate object and bottom-level elements with a degenerate type facet. In order to assign model elements to levels, the unified modelling element includes an integer attribute for level. [7]

In Figure 2.3 (b), O_1 is an intermediate level and the class *Collie* is both an instance of *Breed* and the type of *Lassie*. This figure can be redrawn, employing the notion of unified modelling elements, as Figure 2.4 (a). In the figure, the unified modelling element *ModelElement* unifies *Metaclass*, *Class* and *Object* in Figure 2.3 (b). The model elements *Breed*, *Collie* and *Lassie* are all instances of *ModelElement*.

2.6.5 Deep instantiation

The situation in which a model element may have higher-order instances is referred to as *deep instantiation* as opposed to the case of *shallow instantiation* where only first-order instances can be represented [4, 7].

The maximum order of instances of a class is specified by its *potency* [4, 7]. Potency is a non-negative integer assigned independently for each class that is not an instance of another class. An instance of a class has the potency value of its type decremented by one. However, the potency value may not be less than zero. Hence, the potency of a class gives the maximum order of its instances. A class under the notion of shallow instantiation is of potency 1.

Under deep instantiation, it is useful to enable a model element to characterise its higher-order instances. As an example, in Figure 2.4 (b) the attribute denoted age^2 implies that each second order instance of *Breed*, e.g., *Lassie*, has a value for *age*; the intuition is that every dog must have an age. More generally, each attribute of a class is assigned an integer value termed *potency* that resembles the potency assigned for classes in that an instance of the class has an attribute by the same name as its type but with the potency decremented by one; an attribute of potency one turns into a value in an instance. [4, 7]

In Figure 2.4 (b), the attribute *age* of *Breed* is of potency 2. Further, by virtue of being an instance of *Breed*, *Collie* has an attribute by the same name but of potency 1. Finally, a value (7) is assigned for *age* in *Lassie*.

[2]Note that the term *unified modelling element* adopted from [7] is somewhat incompatible with the terminology of *model* and *language element* adopted in this dissertation.

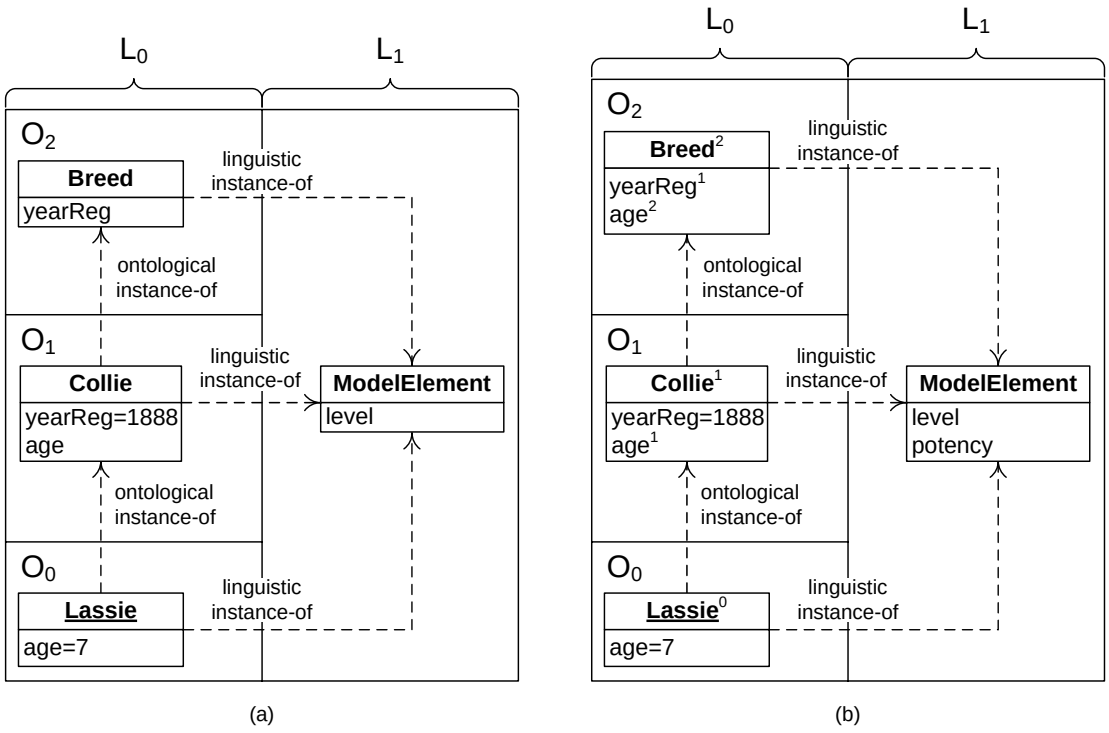

Figure 2.4: (a) Unified modelling elements. (b) Deep instantiation.

The ability of classes to characterise their higher-order instances need not be restricted to attributes. Instead, the notion of deep instantiation can be extended to cover, e.g., associations and constraints. Thus extended, a metamodel can be used to define a semantics for a (domain-specific) modelling language in addition to abstract syntax. The notion of deep instantiation will be extended as outlined above in Nivel [IV] and the applicability of Nivel to software variability modelling languages will be demonstrated in Section 5 of [IV] and in Chapter 5.

2.7 Metamodelling languages and frameworks

In this section, an overview of a number of metamodelling languages and frameworks is provided.

2.7.1 UML

UML is often used to define language metamodels. A kind of metamodel can be defined using a *profile* consisting of a number of *stereotypes*. Each stereotype defines an extension that can be applied to one or more metaclasses, such as *Class*. A stereotype may define additional properties and constraints that become properties and constraints of the instances of a metaclass to which the stereotype is applied.

To construct a language metamodel using profiles and stereotypes, one needs to map the language elements to UML metaclasses. That is, for each language element, one must find the UML metaclass that best matches the intended abstract syntax and semantics of the language element and create a stereotype extending that metaclass. Note that the abstract syntax and semantics of a UML metaclass may only be restricted by stereotypes. Hence, the abstract syntax for the selected UML metaclass must contain a superset of features intended for the language element, e.g., associations from the language element to other elements, and the semantics of the UML metaclass must be less restrictive than that of the language element.

Finally, if a suitable mapping between the language elements and UML metaclasses is found, one needs to define suitable properties and constraints for the stereotypes to achieve the desired semantics. There is no standard way for defining the constraints, although the UML specification itself applies OCL [92]. Note also that UML itself has no standard formal or otherwise rigorous semantics that would enable automated or otherwise formal reasoning about the language. Consequently, no semantics is obtained for the profile or the language defined using the profile, besides the intuitive descriptions given in the UML standard [121].

The profile mechanism is characterised as a "lightweight" and "not a first-class" extension mechanism [121, pp. 647–648] and the Meta Object Facility (MOF) [81] as the mechanism for first-class extensibility.

2.7.2 MOF

MOF is endorsed by the OMG as a metamodelling facility providing support for any number of *layers* [81, p. 8]:

> Note that key modeling concepts are *Classifier* and *Instance* or *Class* and *Object*, and the ability to navigate from an instance to its metaobject, i.e., its classifier. This fundamental concept can be used to handle any number of layers, sometimes referred to as metalevels. The MOF 1.4 Reflection interfaces allow traversal across any number of metalayers recursively.

Unfortunately, the MOF specification [81] is, at best, ambiguous and vague in specifying an abstract syntax and a semantics for the language elements (*Classifier* and *Instance*) referred to in the quotation above. Also, the specification does not provide examples of how MOF is applied in metamodelling.

2.7.3 Telos

Telos [87] is a language that can be used to represent knowledge using any number of metalevels. Elements on different levels are represented uniformly as *individuals* that may likewise uniformly be described using *attributes*. Individuals and attributes are collectively termed *propositions*.

Propositions are organised based on three *dimensions*: aggregation, generalisation and classification. *Aggregation* is used to collect attributes related to a proposition into *objects*. *Generalisation* pertains to organising propositions into generalisation hierarchies. *Classification* refers to the requirement that each proposition

is an *instance of* one or more propositions, or classes. The instance-of relation is used to characterise propositions using various terms: a *token* has no instances and is intended to represent concrete entities in the domain of discourse, a *simple class* with only tokens as instances, a *metaclass* with only simple classes as instances, a *metametaclass* with only metaclasses as instances etc. Each such term defines a *plane* of which there may be an unbounded number. A class that may have instances along more than one such plane is termed an *ω-class*. [87]

The classes constituting the definition of Telos, e.g., the above-described classes *Individual*, *Attribute* and *Proposition*, are included in the classification hierarchy and serve as examples of ω-classes.

2.8 Product configuration

This section provides an overview of product configuration research, a subfield of artificial intelligence [47].

Research in the product configuration domain is based on the notion of *configurable product*: a configurable product is a product such that each individual product is adapted to the requirements of a particular customer order. Historically, the configurable products studied in the domain have been non-software products, typically mechanical and electronics ones. A modular structure is typical of a configurable product: individual product consist of pre-designed components and selecting different components leads to different product variants. [109]

The possibilities for adapting a configurable product are predefined in a *configuration model* that explicitly and declaratively defines the set of legal products. A specification of an individual product, *configuration*, is produced in the *configuration task* based on the configuration model and a set of customer requirements.

There are two widely cited conceptualisations of configuration knowledge [110, 48]. The most important concepts in these are: components, ports, resources, and functions. A *component* represents a distinguishable entity in a product: a configuration is composed of components that may be composed of other components. A *port* is a connection interface of a component. Ports may be *connected* with each other. A component may produce and consume *resources*; the production and consumption of each resource must be balanced. Finally, a *function* is an abstract characterisation of a product that, e.g., a customer could use to describe it.

2.9 Weight constraint rule language

In this section, a brief overview of Weight Constraint Rule Language (WCRL) [105] is provided. The overview provided in this section is based on Section 3 of [IV].

WCRL is a general-purpose knowledge representation language syntactically similar to logic programs with a declarative formal semantics based on the stable model semantics of logic programs: the rules of the program are interpreted as constraints on a solution set for the program, whereas the usual logic programming paradigm is based on a goal-directed backward chaining query evaluation [88].

2.9.1 Syntax of weight constraint rules

A *weight constraint rule* is an expressions of the form

$$C_0 \leftarrow C_1, \ldots, C_n$$

where each C_i is a *weight constraint*. A weight constraint is of the form:

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq U$$

where each a_i and b_j is an *atom*. Atom a and not-atom $not\ b$ are called *literals*. Each literal is associated with a weight: e.g., w_{a_1} is the weight of a_1. The numbers L and U are the *lower* and *upper bound* of the constraint, respectively.

A number of notational shorthands are used. Constraints where every weight has value one, i.e., constraints of the form

$$L \leq \{a_1 = 1, \ldots, a_n = 1, not\ b_1 = 1, \ldots, not\ b_m = 1\} \leq U$$

are termed *cardinality constraints* and written as

$$L\,\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}\,U$$

A missing lower bound is interpreted as $-\infty$ and a missing upper bound as ∞. A rule of the form

$$\{a_1, \ldots, a_n\} \leftarrow C_1, \ldots, C_n$$

is termed a *choice rule*. Constraints of the form $1 \leq \{l = 1\}$, where l is a literal, are written simply as l.

The shorthand

$$\leftarrow C_1, \ldots, C_n$$

is used for rules where the head C_0 is an unsatisfiable constraint, such as $1 \leq \{\}$. This kind of rules are termed *integrity constraints*. Finally, rules with an empty body, that is, of the form

$$C_0 \leftarrow$$

are termed *facts*.

2.9.2 Stable model semantics

A set of literals S is defined to *satisfy* a weight constraint if (and only if):

$$l \leq \sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \leq u$$

That is, a weight constraint is satisfied if the sum of weights of the literals satisfied by S is between the lower and upper bounds. A weight constraint rule is *satisfied* by a set of atoms S if and only if the head C_0 is satisfied whenever each constraint C_i in its body is satisfied. A *program* P is defined as a set of weight constraints. P is satisfied by S if and only if every weight constraint rule in P is satisfied by S.

A set of atoms S is a *stable model* of program P if (i) S satisfies P and (ii) S is *justified* by P. Intuitively, S is justified or *grounded* by P if every atom a in S has a non-circular justification. An atom is justified by a program if it appears in the head of a satisfied rule. Non-circular justification requires in addition that the body of the justifying rule is satisfied without assuming a.

Example Consider the program:

$$0 \leq \{a = 1, b = 1\} \leq 2 \leftarrow a$$

The set of atoms $\{a\}$ satisfies the only rule in program, and thus the program itself. However, the atom a only has a circular justification: the body is obviously not satisfied without assuming a. Hence, $\{a\}$ is not a stable model of the program. The only stable model is the empty set.

2.9.3 Rules with variables

For practical purposes, WCRL has been extended with a form of *variables* and *function symbols*. In more detail, *domain-restricted rules* with variables are allowed. Intuitively, a domain-restricted program P is divided into two parts: P_{Do} containing the definition of *domain predicates* and P_{Ot} containing all the other rules. The form of rules in P_{Do} is restricted in such a way that P_{Do} has a unique finite stable model. All the rules in P_{Ot} must be domain-restricted in the sense that every variable occurring in a rule must appear in a domain predicate which occurs positively in the body of the rule. The domain predicates in P_{Do} are defined using stratified rules allowing a form of recursion [115].

A rule with variables is treated as a shorthand for all its ground instantiations with respect to the *Herbrand universe* of the program, i.e., the set of atoms that can be composed by applying functional composition from the basic symbols. The ground instantiation contains all the rules that can be obtained by substituting each variable in the rule with one of its possible values, restricted by one or more domain predicates. The atoms occurring in ground rules thus formed are termed the *Herbrand base* of the program and a model consisting of such atoms a *Herbrand model*.

Example The use of variables and domain predicates can be demonstrated using the structure of a company board as an example. The people and their sexes that can possibly serve in a board are represented using facts such as $female(alice) \leftarrow$ and $male(bob) \leftarrow$. Values such as *alice* and *bob* are termed *object constants* in this dissertation and must begin with a lower-case letter. The rules

$$person(X) \leftarrow female(X)$$
$$person(X) \leftarrow male(X)$$

provide the definition for the unary domain predicate *person*. Variables, such as X in the above rules, must begin with an upper-case letter.

The predicate $member(x)$ gives that x is a board member. The fact

$$5\,\{member(X) : person(X)\}\,7 \leftarrow$$

gives that a board consists of at least five and at most seven members. The fact is also an example of a rule including a *conditional literal*. A conditional literal is of the form $l : d$, where l is any predicate and d is a domain predicate. When instantiated, a conditional literal corresponds to the sequence of literals l' obtained by substituting the variables in l by all the combinations allowed by the domain predicate d. For example, with the above-mentioned facts, the rule would be instantiated into

$$5\,\{member(alice), member(bob)\}\,7 \leftarrow$$

which is obviously not satisfied.

The fact that a board has a chair and may have a vice-chair is captured by the following two facts, respectively:

$$1\,\{chair(X) : person(X)\}\,1 \leftarrow$$
$$0\,\{vice(X) : person(X)\}\,1 \leftarrow$$

The rules

$$member(X) \leftarrow chair(X), person(X)$$
$$member(X) \leftarrow vice(X), person(X)$$

state the requirement that both the chair and the vice-chair are also board members.

Finally, an integrity constraint can be used to rule out the possibility that the chair and the vice-chair are the same person:

$$\leftarrow chair(X), vice(X), person(X)$$

Note that the literal $person(X)$ is required to make the above rule domain-restricted.

2.9.4 Computational complexity and implementation

Given a ground WCRL program P, i.e., one not containing variables, and a set of atoms S, deciding whether S is a stable model of P can be decided in polynomial time. Deciding whether program P has a stable model is **NP**-complete.

An inference system *smodels*[3] implements WCRL and has been shown to be competitive in performance compared with other solvers, especially in the case of problems involving structure [105].

[3]See `http://www.tcs.tkk.fi/Software/smodels/`

3 Software variability modelling languages

This chapter is the first of the three chapters describing the artefacts constructed in this dissertation. In this chapter, an overview of three software variability modelling languages, Koalish [I], Forfamel [II] and Kumbang [III], are provided. In addition, a formal semantics for the languages is given by translation to WCRL. Examples of the languages can be found in the original publications [I], [II] and [III].

The chapter is organised as follows. Next, in Section 3.1, follows a discussion on the levels of abstraction the software variability modelling languages are based on. The formalisation principles applied in the translation to WCRL are presented in Section 3.2. The main body of the languages will be discussed in the remaining sections. As the three languages share many concepts and are formalised analogously, the discussion is organised based on the language elements, such as taxonomy of composable types, compositional structure etc. Differences between the languages are discussed separately for each language element in their respective sections.

3.1 Levels of abstraction

The software variability modelling languages are based on three levels of abstraction illustrated in Figure 3.1. The highest level is the *metalevel* containing the metamodel for the languages. The next level is the *model level* on which variability models are located. A *variability model* is a representation of the variability in a software product family. In the terminology of Section 2.1, a variability model is typically constructed in the development phase or during domain engineering. The third and lowest level termed *instance level* contains *configurations*. Intuitively, a configuration represents an individual product. Finding a configuration matching a particular set of requirements for an individual product at hand is a problem related to the deployment phase, or application engineering. A variability model defines a possibly empty set of configurations that are *valid* with respect to the model.

The metalevel and the model level, and the model level and the instance level are related to each other in a similar manner: in both cases, the former can be characterised as a *model of* the latter. Further, in both cases, entities on the former, e.g., *Server* and *Database* in Figure 3.1, can be termed *types* of the latter and entities on the latter, e.g., *:Server* and *:Database* can be termed *instances* of the entities on the former. The terms *type* and *instance* are also used as absolute terms when referring to entities on the type and instance levels, respectively. Finally, a relationship between types on the model level is termed a *definition*.

3.2 Formalisation principles

A formal semantics for the software variability modelling languages is given by translation to WCRL [105]. The approach followed can be termed *translational semantics* [82]: the syntactic constructs of a language are mapped onto constructs of another language, one with a semantics defined. This mapping is denoted by

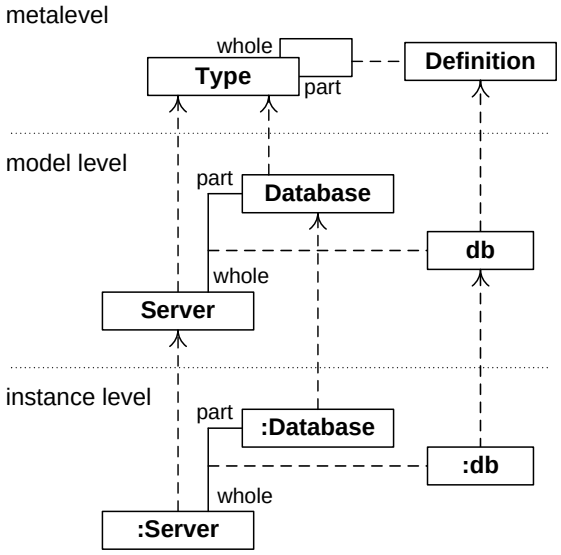

Figure 3.1: Three levels of abstraction in the software variability modelling languages KOALISH, FORFAMEL and KUMBANG

the symbol $t_{\mathcal{L}}$, where $\mathcal{L} \in \{\text{KOALISH}, \text{FORFAMEL}, \text{KUMBANG}\}$. Further, in the terminology of [56], the semantic mapping is $t_{\mathcal{L}}$ and the semantic domain is WCRL.

In symbols: $t_{\mathcal{L}} : \mathcal{V}_{\mathcal{L}} \mapsto \mathcal{W}$, where $\mathcal{V}_{\mathcal{L}}$ denotes the set of syntactically well-formed variability models in language $\mathcal{L}$ and $\mathcal{W}$ the set of WCRL programs. Note that both the mapping and the set of syntactically valid models are different for each of the variability modelling languages.

The formal semantics captures the notion of a *valid configuration*. The mapping $t_{\mathcal{L}}$ is constructed in a way such that each stable model of $t_{\mathcal{L}}(M)$, where $M \in \mathcal{V}_{\mathcal{L}}$, corresponds to a valid configuration of M.

The WCRL program $t(M)$ can be partitioned into two parts: an *axiomatic part* $t_{\mathcal{L},a}$ and a *model-specific* part $t_{\mathcal{P}}(M)$ related to a particular model M. The axiomatic part $t_{\mathcal{L},a}$ contains rules shared by all models $M \in \mathcal{V}_{\mathcal{L}}$. Rules in the model-specific part and axiomatic part will be termed *model-specific* and *axiomatic rules*, respectively. Axiomatic rules will be $\boxed{\text{boxed}}$ to distinguish them from the model-specific rules. When introducing model-specific rules, phrases such as "for each component type t and part definition d" will be used to make explicit the context in which the rule is introduced. Note that the translation is *modular* in the sense that model elements can be translated independently of each other.

The rules in the model-specific part $t_{\mathcal{P}}(M)$ can further be decomposed into two parts: a part encoding the types and definitions appearing in a particular variability model and a part specifying a large enough number of instances of each type to cover all the valid configurations of M.

A number of syntactic conventions are adopted. As required by the WCRL syntax, variables begin with an upper-case and object constants with a lower-case

letter. Names of non-domain predicates are printed in ***bold***. For some predicates, both a domain and a non-domain variant are used; in these cases, the latter will be identified by a lower-case subscript p for "possible", e.g., $hasPart_p$.

3.3 Definition of abstract syntax and main language elements

The syntax of each of the three software variability modelling languages is defined using a different method in the respective original publications. As can be recalled from Section 1.2, the focus of this dissertation is on the abstract (as opposed to concrete) syntax of the conceptual modelling languages developed. Consequently, the syntax of the variability modelling languages will mostly be discussed at an abstract level.

For KOALISH, no explicit abstract syntax is given in [I]. Instead, a concrete syntax is given using EBNF, see Figure 3.2. Being based on Koala [130], the basic language elements of KOALISH include *component* that may contain other components as its *parts*. In addition, components may have connection points termed *interface*. A distinction between types and instances is made both for components and interfaces.

The abstract syntax of FORFAMEL is defined in [II] using a UML class model, see Figure 3.3. The most important language element in FORFAMEL is *feature*. A feature may have *subfeatures*. The subfeatures of a feature give its compositional structure; it is also said that a feature is *composed of* other features. A distinction between *feature types* and their instances, *features*, is made.

A UML profile is constructed in [III] to define an abstract syntax and a form of semantics for KUMBANG. The profile, including the constituting stereotypes and taxonomic relations between them, is illustrated in Figure 3.4. Note that the profile, as illustrated in Figure 3.4, does not itself define an abstract syntax or any form of semantics for KUMBANG but does so only when considered in conjuction with the relevant elements in the UML metamodel (*Classifier*, *Property*, etc.). KUMBANG synthesises KOALISH and FORFAMEL. Consequently, its main language elements include both component, interface and feature. Just as in KOALISH and FORFAMEL, a distinction between types and instances is made for all main language elements.

In the remainder of this chapter, component and feature types will collectively be referred to as *composable types*.

3.4 Taxonomy of composable types

Composable types, excluding component types in KOALISH, may be organised in generalisation hierarchies using the taxonomic *isa* relation. For types t_{sub} and t_{super}, $isa(t_{sub}, t_{super})$ implies t_{sub} is a *subtype of* t_{super} and t_{super} is a *supertype of* t_{sub}.

A composable type may be *abstract*. An abstract type may not have direct instances or, in other words, an instance of an abstract type must always be an instance of a subtype of the type.

Semantics Formal semantics for the software variability modelling languages will be given under headings such as this. The semantics will be given separately for each

⟨*koalish model*⟩ ::= **Koalish model** ⟨*identifier*⟩ **root component** ⟨*identifier*⟩ ⟨*type definition*⟩*
⟨*type*⟩ ::= ⟨*component type*⟩ | ⟨*interface type*⟩ | ⟨*attribute type*⟩
⟨*component type*⟩ ::= **component type** ⟨*identifier*⟩ **{** ⟨*section*⟩* **}**
⟨*interface type*⟩ ::= **interface type** ⟨*identifier*⟩ **{** (⟨*identifier*⟩ **;**)* **}**
⟨*attribute type*⟩ ::= **attribute type** ⟨*identifier*⟩ **= {** ⟨*valuelist*⟩ **}**
⟨*section*⟩ ::= ⟨*attribute section*⟩ | ⟨*provides section*⟩ | ⟨*requires section*⟩ | ⟨*constraint section*⟩ | ⟨*contains section*⟩
⟨*contains section*⟩ ::= **contains** ⟨*part definition*⟩*
⟨*attribute section*⟩ ::= **attributes** ⟨*attribute definition*⟩*
⟨*provides section*⟩ ::= **provides** ⟨*interface definition*⟩*
⟨*requires section*⟩ ::= **requires** ⟨*interface definition*⟩*
⟨*connects section*⟩ ::= **connects** ⟨*connection*⟩*
⟨*constraint section*⟩ ::= **constraints** ⟨*constraint*⟩*
⟨*part definition*⟩ ::= ⟨*identifier list*⟩ ⟨*identifier*⟩ [⟨*cardinality*⟩] **;**
⟨*attribute definition*⟩ ::= ⟨*identifier*⟩ ⟨*identifier list*⟩ **;**
⟨*interface definition*⟩ ::= ⟨*identifier*⟩ ⟨*interface details*⟩ (**,** ⟨*interface details*⟩)* **;**
⟨*interface details*⟩ ::= ⟨*identifier*⟩ (**optional** | **grounded**) [(**optional** | **grounded**)]
⟨*connection*⟩ ::= ⟨*identifier*⟩ [**.** ⟨*identifier*⟩] **=** ⟨*identifier*⟩ [**.** ⟨*identifier*⟩] **;**
⟨*cardinality*⟩ ::= [⟨*integer literal*⟩ [**-** ⟨*integer literal*⟩]]
⟨*identifier list*⟩ ::= ⟨*identifier*⟩ | **(** ⟨*identifier*⟩ ⟨*identifier list end*⟩* **)**
⟨*value list*⟩ ::= ⟨*value*⟩ (**,** ⟨*value*⟩)*
⟨*value*⟩ ::= ⟨*integer literal*⟩ | ⟨*identifier*⟩

Figure 3.2: The concrete syntax of KOALISH in EBNF, excluding the syntax for constraints. Adapted from [I].

language element in their respective sections; this is possible, as the semantics for different language elements are modular in the sense that rules formalising different language elements do not interfere with each other. For example, in this section the semantics for taxonomy of component types will be given.

A type in a variability model is represented using a predicate symbol; Table 3.1 contains a summary of how different model elements in variability models and configurations are represented in WCRL. The *unique name assumption* is adopted: each type is represented by exactly one predicate symbol and a predicate symbol may not represent more than one type; the same assumption is adopted for other language elements as well. The predicate symbol representing type t will be simply t. An instance is represented by an object constant.

For each pair of types (t_{sub}, t_{super}) in the *isa* relation, introduce the rule:

$$t_{super}(I) \leftarrow t_{sub}(I)$$

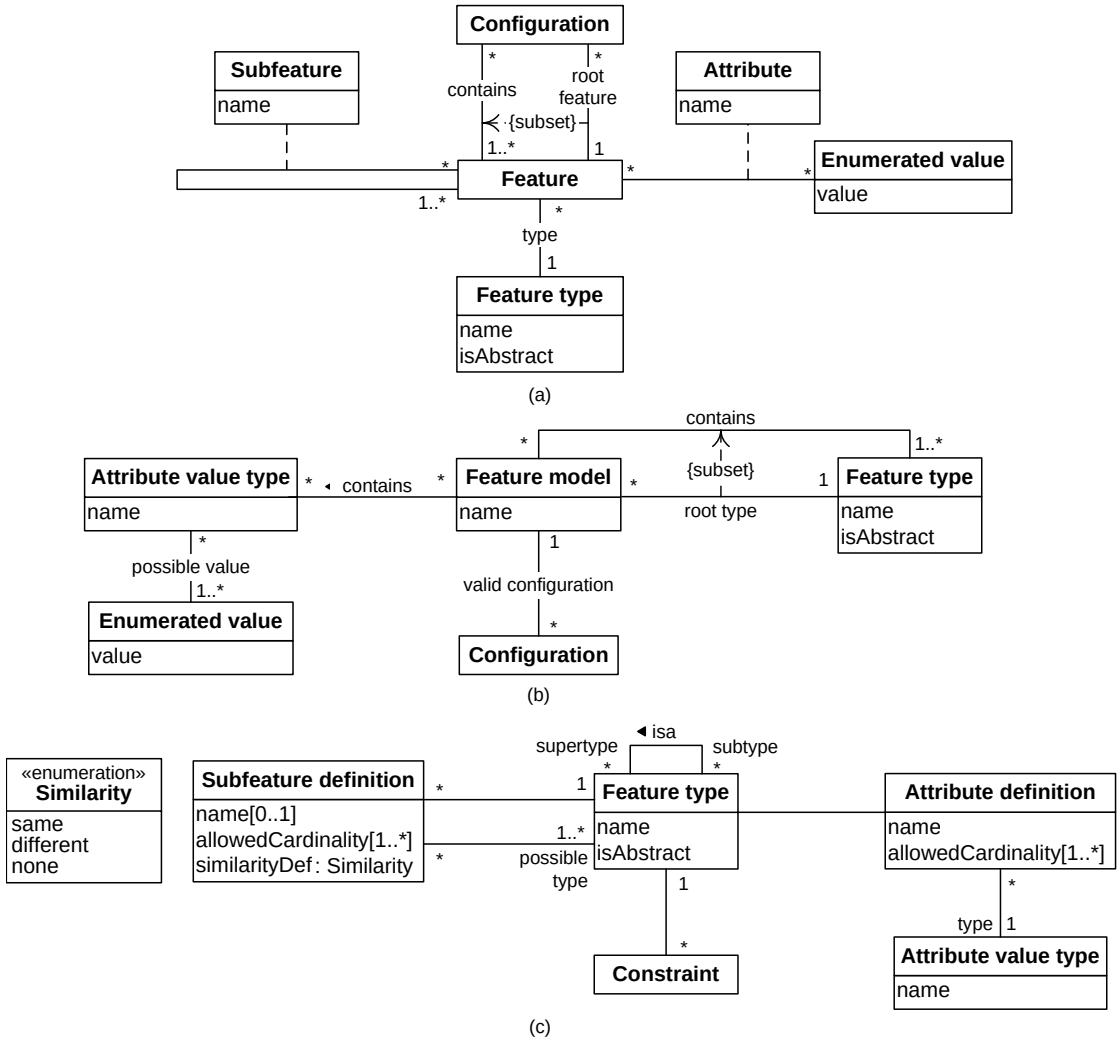

Figure 3.3: The abstract syntax of FORFAMEL given as a UML class model [II].

3.5 Compositional structure

Composable instances, i.e., components and features, may be characterised using a *compositional structure*, i.e., components (features) may have other components (features) as their part. The compositional structure is specified using *part definitions* in composable types. A part definition relates a *whole type* and a number of *possible part types*. In addition, a part definition contains a *part name*, a *cardinality* and a *similarity value*[4], with the exception that a part definition in KOALISH does not contain a similarity value.

The part name identifies the role in which instances of the possible part types

[4]The term *definition* was used instead of *value* in the original publications [I], [II] and [III] when referring to properties of part and interface definitions.

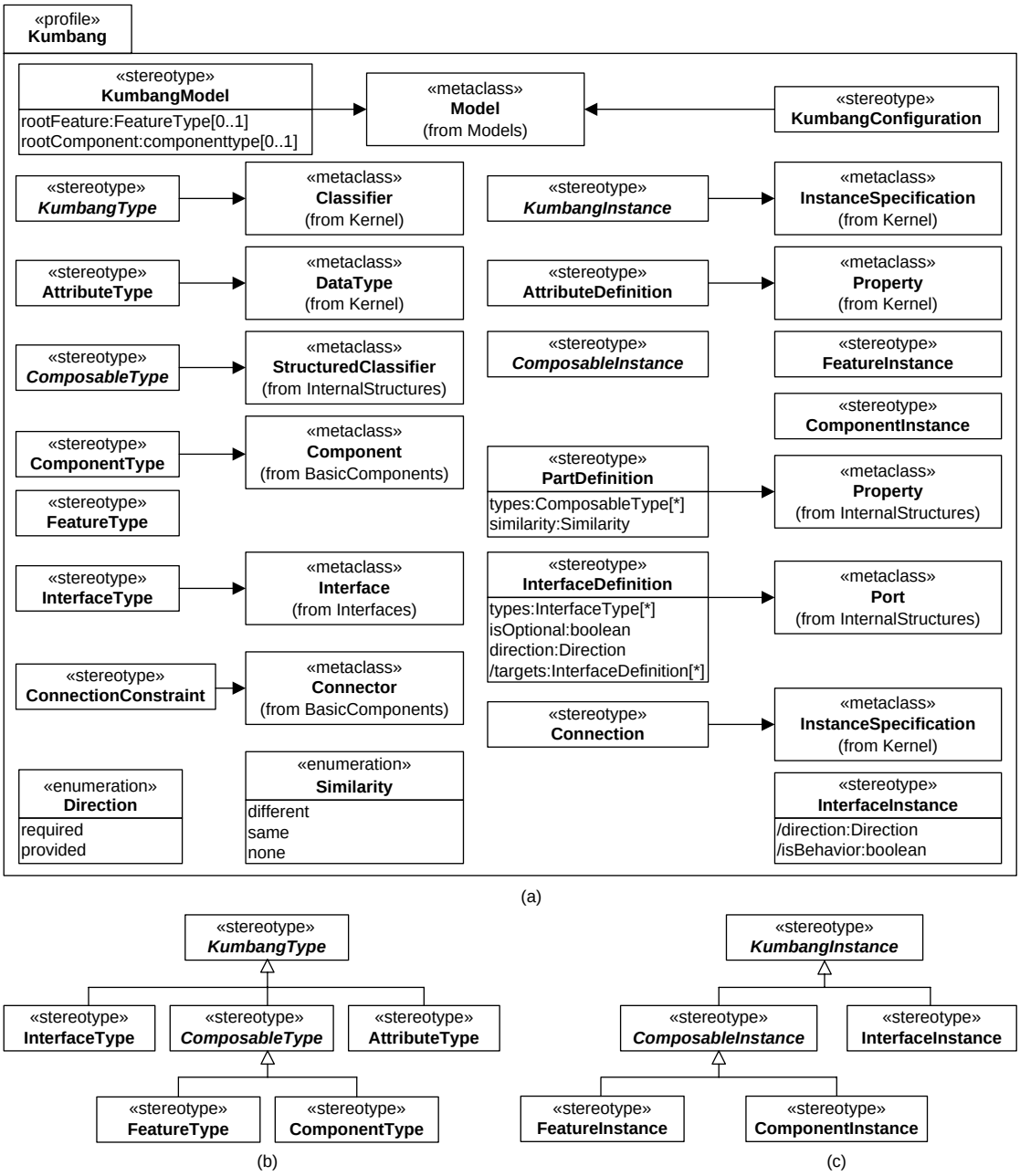

Figure 3.4: The absract syntax of KUMBANG profile given as a UML profile [III].

are parts of instances of the whole type. The cardinality is an integer range $[L..U]$ with the semantics that a whole in a valid configuration must have at least L and at most U parts by the part name.

The *similarity* value is one of *same*, *different* and *none*. The value *same* implies that the parts of each whole in a valid configuration must be of the same type, whereas *different* implies they must be of different types. If the value is *none*, no

Table 3.1: Representation of language elements in the software variability modelling languages in WCRL

Model element	WCRL representation
Type t	Unary predicate symbol $t(i)$: i is an instance of t
Instance i	Object constant i
Attribute type a	Unary predicate symbol $a(v)$: v is a value in a
Possible attribute value e	Object constant e
Function f	Object constant f

restrictions on the respective types are implied.

In a variability model, exactly one component and feature type must be defined as the *root type*, with the exception that if the model does not include component or feature types, no respective root type need or may be defined. A valid configuration must contain exactly one instance of each root type, termed *root instance*, and any other instance in the configuration must be a part of a whole. It is said that an instance is *justified* by either being the root instance or a part of a whole.

Semantics A valid configuration must contain exactly one instance of each root type. That an instance i is justifiably *in* configuration is represented by the unary predicate $\boldsymbol{in}(i)$. See Table 3.2 for an overview of the predicates used in the formalisation. The predicate $root_{\mathcal{K}}(i)$ gives that instance i is an instance of the root type for $\mathcal{K} \in \{\mathcal{C}, \mathcal{F}\}$, where $\mathcal{C}$ stands for "component" and $\mathcal{F}$ for "feature".

A valid configuration must contain, for each $\mathcal{K}$, exactly one instance of the root type:

$$\boxed{1\,\{\boldsymbol{in}(I) : root_{\mathcal{K}}(I)\}\,1 \leftarrow} \tag{3.1}$$

For each root type t of the kind $\mathcal{K}$, a rule of the form

$$root_{\mathcal{K}}(I) \leftarrow t(I)$$

is introduced.

The ternary predicate $\boldsymbol{hasPart}(w, p, n)$ gives that instance w has instance p as its part with the part name n. For each part definition with part name n in composable type w with cardinality $[L, U]$, introduce a rule:

$$L\,\{\boldsymbol{hasPart}(W, P, n) : hasPart_p(W, P, n)\}\,U \leftarrow \boldsymbol{in}(W), w(W) \tag{3.2}$$

Further, if the *similarity* value is *same*, the parts must be of the same type. Therefore, for each subset $\{p_1, p_2\}, p_1 \neq p_2$ of the set of possible part types in the part definition, introduce a rule of the form:

$$\begin{gathered} \leftarrow \boldsymbol{hasPart}(W, P_1, n), hasPart_p(W, P_2, n), \\ \boldsymbol{hasPart}(W, P_1, n), hasPart_p(W, P_2, n), w(W), p_1(P_1), p_2(P_2) \end{gathered} \tag{3.3}$$

Table 3.2: Predefined predicates used in the formalisation of the software variability modelling languages in WCRL. Predicates for which a corresponding domain predicate is defined, denoted by subscript p, have the column p checked.

Predicate	p	Semantics
$\boldsymbol{in}(i)$		The instance i is justified in the configuration.
$root_{\mathcal{K}}(i)$		The instance i is the root feature for $\mathcal{K}$.
$\boldsymbol{hasPart}(w, p, n)$	$\times$	Instance w has instance p as its part with name n.
$\boldsymbol{hasAttr}(i, n, v)$		Instance i has value v for attribute name n.
$\boldsymbol{hasInt}(c, i, n)$	$\times$	Component c has interface i with name n.
$prov(i)$		The interface i is a provided interface.
$req(i)$		The interface i is a required interface.
$grounded(i)$		The provided interface i is grounded.
$\boldsymbol{cn}(s, t)$	$\times$	The interface s is connected to interface t.
$pc(s, t)$		The pair of interfaces (s, t) is place compatible.

Finally, if *similarity* takes value *different*, a rule of the form

$$\begin{gathered} \leftarrow \boldsymbol{hasPart}(W, P_1, n), hasPart_p(W, P_2, n), \\ \boldsymbol{hasPart}(W, P_1, n), hasPart_p(W, P_2, n), w(W), p(P_1), p(P_2) \end{gathered} \tag{3.4}$$

is introduced for each possible part type p.

An instance that is a part of a whole is justifiably in the configuration:

$$\boxed{\boldsymbol{in}(P) \leftarrow \boldsymbol{hasPart}(W, P, N), hasPart_p(W, P, N)} \tag{3.5}$$

3.6 Attribute

Instances may be characterised by *attributes*. An attribute is a name–value pair. *Attribute definitions* in types are used to specify which attributes their instances may and must have. An attribute definition consists of a *name* and an *attribute type*[5]. An attribute type is identified by a name and consists of a finite set of possible values an attribute of the type may take.

Semantics An attribute type is represented in WCRL using a predicate symbol and its constituting values by object constants. The predicate $a(v)$ gives that attribute type a contains value v. Hence, for each attribute type a and possible value v, a fact of the form

$$a(v) \leftarrow$$

is introduced.

[5]The term *attribute value type* is used instead of *attribute type* in [II].

Attribute definitions are translated into rules as follows. For each attribute definition of type t with name n and attribute type a, introduce a rule of the form:

$$1\,\{hasattr(I, n, V) : a(V)\}\,1 \leftarrow \boldsymbol{in}(I), t(I)$$

3.7 Interface and connection

A component instance may have *interfaces*. An interface consists of a set of *functions*, each of which is an enumerated value. An interface has a *direction*, either *provided* or *required* and may be *grounded*.

A component type specifies the interfaces its instances may and must have using *interface definitions*. An interface definition consists of an interface *name*, a non-empty set of *possible interface types* and *direction*, *optionality* and *groundedness* values.

The interface *name* specifies the role in which an interface is an interface of the component. The set of possible interface types contains the interface types the instances of which may occur as interfaces of instances of the component type with the interface name.

An *interface type* consists of a set of functions. An interface has the same functions as its type.

Just as an interface is either a provided or a required interface, the *direction* in an interface definition takes one of the values *provided* and *required*. An interface with the name of the definition has the same value for direction as its definition.

The *optionality* value in an interface definition takes one of the values *optional* and *mandatory*. The value *mandatory* implies that an instance of the component type must have an interface with the interface name, whereas the value *optional* implies that an instance may, but needs not, have such an interface. The definition is termed *mandatory* or *optional* based on the optionality value.

The *groundedness* is a Boolean value. An interface based on the definition is termed *grounded* if (and only if) the groundedness value is *true*. A grounded interface corresponds to an interface that is implemented in a module in Koala, see Section 2.3.1.

A valid KOALISH configuration must satisfy the Koala connection constraints.

Semantics In the formalisation, an interface type is represented by a unary predicate symbol. Interfaces instances and functions are represented by object constants.

The predicate $hasFunc(i, f)$ gives that interface i contains function f. For each interface type t and function f contained in t, introduce a rule of the form:

$$hasFunc(I, f) \leftarrow t(I)$$

The ternary predicate $\boldsymbol{hasInt}(c, i, n)$ gives that component c has interface i with name n. For each interface definition with name n in component type t, a rule of the following form is introduced:

$$L\,\{\boldsymbol{hasInt}(C, I, n) : hasInt_p(C, I, n)\}\,1 \leftarrow \boldsymbol{in}(C), t(C)$$

where L is 0 if the definition is optional and 1 otherwise.

The notion of subtyping between interfaces is formalised as follows. The predicate $subtypeOf(i_{sub}, i_{super})$ gives that interface i_{sub} is a subtype of i_{super}, i.e., the functions contained in i_{sub} is a superset of those contained in i_{super}:

$$\boxed{\begin{array}{c} subtypeOf(I_{sub}, I_{super}) \leftarrow not\ notSubtypeOf(I_{sub}, I_{super}), \\ hasInt_p(C_{sub}, I_{sub}, N_{sub}), hasInt_p(C_{super}, I_{super}, N_{super}) \end{array}} \tag{3.6}$$

where the auxiliary predicate $notSubtypeOf(i_1, i_2)$ holds for the pair (i_1, i_2) if i_2 contains a function not contained in i_1:

$$\boxed{notSubtypeOf(I_1, I_2) \leftarrow hasFunc(I_2, F), not\ hasFunc(I_1, F)} \tag{3.7}$$

The relative positions and directions of pairs of interfaces that may be connected according to the Koala connection constraints are illustrated in Figure 3.5, see also Figure 2.2. Such a pair (I_{from}, I_{to}) is termed *place compatible* and the fact that the pair is place compatible is represented by the binary domain predicate $pc(I_{from}, I_{to})$.

The case illustrated in Figure 3.5 (a) is captured by the rule:

$$\boxed{\begin{array}{c} pc(I_{from}, I_{to}) \leftarrow hasInt_p(C_{from}, I_{from}, N_{from}), hasInt_p(C_{to}, I_{to}, N_{to}), \\ hasPart_p(C_{to}, C_{from}, P_{from}), req(I_{from}), req(I_{to}) \end{array}} \tag{3.8}$$

the case in Figure 3.5 (b) by the rule:

$$\boxed{\begin{array}{c} pc(I_{from}, I_{to}) \leftarrow hasInt_p(C_{to}, I_{to}, N_{to}), hasInt_p(C_{from}, I_{from}, N_{from}), \\ hasPart_p(C_{from}, C_{to}, P_{to}), prov(I_{to}), prov(I_{from}) \end{array}}$$

and the case in Figure 3.5 (c) by the rule

$$\boxed{\begin{array}{c} pc(I_{from}, I_{to}) \leftarrow hasInt_p(C_{from}, I_{from}, N_{from}), hasInt_p(C_{to}, I_{to}, N_{to}), \\ hasPart_p(C, C_{from}, P_{from}), hasPart_p(C, C_{to}, P_{to}), C_{from} \neq C_{to}, req(I_{from}), prov(I_{to}) \end{array}}$$

The predicate $\boldsymbol{cn}(i_{from}, i_{to})$ gives that interface i_{from} is connected to interface i_{to}. The corresponding predicate $cn_p(i_1, i_2)$ representing the possibility that the pair of interfaces is defined as:

$$\boxed{cn_p(I_{from}, I_{to}) \leftarrow subtypeOf(I_{to}, I_{from}), pc(I_{from}, I_{to})} \tag{3.9}$$

Intuitively, a pair of interfaces (i_1, i_2) may be bound if the target interface i_2 is a subtype of the source interface i_1 and the pair is place compatible.

Connections between pairs of interfaces are enabled by the rule:

$$\boxed{0\,\{\boldsymbol{cn}(I_{from}, I_{to})\}\,1 \leftarrow \boldsymbol{in}(I_{from}), \boldsymbol{in}(I_{to}), cn_p(I_{from}, I_{to})}$$

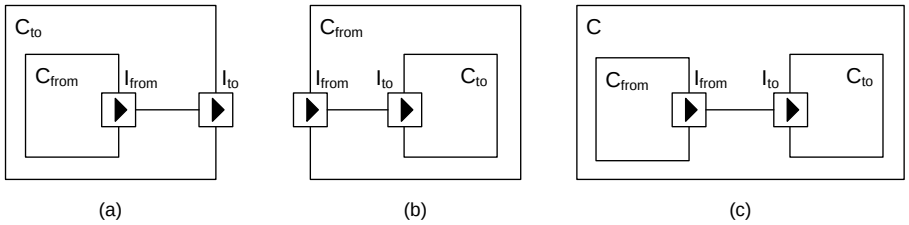

Figure 3.5: Possibilities for connecting interfaces in KOALISH and KUMBANG. (a) Between a pair of required interfaces. (b) Between a pair of provided interfaces. (c) From a required to a provided interface.

Finally, it is required that each non-ground interface must be bound to at least one

$$\boxed{\begin{array}{c}\leftarrow \{\boldsymbol{cn}(I_{from}, I_{to}) : cn_p(I_{from}, I_{to})\}\, 0, \\ hasInt_p(C, I_{from}, N), \boldsymbol{in}(I_{from}), not\ grounded(I_{from})\end{array}} \tag{3.10}$$

and at most one other interface

$$\boxed{\begin{array}{c}\leftarrow 2\,\{\boldsymbol{cn}(I_{from}, I_{to}) : cn_p(I_{from}, I_{to})\}\,, \\ hasInt_p(C, I_{from}, N), \boldsymbol{in}(I_{from}), not\ grounded(I_{from})\end{array}} \tag{3.11}$$

3.8 Constraints

When modelling software variability, it may be the case that not all relevant interrelations between elements can be expressed using the constructs described in previous sections. Therefore, the variability modelling languages include the notion of *constraint*. Constraints can be defined for composable types. Syntactically, a constraint is a Boolean condition that can be evaluated in the context of a composable instance. In a valid configuration, all constraints of all instances must evaluate to *true*.

The variability modelling languages are not committed to any specific constraint language. However, as an example of a suitable constraint language, an overview of *Kumbang constraint language* is given in [III].

In KUMBANG, it is necessary to enable the specification of interdependencies between the two points of view. The related mechanism is a special case of constraints termed *implementation constraints*. Such constraints are attached to feature types and describe what must be true of the architecture in order for the feature to be delivered by a product in the family. Even though the feature and the architecture view may be related through implementation constraints, the two views are still independent of each other in that they are both subject to well-formedness and consistency rules of their own. Consequently, it is not possible that either of the views becomes inconsistent in a valid configuration in order to satisfy one or more implementation constraints.

Input: composable type w_t to be instantiated
Output: set of rules representing w_i, an instance of w_t
let w_i be a new object constant
add fact $w_t(w_i) \leftarrow$
foreach part definition d *of* w_t **do**
  if $d.similarity = different$ **then**
    $maxCard := 1$
  else
    $maxCard := d.card.upper$
  foreach composable type $t \in d.part$ **do**
    foreach $p_t \in t \cup t.subtypes$ **do**
      if $p_t.isAbstract$ **then continue**
      for $j := 1$ **to** $maxCard$ **do**
        $p_i := instantiate(p_t)$
        add fact $hasPart_p(w_i, p_i, d.name) \leftarrow$
foreach interface definition d *of* w **do**
  foreach interface type $i_t \in d.types$ **do**
    let i_i be a new object constant
    add fact $hasInt_p(w_i, i_i, d.name) \leftarrow$
    if $d.direction = provided$ **then**
      add fact $provided(i_i) \leftarrow$
    else
      add fact $required(i_i) \leftarrow$
    if $d.isGrounded = true$ **then** add fact $grounded(i_i) \leftarrow$
return w_i

Figure 3.6: Algorithm $instantiate(t)$ for instantiating a variability model, initially called with each root type as the input parameter in turn. For simplicity, same symbols are used for model elements and the object constants representing them.

3.9 Instantiation

In addition to the above-discussed rules representing variability models, rules defining the elements that may possibly appear in a configuration must be explicitly represented in the WCRL encoding of the variability model; in other words, a variability model must be *instantiated*. The idea is that a number of instances and relationships between them large enough to cover any valid configuration of the variability model are created. In addition, the instances are assigned types and places in the compositional hierarchy using the ternary domain predicates $hasPart_p$ and $hasInt_p$ and direction and possible groundedness is specified for interfaces through the unary predicates *provided*, *required* and *grounded*. An algorithm based on the idea is presented as Figure 3.6; note that the algorithm is *not* about solving the configuration task related to a variability model.

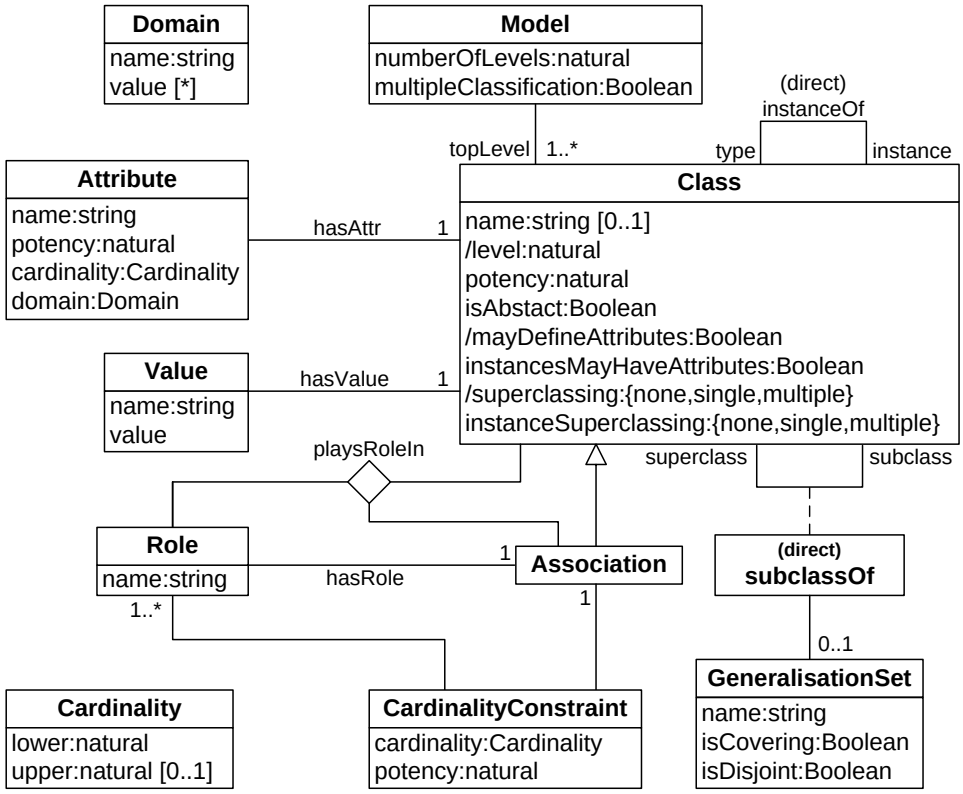

Figure 4.1: The abstract syntax of Nivel given as a UML class model [IV]

4 Nivel—a metamodelling language

Nivel [IV] is a metamodelling language developed as a part of this dissertation. Nivel does not commit to a single modelling paradigm, such as object-orientation, and therefore covers a large variety of different modelling purposes. Nivel is based on a core set of modelling concepts, i.e., class, instantiation, association, generalisation, attribute and value, and incorporates a number of recent ideas including strict metamodelling [6], distinction between ontological and linguistic instantiation [9, 69], unified modelling elements [7] and deep instantiation [4, 11]. Nivel supports modelling on any number of levels in a uniform manner.

The abstract syntax of Nivel is illustrated in Figure 4.1 using the UML class diagram notation. However, the figure is not intended to serve as an exhaustive description of the abstract syntax of Nivel, let alone as a definition of a semantics for Nivel. Instead, the figure is intended to give an intuitive account of the concepts of Nivel to a reader familiar with the UML class diagram notation.

Nivel elaborates on its underlying concepts in various ways. The notion of deep instantiation previously defined for classes and attributes is extended to cover associations. A role in an association may be played by several classes that do not

share a superclass. Generalisations between associations are considered in detail, including cases in which roles are redefined in subclasses. Both multiple classification and restricting to single classification are supported. Classes at a higher level exercise control over their instances in terms of whether and how they may participate in generalisations and define attributes.

4.1 Language elements

In this section, a brief overview of the language elements of NIVEL is provided.

Class is the central language element of NIVEL in the sense all other language elements are directly related to *class*. A class is an entity with an identity that can be used to distinguish a class from other classes. A *model* consists of a set of classes, some of which are *top-level* classes.

Instantiation is a binary relation between classes. If the pair (i, t) is in the relation, class i is termed an *instance of* t, and class t a *type of* i. A restriction to *single classification* can be made, in which case a class not on the top level must be an instance of exactly one class; otherwise such a class must be an instance of at least one type. No type may be defined for a top-level class.

A class may simultaneously be an instance and a type. Hence, being and instance or a type are not intrinsic properties of a class but roles played by a class in the *instanceOf* relation. The *potency* of a class gives the maximum order of its instances.

There can be *generalisations* between pairs of classes. A generalisation is a binary relationship between a pair of classes (a, b), represented using the *subclassOf* relation. The class a is termed a *subclass of* b, and class b a *superclass of* a. An instance of a subclass is also an instance of all the superclasses of the subclass.

A generalisation may belong to a *generalisation set*. All generalisations in a generalisation set must share the same superclass. *Disjointness* and *covering constraints* may be applied to generalisation sets. In the former (latter) case, an instance of the superclass of the generalisation set must be an instance of at most (least) one of the subclasses of the generalisation set.

A class may have *attributes* and named *values*. An attribute describes what values the instances, including higher-order instances, of a class may and must have. The ability to describe higher-order instances is achieved using the *potency* of an attribute: an instance of a class has the same attributes as the class but the potency decremented by one. When a class with an attribute of potency 1 is instantiated, the instance has a corresponding value. Hence, an attribute of potency p describes the instances of order p of the class.

An *association* is a relationship among a set of classes. An association defines a set of roles each of which is played by one or more classes, which is in contrast with associations (relations) in most previous conceptual modelling languages, such as UML and ER modelling, where a role must be played by exactly one class. Also in contrast with such languages, an association in NIVEL is also a class and may hence have higher-order instances. Consequently, an association in NIVEL may specify a relationship of interest between higher-order instances of the participating classes.

Table 4.1: Some predicates used in the WCRL encoding of NIVEL. Symbols in the *Variants* column: 'p' – possible, 'd' – direct, 'D' – declared, 't' – transitive, '$-$' – actual, 'Der' – derived. The semantics are described for the actual variant. Non-domain predicates are printed in ***bold***. Adapted from [IV].

Predicate	Variants	Semantics
class(c)	$-\ p$	Object constant c represents a class
$topLevel(c)$	$-\ D$	Class c is on top level
subclassOf(a, b)	$-\ p\ d\ D$	Class a is a subclass of class b
instanceOf(i, t)	$-\ p\ d\ pd\ D\ Der$	Class i is an instance of class t
$singleClassification$	none	No multiple classification is allowed
$abstract(c)$	none	Class c is abstract
onLevel(c, l)	none	Class c is on level l
hasAttr(c, n, p, d, l, u)	$-\ p\ D$	Class c has attribute named n with potency p, domain d, cardinality lower bound l and upper bound u
hasValue(c, n, v)	$-\ p\ D$	Class c has value v under name n
$association(a)$		Class a is an association
$hasRole(a, r)$	$-\ p\ D$	Association a has role r
playsRoleIn(c, r, a)	$-\ d\ p\ D$	Class c plays role r in association a; (c, r, a) is a roleplay

4.2 Formal semantics

A formal semantics is given for NIVEL by translation to Weight Constraint Rule Language (WCRL) [IV]. The formal semantics enables decidable, automated reasoning about NIVEL. Table 4.1 contains a summary of predicates used in the formal encoding of NIVEL relevant for KUMBANG$_{\text{NIVEL}}$ to be discussed next.

5 Defining Kumbang using Nivel

This chapter shows how KUMBANG can be defined using NIVEL. Intuitively, the definition is given as a mapping from the language elements of KUMBANG, as illustrated in Figure 3.4, to a set of NIVEL elements constituting the top-level of a model. The term $\text{KUMBANG}_{\text{NIVEL}}$ will be used to refer to the resulting NIVEL model.

The structure of this chapter parallels that of Chapter 3. First, the levels of abstraction employed in $\text{KUMBANG}_{\text{NIVEL}}$ are studied in Section 5.1. Thereafter follow sections on the taxonomy of types (Section 5.2), compositional structure (Section 5.3) and attribute (Section 5.4). Interface and connection are the topic of Section 5.5. The chapter is concluded in Section 5.6 with a discussion on the instantiation of model elements that may possibly appear in a valid configuration.

5.1 Levels of abstraction

$\text{KUMBANG}_{\text{NIVEL}}$ consists of three levels. The top level, i.e., level 0, is illustrated in Figure 5.1. Variability models are represented by the first-order instances of *KumbangModel* and located on level 1. Configurations are represented by the second-order instances of *KumbangModel* and are thus on level 2. The elements constituting variability models and configurations are discussed in the subsequent sections.

5.2 Taxonomy of composable types

Figure 5.1 (a) illustrates the taxonomy of classes used to represent types and instances in KUMBANG. The root of the taxonomy, *KumbangType*, is an abstract class of potency 2. The first-order instances of *KumbangType* represent types in variability models and the second-order instances represent instances in configurations.

Of the two direct subclasses of *KumbangType*, *InterfaceType* is a concrete class that represents the idea of an interface. *ComposableType* is abstract and represents the idea of a composable entity. The concrete subclasses of *ComposableType*, *ComponentType* and *FeatureType*, represent components and features, respectively.

Any number of superclasses may be defined for a composable type, which is enabled by setting the attribute *instanceSuperclassingMultiple* of *ComposableType* to *true*, as denoted by the two-headed arrow in the illustration of the class in Figure 5.1 (a).

5.3 Compositional structure

The representation of compositional structure in $\text{KUMBANG}_{\text{NIVEL}}$ is illustrated in Figure 5.1 (c). The core construct is the abstract binary association *hasPart* with roles *whole* and *part*, both played by *ComposableType*. The first-order instances of the association represent part definitions and the second-order instances part-whole relationships in configurations. The name, cardinality and similarity value of a part definition are represented as attributes of *hasPart*.

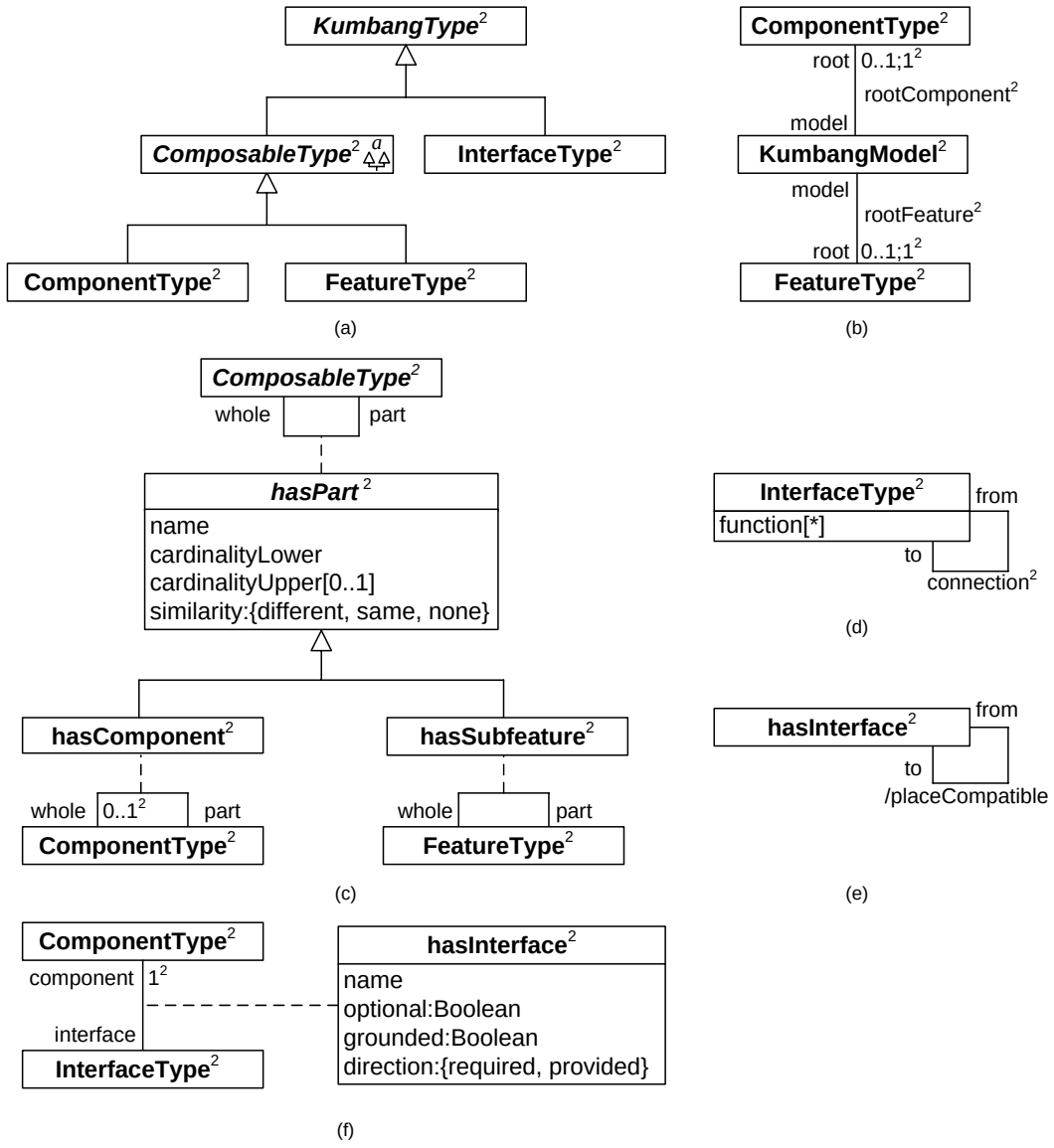

Figure 5.1: The top level of KUMBANG$_{\text{NIVEL}}$, i.e., KUMBANG represented using NIVEL. The syntactic constructs adopted from UML class diagrams are used in roughly the same meaning in NIVEL as in UML; novel constructs are explained in the text. (a) Taxonomy of KUMBANG types. (b) Model and roots. (c) Compositional structure. (d) The notion of subtyping for interfaces (first-order instances) and connections between interfaces (second-order instances). (e) Place compatibility. (f) Interfaces of components.

The *hasPart* association is specialised for component and feature types into concrete associations *hasComponent* and *hasSubfeature*, respectively. For both specialised associations, both roles are redefined in a way such that a component may only have other components as its part and a feature may only have other features as its subfeature. Further, an additional cardinality constraint applies to *hasComponent*: a component may only be a part of a single whole.

The idea of root is illustrated in Figure 5.1 (b). The associations *rootComponent* and *rootFeature* are both of potency 2. The first-order instances of the associations represent the idea of a root type in a variability model and the second-order instances that of a root instance in a configuration.

Cardinality constraints apply to both associations. As discussed in Section 3.5, at most one root component type and root feature type may be defined for a KUMBANG model. Further, if a root type, either component or feature, is defined, a configuration must contain exactly one instance of the type.

Constraints A number of additional constraints supplementing the metamodel presented in Figure 5.1 are needed to complete the semantics of $\text{KUMBANG}_{\text{NIVEL}}$. They are given under separate headings such as this.

When discussing the constraints, reference is made to the corresponding rules in Chapter 3 to highlight the interrelation between the two approaches to formalising KUMBANG. In case two or more similar constraints are needed, only one of them may be written out and the rest be omitted to improve readability. Just as in Chapter 3, the convention that non-domain names of predicates are printed in ***bold*** is adopted. See Table 4.1 for reference on the NIVEL predicates appearing in the constraints.

The restrictions implied by the *similarity* attribute of the *hasPart* association are encoded as follows for the value *same*:

$$\begin{gathered}
\leftarrow \boldsymbol{hasPart^2}(W, P_1, N), hasPart_p^2(W, P_1, N), \\
\boldsymbol{hasPart^2}(W, P_2, N), hasPart_p^2(W, P_2, N), P_1 \neq P_2, \\
instanceOf_D(D, hasPart), \\
\boldsymbol{hasValue}(D, similarity, same), hasValue_p(D, similarity, same), \\
\boldsymbol{hasValue}(D, name, N), hasValue_p(D, name, N), \\
\boldsymbol{playsRoleIn}(W, whole, D), playsRoleIn_p(W, whole, D), \\
\boldsymbol{instanceOf_d}(P_1, T_1), instanceOf_p(P_1, T_1), \\
\boldsymbol{instanceOf_d}(P_2, T_2), instanceOf_p(P_2, T_2), T_1 \neq T_2, \\
instanceOf_p(T_1, composableType), instanceOf_p(T_2, composableType)
\end{gathered}$$

cf. Rule 3.3. In the above rule, the ternary predicate $\boldsymbol{hasPart^2}(w, p, n)$ is the tuple representation of the second order instances of the *hasPart* association and gives that composable instance w has composable instance p as its part with name n; a possible variant of the predicate is defined with the obvious semantics. The corresponding rule for the value *different* is similar and omitted for brevity.

The unary predicate ***justified***(i) gives that instance i is justified. An instance may be justified by being a root instance; cf. Rule 3.1:

$$\textbf{\textit{justified}}(I) \leftarrow \textbf{\textit{fRoot}}^{\textbf{2}}(I, C), \mathit{fRoot}_p(T, M),$$
$$\mathit{instanceOf}_{pd}(I, T), \mathit{instanceOf}_{pd}(C, M)$$

where the predicate $\mathit{fRoot}_p(t, m)$ is the tuple representation of the *rootFeature* association giving that feature type t is the root feature type in KUMBANG model m. Further, the predicate $\textbf{\textit{fRoot}}^{\textbf{2}}(f, c)$ is the tuple representation of the second-order instances of the *rootFeature* association giving that feature f is the root feature in configuration c.

A component root is justified in a similar manner; the related rules are omitted for brevity.

In addition to being justified by being a root instance, an instance may be justified by being a part of some other instance, cf. Rule 3.5:

$$\textbf{\textit{justified}}(P) \leftarrow \textbf{\textit{hasPart}}^{\textbf{2}}(W, P, N), \mathit{hasPart}^2_p(W, P, N)$$

No instance without justification may appear in a valid configuration:

$$\leftarrow \mathit{instanceOf}_{tp}(I, \mathit{kumbangType}, 2), \textbf{\textit{class}}(I), \mathit{not}\ \textbf{\textit{justified}}(I)$$

The cardinality values of part definitions are given a semantics as follows. The case of too few parts is ruled out by the integrity constraints:

$$\leftarrow \left\{\textbf{\textit{hasPart}}^{\textbf{2}}(W_i, P, N) : \mathit{hasPart}^2_p(W_i, P, N)\right\} L - 1,$$
$$\textbf{\textit{hasValue}}(D, \mathit{cardLower}, L), \mathit{hasValue}_p(D, \mathit{cardLower}, L), L \geq 1,$$
$$\textbf{\textit{hasValue}}(D, \mathit{name}, N), \mathit{hasValue}_p(D, \mathit{name}, N),$$
$$\textbf{\textit{playsRoleIn}}(W_t, \mathit{whole}, D), \mathit{playsRoleIn}_p(W_t, \mathit{whole}, D),$$
$$\textbf{\textit{instanceOf}}(W_i, W_t), \mathit{instanceOf}_p(W_i, W_t)$$

cf. Rule 3.2. The case of too many parts is covered by an analogous rule that is omitted for brevity.

5.4 Attribute

A composable type may be characterised by attributes. This is enabled by setting the attribute *instancesMayDefineAttributes* to *true* for *ComposableType*, as illustrated by the letter *a* in the top-right corner of *ComposableType* in Figure 5.1 (a).

Attributes in KUMBANG are mandatory in the sense that exactly one value must be selected for each attribute in a valid configuration, whereas attributes in NIVEL may be specified a cardinality enabling both optional and set-valued attributes.

Constraints The lower bound of attributes of composable types is required to be equal to 1:

$$\leftarrow \mathit{hasAttr}_D(I, N, P, D, L, U), \mathit{instanceOf}_p(I, \mathit{composableType}), L \neq 1$$

The upper bound must likewise equal 1:

$$\leftarrow \mathit{hasAttr}_D(I, N, P, D, L, U), \mathit{instanceOf}_p(I, \mathit{composableType}), U \neq 1$$

5.5 Interface and connection

An interface type in KUMBANG is represented by an instance of *InterfaceType*, see Figure 5.1 (d). The functions of an interface type are represented using the *function* attribute. Similarly, the Interfaces in configurations are represented by the second-order instances of *InterfaceType*.

An interface definition in a KUMBANG model is represented by a first-order instance of the binary association *hasInterface* of potency 2, see Figure 5.1 (f). The two roles of the association, *component* and *interface* are played by *ComponentType* and *InterfaceType*, respectively. The association has a number of attributes corresponding to the various values of interface definition discussed in Section 3.7. That an interface must be an interface of a single component is captured by a cardinality constraint. The interfaces of components in a configuration are represented by the second-order instances of *hasInterface*.

The notion of subtyping is represented by the first-order instances of the binary association *connection* with roles *from* and *to* both played by *InterfaceType*, see Figure 5.1 (d). Connections between interfaces are represented using the second-order instances of *connection*.

The notion of place compatibility is represented by the *placeCompatible* association, see Figure 5.1 (e). The two roles of the association, *from* and *to*, are both played by *hasInterface*.

Constraints Just as components and features, interfaces must be justified:

$$\boldsymbol{justified}(I) \leftarrow \boldsymbol{hasInterface^2}(C, I, N), hasInterface_p^2(C, I, N)$$

where the ternary predicate $\boldsymbol{hasInterface^2}(c, i, n)$ is the tuple representation of the second order instances of *hasInterface* association. The semantics is that component c has interface i as its interface with name n.

A suitable number of interfaces must appear in a valid configuration. The case of too few interfaces is ruled out as follows:

$$\leftarrow \left\{\boldsymbol{hasInterface^2}(C_i, I, N) : hasInterface_p^2(C_i, I, N)\right\} 0,$$
$$\boldsymbol{hasValue}(S_t, optional, false), hasValue_p(S_t, optional, false),$$
$$\boldsymbol{hasValue}(S_t, name, N), hasValue_p(S_t, name, N),$$
$$\boldsymbol{playsRoleIn}(C_t, component, S_t), playsRoleIn_p(C_t, component, S_t),$$
$$\boldsymbol{instanceOf}(C_i, C_t), instanceOf_p(C_i, C_t)$$

The case of too many interfaces is covered by a similar rule that is omitted for brevity.

The Koala connection constraints discussed in Sections 2.3.1 and formalised in Section 3.7 must be stated as constraints in KUMBANG$_{\text{NIVEL}}$ as well.

As a part of expressing these constraints, the extension of a number of associations must be *derived* using special constraints. The notion of derived model elements is not part of NIVEL as defined in the scope of this dissertation [IV]. However, the derivations discussed below are needed to make KUMBANG$_{\text{NIVEL}}$ complete and also serve as examples of derivations.

The binary predicate $notSubtypeOf(i_1, i_2)$ giving that interface type i_1 is not a subtype of i_2 is defined as follows, cf. Rule 3.7:

$$\begin{gathered} notSubtypeOf(I_{from}, I_{to}) \leftarrow \\ not\ hasValue_D(I_{from}, func, F), hasValue_D(I_{to}, func, F), \\ instanceOf_D(I_{from}, interfaceType), instanceOf_D(I_{to}, interfaceType) \end{gathered}$$

Next, the binary predicate *notSubtypeOf* can be used to derive the first-order instances of the *connection* association, cf. Rule 3.6:

$$\begin{gathered} instanceOf_{Der}(cn(I_{from}, I_{to}), connection) \leftarrow not\ notSubtypeOf(I_{to}, I_{from}), \\ instanceOf_D(I_{from}, interfaceType), instanceOf_D(I_{to}, interfaceType) \end{gathered} \tag{5.1}$$

In the above rule, the binary function symbol *cn* (for "connection") is used to construct a new object constant unique to the pair of interfaces (i_{from}, i_{to}) and the set of first order instances of the *connection*. Further in the rule, $instanceOf_{Der}(i, t)$ is a domain predicate with the semantics that i is a derived instance of t, the subscript *Der* for "derived". A derived instance is a special case of a possible instance: a derived instance may but needs not appear in a valid configuration. The predicate must be introduced for reasons related to the notion of domain restrictedness in WCRL; this notion and different classes of predicates used in the formalisation of NIVEL are discussed in more detail in [IV].

The roleplays in the derived instances must be likewise derived separately for the *from* role:

$$\begin{gathered} playsRoleIn_D(I_{from}, from, cn(I_{from}, I_{to})) \leftarrow \\ instanceOf_{Der}(cn(I_{from}, I_{to}), connection), \\ instanceOf_D(I_{from}, interfaceType), instanceOf_D(I_{to}, interfaceType) \end{gathered} \tag{5.2}$$

The derivation for the *to* role is omitted for brevity.

The derived first-order instances of *connection* must appear in a valid configuration as derived or in other words, it may not be the case that a derived instance does not appear in a configuration:

$$\begin{gathered} \leftarrow not\ \boldsymbol{instanceOf_d}(cn(I_{from}, I_{to}), connection), \\ instanceOf_{Der}(cn(I_{from}, I_{to}), connection), \\ instanceOf_D(I_{from}, interfaceType), instanceOf_D(I_{to}, interfaceType) \end{gathered} \tag{5.3}$$

The notion of place compatibility is represented using the *placeCompatible* association. Instances of the association are derived for the three cases illustrated in Figure 3.5. Case (a) is represented by the rule

$$\begin{gathered} instanceOf_{Der}(pc(D_{from}, D_{to}), placeCompatible) \leftarrow \\ instanceOf_D(D_{from}, hasInterface), instanceOf_D(D_{to}, hasInterface), \\ playsRoleIn_D(C_{from}, component, D_{from}), instanceOf_D(C_{from}, componentType), \\ playsRoleIn_D(C_{to}, component, D_{to}), instanceOf_D(C_{to}, componentType), \\ playsRoleIn_D(C_{from}, part, D_{part}), playsRoleIn_D(C_{to}, whole, D_{part}), \\ instanceOf_D(D_{part}, hasComponent), \\ hasValue_D(D_{from}, direction, required), hasValue_D(D_{to}, direction, required) \end{gathered}$$

cf. Rule 3.8. In the above rule, the binary function symbol *pc* stands for "place compatible" and is used for similar purposes as the symbol *cn* in Rule 5.1. Cases (b) and (c) are covered by similar rules that are omitted for brevity.

As in the case of *connection* above, the roleplays in the derived instances of *placeCompatible* must be derived using separate rules. The rules are similar to Rule 5.2 and hence omitted. Also in parallel with Rule 5.3, the derived instances of the association must be included in a valid configuration; this rule is likewise omitted for brevity.

The second-order possible instances of *connection* representing connections between interfaces in configurations, are derived as follows:

$$\begin{gathered}
instanceOf_{Der}(cn^2(I_{from}, I_{to}), cn(T_{from}, T_{to})) \leftarrow \\
instanceOf_{Der}(cn(T_{from}, T_{to}), connection), \\
instanceOf_{D}(I_{from}, T_{from}), instanceOf_{D}(T_{from}, interfaceType), \\
instanceOf_{D}(I_{to}, T_{to}), instanceOf_{D}(T_{to}, interfaceType), \\
instanceOf_{D}(L_{from}, D_{from}), playsRoleIn_{D}(I_{from}, interface, L_{from}), \\
instanceOf_{D}(L_{to}, D_{to}), playsRoleIn_{D}(I_{to}, interface, L_{to}), \\
instanceOf_{Der}(pc(D_{from}, D_{to}), placeCompatible), \\
instanceOf_{D}(D_{from}, hasInterface), \\
playsRoleIn_{D}(D_{from}, from, pc(D_{from}, D_{to})), \\
instanceOf_{D}(D_{to}, hasInterface), \\
playsRoleIn_{D}(D_{to}, to, pc(D_{from}, D_{to}))
\end{gathered} \tag{5.4}$$

cf. Rule 3.9. The binary binary function symbol, cn^2 is similar to *cn* and *pc* above, but now the context is the second-order instances of the *connection* association. In the derived instances, the source (*from*) interface will be a supertype of the target (*to*) interface and the relative positions of the interfaces will be place compatible, as required by the Koala connection constraints. The derivation of the roleplays for the derived associations is again analogous to Rule 5.2 and hence omitted.

Finally, the constraints on connections between interfaces must be formulated. Towards this end, an auxiliary predicate $cn^2_p(i_{from}, i_{to})$ giving the tuple representation of the possible second-order instances of the *connection* association:

$$\begin{gathered}
cn^2_p(I_{from}, I_{to}) \leftarrow \\
instanceOf_{D}(I_{from}, T_{from}), instanceOf_{D}(T_{from}, interfaceType), \\
instanceOf_{D}(I_{to}, T_{to}), instanceOf_{D}(T_{to}, interfaceType), \\
instanceOf_{Der}(cn^2(I_{from}, I_{to}), cn(T_{from}, T_{to})), \\
instanceOf_{Der}(cn(T_{from}, T_{to}), connection)
\end{gathered}$$

For actual connections, $\boldsymbol{cn^2}(i_{from}, i_{to})$ is defined as:

$$\begin{gathered}
\boldsymbol{cn^2}(I_{from}, I_{to}) \leftarrow cn^2_p(I_{from}, I_{to}), \\
instanceOf_{D}(I_{from}, T_{from}), instanceOf_{D}(I_{to}, T_{to}), \\
\boldsymbol{instanceOf_d}(cn^2(I_{from}, I_{to}), cn(T_{from}, T_{to})), \\
instanceOf_{D}(T_{from}, interfaceType), instanceOf_{D}(T_{to}, interfaceType)
\end{gathered}$$

The above-defined predicates can now be used to express the constraint that each non-grounded interface must be connected to exactly one interface. This is achieved by stating that it may not be the case that such an interface is connected to no interface:

$$\leftarrow \left\{ \boldsymbol{cn^2}(I_{from}, I_{to}) : cn_p^2(I_{from}, I_{to}) \right\} 0,$$
$$\boldsymbol{playsRoleIn}(I_{from}, interface, L_{from}), playsRoleIn_D(I_{from}, interface, L_{from}),$$
$$\boldsymbol{instanceOf_d}(L_{from}, D_{from}), instanceOf_D(L_{from}, D_{from}),$$
$$instanceOf_D(D_{from}, hasInterface), hasValue_D(D_{from}, grounded, false)$$

cf. Rule 3.10, or to two or more interfaces:

$$\leftarrow 2 \left\{ \boldsymbol{cn^2}(I_{from}, I_{to}) : cn_p^2(I_{from}, I_{to}) \right\},$$
$$\boldsymbol{playsRoleIn}(I_{from}, interface, L_{from}), playsRoleIn_D(I_{from}, interface, L_{from}),$$
$$\boldsymbol{instanceOf_d}(L_{from}, D_{from}), instanceOf_D(L_{from}, D_{from}),$$
$$instanceOf_D(D_{from}, hasInterface), hasValue_D(D_{from}, grounded, false)$$

cf. Rule 3.11.

5.6 Instantiation

As in the case described in Section 3.9, the elements that may possibly appear in configurations must be explicitly represented as WCRL rules. An algorithm based on ideas similar to the one given as Figure 3.6 can be used in conjunction with NIVEL as well. However, the algorithm must be adjusted to accommodate the differences in the knowledge representation schemes adopted here and in Chapter 3. Such an algorithm is given as Figure 5.2.

Input: composable type w_t to be instantiated
Output: set of rules representing w_i, an instance of w_t
let w_i be a new object constant
add fact $instanceOf_D(w_i, w_t) \leftarrow$
foreach d_t such that d_t:$hasPart \wedge w_t \in d_t \rightarrow whole$ **do**
    let d_i be a new object constant
    add fact $instanceOf_D(d_i, d_t) \leftarrow$
    add fact $playsRoleIn_D(w_i, whole, d_i) \leftarrow$
    if $d_t.similarity = different$ **then**
        $maxCard := 1$
    else
        $maxCard := d_t.cardinalityUpper$
    foreach t such that t:$composableType \wedge t \in d_t \rightarrow part$ **do**
        foreach $p_t \in t \cup t.subtypes$ **do**
            if $p_t.isAbstract$ **then continue**
            for $j := 1$ **to** $maxCard$ **do**
                $p_i := instantiate(p_t)$
                add fact $playsRoleIn_D(p_i, part, d_i) \leftarrow$
foreach d_t such that d_t:$hasInterface \wedge w_t \in d_t \rightarrow component$ **do**
    let d_i be a new object constant
    add fact $instanceOf_D(d_i, d_t) \leftarrow$
    add fact $playsRoleIn_D(w_i, component, d_i) \leftarrow$
    foreach i_t such that i_t:$interfaceType \wedge i_t \in d_t \rightarrow interface$ **do**
        let i_i be a new object constant
        add fact $playsRoleIn_D(i_i, interface, d_i) \leftarrow$
return w_i

Figure 5.2: Algorithm $instantiate(t)$ for instantiating a composable type in KUMBANG$_{\text{NIVEL}}$ model, initially called for the component and feature roots as the input parameter. For notational simplicity, same symbols are used for model elements and the object constants representing them. In the algorithm, $a \rightarrow r$ denotes the set of classes playing role r in association a, $i:t$ a Boolean that is true if (and only if) i is an instance of t, and $c.a$ the value named a of class c.

6 Discussion and comparison with previous work

In this chapter, it is shown how the constructions described in three preceding chapters address the research problems and questions posed in Section 1.2 and compares to previous work. The chapter is structured according to the research problem: Section 6.1 addresses the first research problem, related to software variability modelling languages whereas Section 6.2 iterates on the second problem concerning metamodelling languages. Both sections are further divided into subsections based on the detailed research questions.

6.1 Software variability modelling languages

This section elaborates on the three software variability modelling languages developed in this dissertation, namely Koalish [I], Forfamel [II] and Kumbang [III] and on how they address the first research problem. The research questions related to the research problem will be answered in their respective subsections.

6.1.1 Conceptual basis

The conceptual bases of the software variability modelling languages draw from a number of sources. The notions of software architecture and feature underlying Koalish and Forfamel, respectively, are adopted from the software engineering domain, within which research on software variability is centred around the notion of software product family. A product family architecture [23] and a managed set of features [34] are defining characteristics of a software product family, see Section 2.1.

While the main concepts of the three software variability modelling languages stem from the software engineering domain, many aspects of the language design are drawn from the product configuration domain, especially the configuration ontology by Soininen et al. [110].

The three-level organisation of the software variability modelling languages (see Section 3.1) resembles the three-level basic structure of the configuration ontology by Soininen et al. [110]: both are based on three levels and the roles of the levels are roughly equivalent. A similar approach has been suggested in the context of feature modelling [32], with the three levels termed *family-metamodelling*, *family modelling* and *application modelling* roughly corresponding to the three levels in the above-discussed approaches.

As discussed in Section 2.3, a large number of ADLs have been suggested [80]. Most of these are intended for single-system modelling and hence are not well, if at all, suited for modelling variability. The most important exceptions are Koala [130, 128, 129] and xADL 2.0 [43, 123].

Even though intended for describing single-system architecture, the modelling concepts of any ADL could in principle have been extended with variability mechanisms. Koala was chosen as the basis for extension due its practical relevance: unlike most, if not all, other ADLs, Koala is in industrial use. This choice has later gained additional support: since version 2.0 of UML, its component diagram notation is

based on concepts similar to those underlying Koala [120]. Koala has also influenced the Com^2 component model [96].

KOALISH extends Koala through a number of variability mechanisms. Most importantly, the definition of the compositional structure of components in KOALISH may include a number of possible types for the part and the number of components occurring as a part may be specified using a cardinality; Koala only allows a single possible type of which exactly one instance must occur as part. Similarly for interfaces, multiple alternative types may be specified in KOALISH whereas the type of an interface is fixed in Koala. The variability mechanisms of Koala defined in implementation-related terms are given a more abstract treatment in KOALISH.

FORFAMEL synthesises previous feature-modelling languages in the sense that it covers the conceptual core shared by most, if not all, previous feature-modelling languages and includes a number of extensions, such as shared subfeatures, feature and group cardinalities and attributes.

A number of factors differentiate previous feature-modelling languages from each other and from FORFAMEL. A key factor is how the concept of configuration is defined. In most feature-modelling languages, a configuration is defined as a set of features, or more specifically, as a subset of the features appearing in the feature model. In FORFAMEL, on the other hand, a configuration consists of a set of features that are instances of feature types organised in a feature model and further includes the subfeature relationship and the attributes of features.

More complex conceptions of configuration have likewise been adopted in previous feature-modelling languages: a configuration may contain instances of features in feature models [32]; alternatively, a configuration can be represented by strings encoding the subfeature relation [39].

While a configuration defined as a subset is a sufficiently expressive structure in basic forms of feature modelling, e.g., in FODA [63], a more complex notion of configuration is required to conceptualise a number of extensions. First, if two or more features may share a subfeature, as is the case in, e.g., FORM [64], a configuration defined as a set of features fails to capture the intended subfeature relation. That is, it may not be clear of which features a given feature is a subfeature and the problem of deciding whether a configuration is valid with respect to a model becomes **NP**-hard [103].

Second, the notion of feature cardinality [36, 32, 39] allows (in a sense) the same feature to occur two or more times as a subfeature of a parent. Again, a configuration defined as a subset fails to capture the intended subfeature relation. In FORFAMEL, the problem is avoided by considering each feature in a configuration as a distinct instance of a feature type in the model; a similar approach is followed by Cechticky et al. [32]. Alternatively, in the grammar-based approach by Czarnecki et al. [39], strings representing configurations are ordered collections that allow repetitions of tokens representing features.

Third, attributes of features likewise call for more complex conceptions of configuration. In FORFAMEL, the attributes of features are represented by a ternary relation between the feature, attribute name and value. In previous work on feature modelling and attributes [36, 40, 39], attributes in feature models have been

defined similarly as in FORFAMEL; the question of how attributes are reflected in configurations is not given much attention.

The concept of subfeature definition is similar to the concept of part definition in the configuration ontology by Soininen et al. [110]. In essence, a subfeature definition is a reified relationship between a whole type and one or more possible part types with its own set of properties, such as name and cardinality. In previous work on feature modelling, such properties have mostly been assigned to the subfeature itself; however, a similar approach is followed in the work by Czarnecki et al. [39, 40], although in the latter, the reification is "implicit in the metamodel".

The notion of subfeature definition in FORFAMEL unifies the notions of solitary subfeature and feature group. Hence, a subfeature definition includes as special cases notions such as optional subfeature and or-features. The *similarity* attribute is needed to distinguish between cases corresponding to selection from a group of features (value *different*) and repeating the same subfeature a number of times (*same*). For a more detailed discussion on the differences between the notions of subfeature definition in FORFAMEL and subfeature in previous feature-modelling methods, see [II].

KUMBANG is motivated by the observation that in some software product families, it may not be sufficient to model the variability from either the architecture or feature point of view alone; instead, both may be required. Hence, KUMBANG unifies KOALISH and FORFAMEL in that the language can be used to model the variability from both an architectural and a feature point of view in a uniform manner including implementation constraints relating the two points of view; this is also the most important contribution of KUMBANG from the conceptual point of view.

An important contribution of KUMBANG is that the architecture and feature points of view are independent of each other: both points of view are subject to well-formedness and consistency rules of their own stemming from KOALISH and FORFAMEL, respectively. As a result, the introduction of implementation constraints may not lead to either of the views becoming inconsistent in a valid configuration. Further, unlike in some previous work, e.g., [35], no extra analysis is required to ensure the consistency of either of the points of view.

6.1.2 Language definition

The abstract syntax of KOALISH is defined using a combination of natural language and a concrete syntax definition in EBNF, see Figure 3.2 and [I]. Following descriptions in natural language does not require knowledge of any formal language. In addition, natural language can be used to provide intuitive explanations of the language constructs. On the other hand, natural language is almost inevitably ambiguous and descriptions given in natural language tend to be lengthy and, unlike graphical notation at its best, cannot be understood at a glance. Therefore, natural language descriptions best supplement formal definitions.

A concrete syntax given in EBNF or some other notation has the benefit of being precise and unambiguous. On the other hand, such a syntax includes details that are irrelevant from the conceptual point of view and clutter up the definition. Also,

the means for expressing well-formedness constraints are limited in EBNF compared with, e.g., cardinality constraints in NIVEL or multiplicities in UML class diagrams.

The abstract syntax of FORFAMEL is defined using a UML class diagram, see Figure 3.3. This approach is motivated by the fact that UML class diagrams are widely used as a conceptual modelling language for various domains [72] and can therefore be assumed to be well understood, at least in the software engineering community. However, the approach introduces a number of problems.

First, UML class diagrams are based on two levels occupied by classes and instances, whereas the variability modelling languages are best represented using three levels, as shown in Section 3.1, Figure 3.1 and discussed in Section 6.1.1. Therefore, as previously argued by Atkinson and Kühne [11], there is a mismatch between the number of levels, so both types and instances have to be represented as classes. Further, a number of modelling constructs that are included in UML had to be replicated for the FORFAMEL metamodel, including the notion of abstractness of a feature type and attributes of and the subclassing relations between feature types. Also, UML does not provide a formal or otherwise rigorous semantics for what is required of a feature to be an instance of a feature type or, more generally, for a model element an instance of another model element.

In previous work on feature modelling, natural language [63, 64, 54, 37, 126, 75, 19, 17] and UML class diagrams [36, 32, 133, 38, 40, 39, 30] have been the predominant approaches to specifying the abstract syntax for feature models. In most languages, an abstract syntax is defined for feature models only, excluding configurations; this may in part be due to the fact that a configuration is defined as a subset of the features appearing in the feature models.

The abstract syntax of KUMBANG is defined using a UML profile based on the approach described in Section 2.7.1. Applying the profile mechanism enables modelling on three levels and thus overcomes some of the above-discussed difficulties related to using standard UML class diagrams.

However, as discussed in Section 2.7.1, the profile mechanism introduces a set of problems of its own. First, the stereotypes included in profiles are technically extensions to UML language and not intended for representing domain (meta)types [11]. Also, the relation between stereotypes and classes is different from the one between classes and their instances, either objects on M0 or *InstanceSpecification*s on M1. From a cognitive point of view, a profile, such as the one given in Figure 3.4 is hardly self-explaining but extensive knowledge of UML and its implicit semantics is required to comprehend such a description. Moreover, the profile mechanism has itself been criticised for ambiguity, e.g., with respect to whether only classes or all metaclasses may be extended, and problems in its semantics [58]. Also, from the pragmatic point of view, in the context of UML 1.4 [122], there has been confusion about whether stereotypes apply to classes, objects or both [12].

An alternative definition of KUMBANG using NIVEL, $\text{KUMBANG}_{\text{NIVEL}}$, is given in Chapter 5, Figure 5.1; see also [IV] for a representation of FORFAMEL using NIVEL. As NIVEL supports models spanning an arbitrary number of levels, the three levels of abstraction in KUMBANG can be represented using NIVEL in a straightforward manner. Just as seen in relation to FORFAMEL in [IV], the number of model elements

needed to represent KUMBANG is significantly smaller using NIVEL than a UML profile, cf. Figures 5.1 and 3.4: there is no need to represent classes at the instance level (*KumbangConfiguration*, *KumbangInstance* and its subclasses, and *Connection*) in the NIVEL variant. Further, unlike in the case of representing FORFAMEL using a UML class diagram, the definition of concepts such as attribute, abstractness and subclassing needs not be replicated. The association concept in NIVEL can be used to represent part definitions in a straightforward manner, see Figure 5.1 (c).

6.1.3 Formal semantics

The software variability modelling languages are given formal semantics by translation to Weight Constraint Rule Language (WCRL) [105], see Chapter 3. WCRL is well suited for the purpose for a number of reasons. First, WCRL has a declarative formal semantics. The language is decidable and of reasonable computational complexity while still including a form of variables, predicates and function symbols that facilitate knowledge representation. Although general-purpose, WCRL has been shown to be well suited for representing variability of non-software products [108]: the semantics of WCRL include a notion of groundedness that has been argued to be useful in both the product configuration [107] and feature-modelling domain, see Section 2.2; cardinalities can be represented using cardinality constraints in a succinct manner. The *smodels* inference system operating on WCRL has been shown to be competitive in performance compared with other solvers, especially in the case of problems including structure [105]. The *smodels* system is available under the GNU Public Licence.

An alternative formal semantics for KUMBANG is defined in Chapter 5 as a part of $\text{KUMBANG}_{\text{NIVEL}}$. The bulk of the formal semantics is based on the formal semantics of NIVEL. A number of further semantic issues is captured using standard cardinality constraints as defined, e.g., in ER modelling; however, cardinality constraints in NIVEL have not yet been defined a systematic translation to WCRL or given a formal semantics by other means. The residual part of the semantics is represented using WCRL integrity constraints.

The number of rules required to express the residual constraints is moderate (ca. 20), and many of the rules are simple and resemble each other, e.g., the rules for deriving the roleplays in derived associations, see Rule 5.2 for an example. Some of the rules, e.g., Rule 5.4, are lengthy, which is in part due to issues related to WCRL and in part to the inherent complexity of the underlying domain, in this case Koala connection constraints.

The semantics given in terms of NIVEL includes well-formedness checks on the variability models in a declarative form, which is an advantage over the translation defined in Chapter 3 where such checks have to be performed using other means, e.g., procedurally in compiler-like tools [III]. Also, the correctness of the translation depends on the correctness of the instantiation algorithm: there is no rule that prevents an interface instance from being both a provided and a required interface, an abstract type having instances etc.

Previous work on modelling product family architectures has not emphasised

formal semantics. Specifically, in the case on xADL 2.0 [44], semantics are considered the responsibility of the ADL developer. No formal or declarative semantics is given for Koala.

Feature-modelling languages have been given formal semantics using a range of knowledge representation languages, such as propositional logic [75, 133, 17], binary decision diagrams [124, 42], constraint programming [19, 20], and grammars [39, 17]. A generic semantics covering a number of feature-modelling languages have been defined in terms of set theory [102, 103].

As can be recalled from Section 2.2, the groundedness property of feature modelling requires that a subfeature (s) may be included in a valid configuration only if its parent (p) is included. In propositional logic, the requirement is captured by the formula:

$$s \rightarrow p \tag{6.1}$$

intuitively stating that if s is included in the configuration, so must p. Formulae of the above kind must be included in the formalisation in addition to formulae capturing the subfeature relations themselves. As an example, the formula

$$p \rightarrow s_1 \vee s_2 \tag{6.2}$$

could be used to represent that p has an or-feature consisting of s_1 and s_2 as its subfeature. The semantics of the WCRL, on the other hand, includes a groundedness property, which enables capturing the groundedness property for feature modelling in a concise manner. As an example, the above-discussed or-feature can be represented by a rule of the form Rule 3.2:

$$1\,\{s_1, s_2\} \leftarrow p \tag{6.3}$$

Formula 6.1 correctly captures the notion of groundedness when configurations are restricted to trees. However, if shared subfeatures are allowed, the formulae of the form 6.1 lead to the counterintuitive semantics that if feature s is included in a configuration, all its parent must be included as well. This problem can be circumvented by replacing Formula 6.1 by

$$s \rightarrow p_1 \vee \ldots \vee p_n \tag{6.4}$$

where $\{p_1, \ldots p_n\}$ is the set of parent features of s. While formulae of this kind capture the intuition, they combine knowledge from different model elements and thus make the translation non-modular, i.e., model elements, in this case subfeatures, cannot be translated independently of each other.

From an application point of view, it is important that the formal language employed enables automated reasoning. Efficient algorithms and solvers have been devised for propositional logic, constraint satisfaction problem, binary decision diagrams and WCRL. Consequently, formalisations based on these languages are supported by such solvers. On the other hand, a formalisation that does not commit to a specific well-defined computational problem or knowledge representation language, e.g., the one given by Schobbens et al. [102, 103] based on set theory, does not enjoy

such support. Then again, such a formalisation is free from restrictions due to any solver and may thus better enable investigating and proving formal properties [116]. Finally, proprietary modelling languages and solvers, e.g., OPL (optimization programming language) [127] and CPLEX[6], respectively, used by Benavides et al. [19], may provide a wide range of modelling facilities and highly efficient reasoning but may not lend themselves for formal investigations.

Binary decision diagrams [1] are a means for representing Boolean functions. A propositional formula, e.g., one representing a feature model, can be interpreted as a Boolean function of all its propositions. Hence, binary decision diagrams can be used to represent and reason about feature models.

In practical settings, reduced ordered binary decision diagrams are used. Given a formula, constructing such a diagram is computationally expensive, requiring in the worst case a time exponential in the length of the formula. However, once constructed, many computational tasks of interest can be solved in at most polynomial time with respect to the size of the diagram. [42] Hence, binary decision diagrams and algorithms operating on these are an alternative for satisfiability solvers as tools for reasoning about feature models represented using propositional logic.

Propositional logic and constraint programming are not as such well suited for formally representing the more complex notions of configuration including type-instance and part-whole relationships and attributes discussed in Section 6.1.1. As shown in Chapter 3 for WCRL, predicate and variables ranging over object constants enable the concise representation of the above-mentioned notions. Still, the form of WCRL employed in this dissertation is decidable: the WCRL is not a true first-order language; instead, a WCRL program with variables is a shorthand for the all the ground instantiations of its rules [IV], [105]. A similar scheme could be devised for propositional logics, as has been done for, e.g., the planning problem [66]. However, with the exception of FORFAMEL, no such approach has been presented for feature modelling.

A feature model expressed in a language that allows unbound feature cardinalities (denoted by "*") for solitary features may have valid configurations that are infinite. Neither propositional logic nor WCRL enable reasoning about infinite models and hence configurations. Instead, first-order and higher-order logics could be used. Likewise, the stable model semantics underlying WCRL has been extended towards infinite models [22]. However, the ability of reasoning about infinite models comes at the cost of losing decidability. Alternatively, decidable subsets of such languages can be used, such as description logics [13] and finitary programs [22]. However, such subset imply syntactic restrictions that many, especially practioners, may find hard to accept.

6.2 Metamodelling languages

This section elaborates on the second research problem, related to metamodelling languages. In particular, the two subsections show how the related research questions are addressed.

[6]See `http://www.ilog.com/products/cplex/`

6.2.1 Conceptual basis

In this section, certain concepts of NIVEL are discussed and reflected against previous work; see [IV] for a more detailed discussion of the conceptual basis of NIVEL.

In short, the conceptual basis of NIVEL consists of concepts (class, instantiation, association, generalisation, attribute and value) found in existing modelling languages, most importantly UML, ER modelling and Telos. These concepts have been generalised to allow modelling on any number of levels in a uniform manner using the notions of unified modelling elements [7] and deep instantiation [4, 11].

On the other hand, NIVEL does not include a number of modelling concepts essential in some modelling languages: examples include class-valued attributes, also termed *properties* [121, 2], and powertypes [93, 121, 52]. If included in NIVEL, these concepts would be redundant, at least to an extent: class-valued attributes can in most cases be replaced by binary associations and powertypes by instantiation.

Unlike in UML and ER modelling, a role in NIVEL can be played by more than one class. Hence, the concept of role resembles the concept by the same name in the LODWICK modelling language [112], which is in turn based on the role concept defined by Bachman [14] and revisited by Steimann [113]. In LODWICK, each role is still played by a single *role type* filled by a number of *natural types*. However, the taxonomy of role types is separate from that of natural types.

The concept of part definition in the software variability modelling languages serves as an example of the utility of the role concept of NIVEL and LODWICK. A part definition cannot be represented in a straightforward manner using typical conceptual modelling concepts for relationships, such as an UML association or a relationship as defined in ER modelling, the problematic issue being multiple possible part types that do not necessarily share a supertype. In principle, the problem could be overcome by introducing a supertype for the purpose where a suitable one does not pre-exist. However, the resulting supertypes are in a sense artificial and their number may become large. Hence, these types may pollute the taxonomy by drawing attention away from the "true" taxonomic relations. This approach has also been discouraged when modelling the variability of non-software products [118].

On the other hand, part definitions can be represented using NIVEL associations in a straightforward manner; see Figure 5.1. LODWICK is also relatively well suited for the purpose: although a role type filled by the possible part types still must be introduced, the role types do not interfere with the taxonomy of natural types.

Of previous work on metamodelling languages, MOF [81] promises to handle any number of levels. However, as mentioned in Section 2.7.2, the MOF specification [81] is ambiguous about the abstract syntax and semantics needed to support the claimed ability to define such layers. Further, unlike NIVEL, MOF is committed to the object-orientated paradigm through the use of terms such as "object", "navigation", "serialization", "operation", "reference", "interchange" etc. and makes references to specific technologies, such as XML and the Java programming language. Finally, there seems to be no attempt to give MOF a formal semantics.

Telos [87] resembles NIVEL in that an individual, roughly corresponding to a class in NIVEL, may be an instance of one or more individuals. Also, individuals may have

attributes and there may be generalisations between them in both Telos and Nivel. However, there are important differences between the languages. Telos is based on extensive use of attributes, whereas Nivel includes the association concept as a language primitive. Unlike Nivel, Telos includes no notion of strictness. Finally, while Nivel strives to distinguish between model elements, language elements and the formal entities representing these, Telos seems to intentionally mix all three.

6.2.2 Formal semantics

This section elaborates on formalising metamodelling languages, particularly from the point of view of Nivel.

Nivel has been given a formal semantics by translation to WCRL [IV]. The benefits of WCRL as a knowledge representation language in general have been discussed in Section 6.1.3 and in particular for Nivel in [IV].

As a point of comparison with formalising generic knowledge representation languages, Berardi et al. [21] have given UML class diagrams with a limited form of constraints and ignoring implementation issues a "natural" formal semantics by translation to first-order predicate logic and subsequently to **EXPTIME**-decidable description logics. The encoding in first-order predicate logic serves as a point of reference in the sense that it is possible to argue for the correctness of other formalisations by showing that it is equivalent to the first-order encoding.

In the case of Nivel, following a similar approach would bring a number of benefits. Perhaps most importantly, unlike WCRL, first-order predicate logic is a well-known knowledge representation language. Consequently, a first-order encoding would be readily understandable by a wider audience than one given in WCRL. Further, should Nivel be given an alternative semantics in terms of a knowledge representation language other than WCRL, a first-order encoding would likely provide an easier point of comparison than one in WCRL.

On the other hand, unlike in the case of first-order predicate logic and description logics, it is questionable whether an encoding in first-order predicate logic is more natural than one in WCRL: specifically, the lack of a groundedness property in first-order predicate logic implies that some form of frame axioms are required to prevent elements without justification appearing in a valid model. The absence of cardinality constraints in first-order predicate logic would likewise complicate the encoding.

First-order predicate logic could be used as a basis for inferences. Unlike when using WCRL as described in Chapters 3 and 5 and shown in Figures 3.6 and 5.2, respectively, unrestricted first-order predicate logic does not require an a priori instantiation of model elements, thus resulting in a more simple translation. Of course, this would come at the cost of losing decidability. A third approach is to resort to decidable subsets of first-order predicate logic, e.g., the description logics $\mathcal{DLR}_{ifd}$ and $\mathcal{ALCQI}$ used by Berardi et al. [21]. However, as discussed in Section 6.1.3, the decidability of such languages comes at the cost of syntactic restrictions.

7 Further work

The work presented in this dissertation can be extended in various ways. The software variability modelling languages developed in this dissertation include either one (KOALISH, FORFAMEL) or two (KUMBANG) points of view on the variability of a software product family. In general, more than two such points of view may be useful. As an example, in a car periphery system case study [III], [60], the intuitiveness and thus utility of the model could have been improved by using four such points of view: hardware architecture, software architecture, features provided to the customer and features of the operating environment. Likewise, it has been suggested that modelling variability in services, such as insurance and telecommunications services, requires four points of view [57].

Although such points of view may be used to represent different real-world phenomena, different points of view may be based on a small set of ideas, such as compositional structure and connections. As an example, consider the two specialisations of the part definition concept in Figure 5.1 (c). Consequently, given the metamodelling capabilities of NIVEL, it would seem reasonable to specify such points of view as instances of a generic model of a point of view. The generic model would hence be located one level above the current top level of, e.g., KUMBANG, see Figure 5.1. This approach is made appealing by the fact that developing variability modelling languages and supporting tools from scratch is a task requiring significant expertise and effort.

The semantics given in this dissertation for NIVEL and software variability modelling languages capture the notion of a valid model and configuration, respectively, but fail to capture the notion of an *incomplete model*, i.e., a model that must be supplemented with further model elements to become a valid model. In the context of software variability, this situation arises during interactive configuration [85] or more generally, in staged configuration [40]. The topic has already been identified for further work in 2000 [107]. Also, it has been argued that the line between types and instances in such cases is unclear [97]. Extending the semantics to cover the notion of incomplete models is therefore an important topic for further work.

Although NIVEL covers the most important conceptual modelling concepts, there are a number of concepts that could be integrated into the language. Class-valued attributes, or *properties*, are extensively used in object-oriented programming and design and would likely be a relevant alternative for binary associations in many modelling scenarios. Including a notion of time in NIVEL would enable representing knowledge and reasoning about temporal properties, including notions such as evolution [76] and dynamic vs. static typing.

In its current form, NIVEL includes a number of restrictions that may be unnecessarily strong. The strict metamodelling rule [6] could be weakened to enable more flexible modelling styles. Possible approaches include modelling spaces [5] and allowing associations between classes on different levels. Also, the requirement that all roles of an association must be specified on the top level may be unnecessarily strong and could be weakened.

A constraint language must be defined for Nivel in order to capture the potentially complex dependencies between model elements, especially if used in a model-driven engineering context. As a first step, a semantics could be specified for different forms of cardinality constraints as found [117]. Further, deriving model elements using constraints is extensively used in the UML specification [121] and has also been studied outside the standard [94]. Hence, generalising the cases of derived model elements presented in Section 5.5 is a topic for further research.

There are many conceivable forms of tool support for Nivel. A graphical modelling tool for creating Nivel models could be more usable than a textual language. Such a tool should support both using the model elements on a given level as a domain-specific language for the level next below it and simply creating models that may contain instantiation relationships between model elements, e.g., similar to Figure 6 in [IV]. Likewise, a programming interface for Nivel could be developed; such interfaces could also be generated for specific Nivel models. A database-like implementation similar to ConceptBase [67] is also conceivable. Finally, programming languages based on the concepts of Nivel could be devised in a manner similar to DeepJava [70].

Any modelling language, either abstract or concrete, and any modelling tool supporting such a language can always be used to create models that are not useful or could be significantly improved on. Therefore, developing artefacts of the abovementioned kind is not enough to make a practical impact on software engineering practices. In addition and more importantly, software engineering students must be taught the underlying paradigm and trained to apply it in practical contexts.

8 Conclusions

Variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context. Increasing amounts of variability are required of software systems, either embedded or stand-alone. The number of possible variants of a software system may be very large. Therefore, efficient methods for managing, modelling and reasoning about software variability are needed. Numerous such methods have been developed. However, most such methods either lack a solid conceptual foundation or are not given a rigorous formal semantics, or both; instead, many authors seem to be more concerned with details of the concrete syntax than the underlying concepts and more interested in providing their methods with tool support than a declarative semantics. Consequently, reasoning about the properties of these languages and comparing them with each other is unnecessarily difficult.

In this dissertation, three novel software variability modelling languages, namely Koalish, Forfamel and Kumbang synthesising the two previous ones, are developed. The languages are based on concepts found relevant to modelling software variability in scientific literature and practice, namely feature and product family architecture. They synthesise and clarify the concepts underlying a number of previous languages. Ideas first developed in product configuration research for modelling variability in non-software products are elaborated on and integrated into the languages. A formal semantics is given for the languages by translation to WCRL. The *smodels* inference system operating on WCRL enables automated, decidable and efficient reasoning about the languages.

One of the goals set for this dissertation was to enable modelling software variability knowledge at different levels of abstraction in a uniform and systematic manner, preferably using an existing conceptual modelling language with a formal semantics. During the course of the work it turned out that no language with the desired modelling capabilities existed. Consequently, a novel conceptual modelling language, Nivel, with the necessary capabilities is developed. Nivel is generic in the sense that its modelling concepts are not specific to software variability or some other similar domain. Instead, Nivel is based on a number of core concepts of previous conceptual modelling languages and incorporates a number of recent ideas including strict metamodelling, distinction between ontological and linguistic instantiation, unified modelling elements and deep instantiation. Hence, Nivel contributes to the theory of conceptual modelling, particularly that of metamodelling. A formal semantics enabling automated, decidable reasoning is given for Nivel by translation to WCRL.

The suitability of Nivel for defining software variability modelling languages is demonstrated through $\text{Kumbang}_{\text{Nivel}}$. It is argued that $\text{Kumbang}_{\text{Nivel}}$ is more compact and easily understandable definition of Kumbang than the original specification given as a UML profile, or the specifications of Koalish and Forfamel using an EBNF grammar and a UML class diagram, respectively. Major parts of the semantics of Kumbang are captured by the semantics of Nivel. Defining

KUMBANG in terms of a generic modelling language also brings software variability modelling closer to other forms of modelling, thus making software variability modelling less of an isolated discipline.

In conclusion, this dissertation contributes to the theory of software variability modelling by introducing software variability modelling languages that integrate concepts previously considered important in the software variability domain with a number of ideas established in the context of product configuration. The resulting languages add conceptual clarity and semantic rigour to previous languages; these qualities are further improved when defined using NIVEL, a metamodelling language also developed in this dissertation. NIVEL itself significantly contributes to the theory of conceptual modelling in general and metamodelling in particular. Further work is required to make the languages developed in this dissertation a practical alternative to the currently predominating modelling languages and tools.

References

[1] Sheldon B. Akers. 1978. Binary Decision Diagrams. IEEE Transactions on Computers c-27, no. 6.

[2] Marcus Alanen and Ivan Porres. 2008. A Metamodeling Language Supporting Subset and Union Properties. Software and Systems Modeling 7, no. 1.

[3] Robert Allen and David Garlan. 1997. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology 6, no. 3, pages 213–249.

[4] Colin Atkinson and Thomas Kühne. 2001. The Essence of Multilevel Metamodeling. In: Martin Gogolla and Cris Kobryn (editors), Fourth International Conference on the Unified Modeling Language (UML 2001), volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Springer.

[5] Colin Atkinson and Thomas Kühne. 2001. Processes and Products in a Multi-Level Metamodeling Architecture. International Journal of Software Engineering and Knowledge Engineering 11, no. 6, pages 761–783.

[6] Colin Atkinson and Thomas Kühne. 2002. Profiles in a Strict Metamodeling Framework. Science of Computer Programming 44, no. 1, pages 5–22.

[7] Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML Infrastructure. ACM Transactions on Modeling and Computer Simulation 22, no. 4, pages 290–321.

[8] Colin Atkinson and Thomas Kühne. 2002. The Role of Metamodeling in MDA. In: International Workshop in Software Model Engineering (in conjunction with UML'02).

[9] Colin Atkinson and Thomas Kühne. 2003. Model-Driven Development: A Metamodeling Foundation. IEEE Software 20, no. 5, pages 36–41.

[10] Colin Atkinson and Thomas Kühne. 2005. Concepts for Comparing Modeling Tool Architectures. In: Lionel Briand and Clay Williams (editors), 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), pages 398–413.

[11] Colin Atkinson and Thomas Kühne. 2007. Reducing Accidental Complexity in Domain Models. Software and Systems Modeling, in press. DOI: 10.1007/s10270-007-0061-0.

[12] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. 2002. Stereotypical Encounters of the Third Kind. In: Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook (editors), 5th International Conference on The Unified Modeling Language (UML 2002), volume 2460 of *Lecture Notes in Computer Science*, pages 100–114. Springer.

[13] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (editors). 2003. The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press.

[14] C. W. Bachman and M. Daya. 1977. The Role Concept in Data Models. In: Third International Conference on Very Large Data Bases (VLDB), pages 464–476. IEEE Computer Society.

[15] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc, and Jean-Michel Bruel. 2003. Formalization of the Whole-Part Relationship in the Unified Modeling Language. IEEE Transactions on Software Engineering 29, no. 5, pages 459–470.

[16] Len Bass, Paul C. Clements, and Rick Kazman. 1999. Software Architecture in Practice. Addison-Wesley, Boston (MA).

[17] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In: Obbink and Pohl [91], pages 7–20.

[18] Don Batory, David Benavides, and Antonio Ruiz-Cortes. 2006. Automated Analysis of Feature Models: Challenges Ahead. Commununications of the ACM 49, no. 12, pages 45–47.

[19] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In: Oscar Pastor and João Falcão e Cunha (editors), 17th Conference on Advanced Information Systems Engineering (CAiSE 2005), volume 3520 of *Lecture Notes in Computer Science*. Springer.

[20] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In: William C. Chu, Natalia Juristo Juzgado, and W. Eric Wong (editors), 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), pages 677–682.

[21] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. 2005. Reasoning on UML Class Diagrams. Artificial Intelligence 168, no. 1-2, pages 70–118.

[22] Piero A. Bonatti. 2004. Reasoning with Infinite Stable Models. Artificial Intelligence 156, no. 1, pages 75–111.

[23] Jan Bosch. 2000. Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach. Addison-Wesley, Boston (MA).

[24] Jan Bosch (editor). 2004. 2nd Groningen Workshop on Software Variability Management: Software Product Families and Populations. University of Groningen, Groningen, The Netherlands.

[25] Jan Bosch. 2004. Software Variability Management. In: Nord [90], pages 315–316.

[26] Jan Bosch. 2004. Software Variability Management (Introduction to Special Issue on Software Variability Management). Science of Computer Programming 53, no. 5, pages 255–258.

[27] Jan Bosch and Charles Krueger (editors). 2004. 8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR 2004), volume 3107 of *Lecture Notes in Computer Science*. Springer.

[28] George E. B. Box and Norman R. Draper. 1987. Empirical Model-building and Response Surfaces. John Wiley & Sons.

[29] Ronald J. Brachman and Hector J. Levesque (editors). 1985. Readings in Knowledge Representation. Morgan Kaufman.

[30] Alexandre Braganca and Ricardo J. Machado. 2007. Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. In: 11th International Software Product Line Conference (SPLC 2007) [61], pages 3–12.

[31] Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt (editors). 1984. On Conceptual Modelling—Perspectives from Artificial Intelligence, Databases, and Programming Languages, Topics in Information Systems. Springer.

[32] Vaclav Cechticky, Alessandro Pasetti, O. Rohlik, and Walter Schaufelberger. 2004. XML-Based Feature Modelling. In: Bosch and Krueger [27], pages 101–114.

[33] Peter P. Chen. 1976. The Entity-Relationship Model—Toward a Unified View of Data. ACM Transactions on Database Systems 1, no. 1, pages 9–36.

[34] Paul C. Clements and Linda Northrop. 2001. Software Product Lines—Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, Boston (MA).

[35] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: R. Glück and M. Lowry (editors), 4th International Conference on Generative Programming and Component Engineering (GPCE), pages 422–437.

[36] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. 2002. Generative Programming for Embedded Software: An Industrial Experience Report. In: Don S. Batory, Charles Consel, and Walid Taha (editors), ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer.

[37] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. Generative Programming. Addison-Wesley, Boston (MA).

[38] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2004. Staged Configuration Using Feature Models. In: Nord [90], pages 266–283.

[39] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing Cardinality-based Feature Models and Their Specialization. Software Process: Improvement and Practices 10, no. 1, pages 7–29.

[40] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practices 10, no. 2, pages 143–169.

[41] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-based Feature Modeling and Constraints: A Progress Report. In: International Workshop on Software Factories at OOPSLA 2005.

[42] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In: 11th International Software Product Line Conference (SPLC 2007) [61], pages 23–34.

[43] Eric Dashofy, André van der Hoek, and Richard M. Taylor. 2002. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In: 24th International Conference on Software Engineering (ICSE 2002), pages 266–276. ACM.

[44] Eric Dashofy, André van der Hoek, and Richard M. Taylor. 2005. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engineering and Methodology 14, no. 2, pages 199–245.

[45] Sybren Deelstra, Marco Sinnema, and Jan Bosch. 2005. Product Derivation in Software Product Families: A Case Study. Journal of Systems and Software 74, no. 2, pages 173–194.

[46] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. 1998. The UML as a Formal Modeling Notation. In: Jean Bézivin and Pierre-Alain Muller (editors), The First International Workshop on The Unified Modeling Language (UML '98), volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer.

[47] Boi Faltings and Eugene C. Freuder. 1998. Special Issue on Configuration. IEEE Intelligent Systems 14, no. 4, pages 29–85.

[48] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. 2001. Conceptual Modeling for Configuration of Mass-Customizable Products. Artificial Intelligence in Engineering 15, no. 2, pages 165–176.

[49] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. 2006. Model-driven Development Using UML 2.0: Promises and Pitfalls. Computer 39, no. 2, pages 59–66.

[50] David Garlan. 2001. Software Architecture. In: John J. Marciniak (editor), Encyclopedia of Software Engineering. John Wiley & Sons, New York.

[51] David Garlan, Robert T. Monroe, and David Wile. 1997. Acme: An Architecture Description Interchange Language. In: J. Howard Johnson (editor), The 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97).

[52] Cesar Gonzalez-Perez and Brian Henderson-Sellers. 2006. A Powertype-based Metamodelling Framework. Software and Systems Modeling 5, no. 1, pages 72–90.

[53] Cesar Gonzalez-Perez and Brian Henderson-Sellers. 2007. Modelling Software Development Methodologies: A Conceptual Foundation. Journal of Systems and Software 80, no. 11, pages 1778–1796.

[54] Martin Griss, John Favaro, and Massimo d'Alessandro. 1998. Integrating Feature Modelling with the RSEB. In: Fifth International Conference on Software Reuse, pages 76–85. IEEE Computer Society.

[55] Anthony Hall. 1990. Seven Myths of Formal Methods. IEEE Software 7, no. 5, pages 11–19.

[56] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer 37, no. 10, pages 64–72.

[57] Mikko Heiskala, Juha Tiihonen, Timo Soininen, and Andreas Anderson. 2006. Four-worlds Model for Configurable Services. In: Joint Conference of International Mass Customization Meeting (IMCM'06) and International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems (PETO'06), pages 199–216.

[58] Brian Henderson-Sellers and Cesar Gonzalez-Perez. 2006. Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0. In: Nierstrasz et al. [89], pages 16–26.

[59] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. Design Science in Information Systems Research. MIS Quarterly 28, no. 1, pages 75–105.

[60] Lothar Hotz, Katarina Wolter, Thorsten Krebs, Sybren Deelstra, Jos Nijhuis, and John MacGregor. 2006. Configuration in Industrial Product Families—The ConIPF Methodology. IOS Press.

[61] IEEE Computer Society. 2007. 11th International Software Product Line Conference (SPLC 2007).

[62] Mikoláš Janota and Joseph Kiniry. 2007. Reasoning about Feature Models in Higher-Order Logic. In: 11th International Software Product Line Conference (SPLC 2007) [61], pages 13–22.

[63] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and Spencer A. Peterson. 1990. Feature-Oriented Domain Analysis (FODA)—Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

[64] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering 5, pages 143–168.

[65] Eero Kasanen, Kari Lukka, and Arto Siitonen. 1993. The Constructive Approach in Management Accounting Research. Journal of Management Accounting Research 5, pages 243–264.

[66] Henry Kautz and Bart Selman. 1992. Planning as Satisfiability. In: 10th European Conference on Artificial intelligence (ECAI '92), pages 359–363. John Wiley & Sons.

[67] Bryan M. Kramer, Vinay K. Chaudhri, Manolis Koubarakis, Thodoros Topaloglou, Huaiqing Wang, and John Mylopoulos. 1991. Implementing Telos. ACM SIGART Bulletin 2, no. 3, pages 77–83.

[68] Thomas Kühne. 2006. Clarifying Matters of (Meta-) Modeling: An Author's Reply. Software and Systems Modeling 5, no. 4, pages 395–401.

[69] Thomas Kühne. 2006. Matters of (Meta-) Modeling. Software and Systems Modeling 5, no. 4, pages 369–385.

[70] Thomas Kühne and Daniel Schreiber. 2007. Can Programming Be Liberated from the Two-Level Style: Multi-Level Programming with DeepJava. In: Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (editors), 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA '07), pages 229–244. ACM.

[71] Kevin Lano. A Compositional Semantics of UML-RSDS. Software and Systems Modeling, in press. DOI: 10.1007/s10270-007-0064-x.

[72] Craig Larman. 2002. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edition. Prentice-Hall.

[73] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. 1995. Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering 21, no. 4, pages 336–355.

[74] Jochen Ludewig. 2003. Models in Software Engineering—An Introduction. Software and Systems Modeling 2, no. 1, pages 5–14.

[75] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In: Gary J. Chastek (editor), Second International Conference on Software Product Lines (SPLC2), volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer.

[76] Tomi Männistö. 2000. A Conceptual Modelling Approach to Product Families and Their Evolution. Ph.D. thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland.

[77] Tomi Männistö and Jan Bosch (editors). 2004. Software Variability Management for Product Derivation—Towards Tool Support, a workshop in SPLC 2004. Helsinki University of Technology, Helsinki University of Technology, Espoo, Finland.

[78] William E. McUmber and Betty H. C. Cheng. 2001. A General Framework for Formalizing UML with Formal Languages. In: 23rd International Conference on Software Engineering (ICSE 2001), pages 433–442.

[79] Nenad Medvidovic, Peyman Oreizy, James E. Robbins, and Richard M. Taylor. 1996. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In: 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 24–32.

[80] Nenad Medvidovic and Richard M. Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26, no. 1, pages 70–93.

[81] 2006. Meta Object Facility (MOF) Core Specification, OMG Available Specification, version 2.0. Technical Report formal/06-01-01, Object Management Group.

[82] Bertrand Meyer. 1990. Introduction to the Theory of Programming Languages. Prentice Hall, New York.

[83] Tomi Männistö, Eila Niemelä, and Mikko Raatikainen (editors). 2007. Software and Services Variability Management Workshop—Concepts, Models and Tools.

[84] 2003. MDA Guide version 1.0.1. Technical report, Object Management Group (OMG).

[85] Varvana Myllärniemi. 2004. Kumbang Configurator—A Tool for Configuring Software Product Families. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering.

[86] John Mylopoulos. 1992. Conceptual modeling and Telos. In: P. Loucopoulos and R. Zicari (editors), Conceptual Modeling, Databases, and CASE, pages 49–68. John Wiley & Sons.

[87] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. 1990. Telos: Representing Knowledge about Information Systems. ACM Transactions on Information Systems 8, no. 4, pages 325–362.

[88] Ilkka Niemelä. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence 25, no. 3-4, pages 241–273.

[89] Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (editors). 2006. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), volume 4199 of *Lecture Notes in Computer Science*. Springer.

[90] Robert L. Nord (editor). 2004. Third Software Product Line Conference (SPLC 2004), volume 3154 of *Lecture Notes in Computer Science*. Springer.

[91] Henk Obbink and Klaus Pohl (editors). 2005. 9th International Software Product Line Conference (SPLC 2005), volume 3714 of *Lecture Notes in Computer Science*. Springer.

[92] 2006. Object Constraint Language, OMG Available Specification, version 2.0. Technical Report formal/06-05-01, Object Management Group.

[93] James Odell. 1994. Power Types. Journal of Object-Oriented Programming 7, no. 2, pages 8–12.

[94] Antoni Olivé. 2003. Derivation Rules in Object-Oriented Conceptual Modeling Languages. In: Johann Eder and Michele Missikoff (editors), 15th International Conference on Advanced Information Systems Engineering (CaISE 2003), volume 2681 of *Lecture Notes in Computer Science*, pages 404–420. Springer.

[95] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. 2007. Metamodel-based Model Conformance and Multiview Consistency Checking. ACM Transactions on Software Engineering and Methodology 16, no. 3.

[96] Chong-Mok Park, Seokjin Hong, Kyoung-Ho Son, and Jagun Kwon. 2007. A Component Model Supporting Decomposition and Composition of Consumer Electronics Software Product Lines. In: 11th International Software Product Line Conference (SPLC 2007) [61], pages 181–192.

[97] Hannu Peltonen, Tomi Männistö, Kari Alho, and Reijo Sulonen. 1994. Product Configurations—An Application for Prototype Object Approach. In: Mario Tokoro and Remo Pareschi (editors), 8th European Conference for Object-Oriented Programming (ECOOP 94), volume 821 of *Lecture Notes in Computer Science*, pages 513–534. Springer.

[98] Klaus Pohl, Patrick Heymans, Kyo-Chul Kang, and Andreas Metzger (editors). 2007. First International Workshop on Variability Modelling of Software-intensive Systems (VaMoS).

[99] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. In: Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002).

[100] André L. Santos, Kai Koskimies, and Antónia Lopes. 2006. A Model-Driven Approach to Variability Management in Product-Line Engineering. Nordic Journal of Computing 13, no. 3, pages 196–213.

[101] Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. Computer 39, no. 2, pages 25–31.

[102] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In: 14th IEEE International Requirements Engineering Conference (RE 2006), pages 136–145. IEEE Computer Society.

[103] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. Computer Networks 51, no. 2, pages 456–479.

[104] Ed Seidewitz. 2003. What Models Mean. IEEE Software 20, no. 5, pages 26–32.

[105] Patrik Simons, Ilkka Niemelä, and Timo Soininen. 2002. Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, no. 1-2, pages 181–234.

[106] Brian C. Smith. 1982. Reflection and Semantics in a Procedural Language. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

[107] Timo Soininen. 2000. An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks. Ph.D. thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland.

[108] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. 2001. Representing Configuration Knowledge with Weight Constraint Rules. In: AAAI

Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning.

[109] Timo Soininen and Markus Stumptner. 2003. Introduction to Special Issue on Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17, no. 1-2, pages 1–2.

[110] Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen. 1998. Towards a General Ontology of Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12, no. 4, pages 357–372.

[111] Herbert Stachowiak. 1973. Allgemeine Modelltheorie. Springer, Wien–New York.

[112] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. Data & Knowledge Engineering 35, no. 1, pages 83–106.

[113] Friedrich Steimann. 2007. The Role Data Model Revisited. Applied Ontology 2, no. 1, pages 89–103.

[114] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques. Software—Practice and Experience 35, no. 8, pages 705–754.

[115] Tommi Syrjänen. 2001. Omega-restricted Logic Programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, volume 2713 of *Lecture Notes in Artificial Intelligence*, pages 267–280. Springer.

[116] Arthur H. M. ter Hofstede and H. A. Proper. 1998. How to Formalize it? Formalization Principles for Information System Development Methods. Information and Software Technology 40, no. 10, pages 519–540.

[117] Bernhard Thalheim. 1992. Fundamentals of Cardinality Constraints. In: Günther Pernul and A Min Tjoa (editors), 11th International Conference on the Entity-Relationship Approach (ER '92), volume 645 of *Lecture Notes in Computer Science*, pages 7–23.

[118] Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen, and Asko Riihihuhta. 1998. Modeling Configurable Product Families. In: 4th WDK Workshop on Product Structuring. Delft University of Technology.

[119] 2007. Unified Modeling Language: Infrastructure, version 2.1.1. Technical Report formal/2007-02-06, Object Management Group (OMG).

[120] 2005. Unified Modeling Language: Superstructure, version 2.0. Technical Report formal/05-07-04, Object Management Group (OMG).

[121] 2007. Unified Modeling Language: Superstructure, version 2.1.1. Technical Report formal/2007-02-05, Object Management Group (OMG).

[122] 2001. OMG Unified Modeling Language Specification, version 1.4. Technical report, Object Management Group (OMG).

[123] André van der Hoek. 2004. Design-time Product Line Architectures for Any-Time Variability. Science of Computer Programming 53, no. 3, pages 285–304.

[124] Tijs van der Storm. 2004. Variability and Component Composition. In: Bosch and Krueger [27], pages 157–166.

[125] Jilles van Gurp and Jan Bosch (editors). 2003. Software Variability Management Workshop, volume IWI preprint 2003-7-01. University of Groningen, Groningen, The Netherlands.

[126] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pages 45–54. IEEE Computer Society.

[127] Pascal van Hentenryck. 1999. The OPL Optimization Programming Language. MIT Press.

[128] Rob van Ommering. 2002. Building Product Populations with Software Components. In: 24th International Conference on Software Engineering (ICSE 2002), pages 255–265.

[129] Rob van Ommering. 2004. Building Product Populations with Software Components. Ph.D. thesis, University of Groningen, The Netherlands.

[130] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. 2000. The Koala Component Model for Consumer Electronics Software. IEEE Computer 33, no. 3, pages 78–85.

[131] Michael von der Beeck. 2006. A Formal Semantics of UML-RT. In: Nierstrasz et al. [89], pages 768–782.

[132] Thomas von der Maßen and Horst Lichter. 2005. Determining the Variation Degree of Feature Models. In: Obbink and Pohl [91], pages 82–88.

[133] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In: Jim Davies, Wolfram Schulte, and Michael Barnett (editors), 6th International Conference on Formal Engineering Methods (ICFEM 2004), volume 3308 of *Lecture Notes in Computer Science*, pages 115–130. Springer.

ISBN 978-951-22-9484-8
ISBN 978-951-22-9485-5 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)