Integrated Management of Variability in Space and Time in Software Families

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Christoph Seidl
geboren am 05.12.1982 in Freiburg im Breisgau

Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden, Deutschland)

Zweitgutachter
Associate prof. Andrzej Wąsowski, PhD
(IT-Universitetet i København, Dänemark)

Dresden im Juli 2015

Confirmation

I confirm that I independently prepared this thesis with the title *Integrated Management of Variability in Space and Time in Software Families* and that I used only the references and auxiliary means indicated in the thesis.

Dresden, July 15, 2015

Dipl.-Inf. Christoph Seidl

Abstract

Software families, such as Software Product Lines (SPLs) or Software Ecosystems (SECOs), comprise a set of closely related software systems in terms of configurable functionality (*variability in space*). A *variability model* represents configuration knowledge on a conceptual level and a *variability realization mechanism* manifests it in realization assets for a concrete configuration selected during a *variant derivation procedure.* When using *feature models* as variability model, a conceptual configuration consists of a valid subset of the available features. With *delta modeling* as variability realization mechanism, the configuration would be resolved to a set of delta modules that each contain transformation operations to alter realization assets (e.g., source code). Through this procedure, a *base variant* of the software family is transformed to a particular *target variant* for the specified configuration.

Over the course of time, software families are subjected to change as part of software evolution (*variability in time*) when new requirements are implemented or defects are fixed. Both dimensions of variability may affect each other so that they cannot generally be handled in isolation for all software families, e.g., if evolution modifies available configuration options. However, current techniques for managing software families focus on the dimension of variability in space, but neglect the dimension of variability in time or treat it as a separate challenge. This makes it impossible to provide multiple versions of configurable functionality and to combine them with different versions of the rest of the software family.

The work of this thesis provides remedy to this problem by presenting an approach for integrated management of variability in space and time in software families. The main contributions of the thesis can be distinguished into three key areas: a variability model, a variability realization mechanism and a variant derivation procedure.

As **variability model**, feature models are extended to *Hyper-Feature Models (HFMs)* to model configuration information related to variability in space and time as features with multiple versions arranged along development lines. Furthermore, a *version-aware constraint language* is created to express dependencies on and incompatibilities with versions and version ranges.

As **variability realization mechanism**, delta modeling is extended by introducing an explicit distinction of configuration and *evolution delta modules*, where the latter may utilize more powerful transformation operations to alter realization artifacts. Furthermore, a *language creation infrastructure* is devised to ease creation of interoperable delta languages for arbitrary languages that can be used to alter the respective realization artifacts.

As **variant derivation procedure**, a conceptual configuration of features and versions from an HFM is resolved to a set of required delta modules by means of a mapping model. An *automatic version selection procedure* reduces the complexity of creating configurations of HFMs to that for common feature models by completing a valid selection of features with a suitable constellation of versions according to a particular strategy. Furthermore, a large part of the delta modules' *application order is derived* from the structure of the HFM regarding the associated features and versions. Applying the required delta modules to a base variant of the software family in the determined sequence yields a target variant that encompasses aspects of variability in space and time through selected functionality at a particular revision.

With the presented approach for integrated management of variability in space and time, it is possible to permit extensions by multiple vendors with independent release cycles, to maintain compatibility with older releases of (parts of) the software family and to empower users by letting them decide on the extent and point in time for updates of variable functionality.

All concepts of the thesis are implemented as tool suite DeltaEcore using a metamodel-based approach to specify and process artifacts. The concepts of the thesis and their implementation are evaluated by applying them in three case studies: a configurable driver software for the domestic service robot TurtleBot, a metamodel family for role-based modeling and programming languages as well as an SPL of feature modeling notations and constraint languages. As conclusion of the thesis, design decisions of the approach and their implications are discussed and possible future application areas are outlined.

Acknowledgements

In the three year process of working on this thesis, a large number of people have supported me in different ways. The following lines give sincere thanks to the people most closely related to the dissertation.

First of all, I thank my supervisor, Uwe Aßmann, for giving me the chance to work on my dissertation, for providing tremendous amounts of freedom to follow up on my own ideas and for being enthusiastic about successes even if the journey took an unexpected turn. I further thank my subject specialist, Thomas Schlegel, whose interest in me and my work has encouraged me personally as well as scientifically. I thank my reviewer, Andrzej Wąsowski, for taking on the effort of reviewing my dissertation and traveling to Dresden to attend the defense. I also thank the European Social Fund (ESF) and the Federal State of Saxony who provided funding for my work on project VICCI (#100098171), which aligned well with the topic of my thesis.

I thank my colleagues in the software engineering group of TU Dresden for their academic feedback on my work and for providing views from different angles. I especially thank Georg Püschel, Julia Schroeter and Florian Heidenreich for their discussions on Software Product Line topics. I also thank Jan Reimann and René Schöne for assisting with various infrastructural services as well as Christian Piechnick for providing the initial version of the TurtleBot driver software. I further thank Birgit Demuth and Katrin Heber for being sources of accurate information and interesting stories alike. In addition, I thank my colleagues Romina Kühn, Ronny Seiger, Alexandra Funke and Florian Niebling from the software engineering for ubiquitous systems group of TU Dresden for adding the “fun” part to the daily grind.

I especially thank Ina Schaefer for paving my way into the Software Product Line community, for her ongoing interest in my work and for providing an environment in which I flourish academically–both in the past and in the future. I further thank my colleagues from various different institutions around the world for scientific discussion as well as for good times at conferences including Sandro Schulze, Mathias Schubanz, Rolf-Helge Pfeiffer, Vander Alves, Tim Winkelmann and Daniela Lettner. I also thank my students who contributed to the contents of this thesis either by interest in the concepts or parts of their implementation in the tool suite including Sven Schuster, Michael Nieke, David Gollasch, Florian Weidner and Dimitri Henkel.

I thank my friends Thordis, Thomas, Conny, Robert, Lena, Kai-Uwe and Silke for being the backbone of my social well-being in Dresden and for getting me out of the office every so often ensuring my basic nutrition by visiting my favorite restaurants. I further thank my friends Markus, Stephanie, Peter, Andreas Bi., Joachim, Stephan, Paul, Andreas Bu., Alan and Christian for providing a place to feel home in Munich and some highly-valued rituals.

I thank my parents for fostering my education and for instilling into me a general desire to strive for excellence. I especially thank my mother Angela and my brother Matthias for being a rock of empathy concerning life beside the dissertation and, even more, for both their personal sacrifices that enabled me to complete both my studies and my dissertation.

Finally, I thank my beloved girlfriend Irena for her continued moral and emotional support in times of hardship and defeat, her understanding and patience in times of frustration and intensive work, her genuine interest and sincerely shared joy in times of success, and, most of all, for being the person she is!

Publications

This doctoral thesis is based on the following peer-reviewed publications ordered by relevance, starting with the most relevant.

[1] **Christoph Seidl**, Ina Schaefer, and Uwe Aßmann. Integrated Management of Variability in Space and Time in Software Families. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*, SPLC'14, 2014.

[2] **Christoph Seidl**, Ina Schaefer, and Uwe Aßmann. DeltaEcore-A Model-Based Delta Language Generation Framework. In *Modellierung*, Modellierung'14, 2014.

[3] **Christoph Seidl**, Ina Schaefer, and Uwe Aßmann. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'14, 2014.

[4] Thomas Kühn, Max Leuthäuser, Sebastian Götz, **Christoph Seidl**, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE)*, SLE'14, 2014.

[5] **Christoph Seidl**, Ina Schaefer, and Uwe Aßmann. Variability-Aware Safety Analysis using Delta Component Fault Diagrams. In *Proceedings of the 4th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, FMSPLE'13, 2013.

[6] **Christoph Seidl** and Uwe Aßmann. Towards Modeling and Analyzing Variability in Evolving Software Ecosystems. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'13, 2013.

Further publications directly related to the thesis.

[7] **Christoph Seidl**, Florian Heidenreich, and Uwe Aßmann. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, SPLC'12, 2012.

[8] Benjamin Klatt, Klaus Krogmann, and **Christoph Seidl**. Program Dependency Analysis for Consolidating Customized Product Copies. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, ICSME'14, 2014.

[9] **Christoph Seidl** and Irena Domachowska. Teaching Variability Engineering to Cognitive Psychologists. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, SPLC'14, 2014.

Further peer reviewed publications indirectly related to this thesis.

[10] Georg Püschel, Christian Piechnick, Sebastian Götz, **Christoph Seidl**, Sebastian Richly, and Uwe Assmann. A Black Box Validation Strategy for Self-adaptive Systems. In *Proceedings of The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*. XPS Press, 2014.

[11] Georg Püschel, Christian Piechnick, Sebastian Götz, **Christoph Seidl**, Sebastian Richly, and Uwe Aßmann. A Combined Simulation and Test Case Generation Strategy for Self-Adaptive Systems. *International Journal on Advances in Software*, 7(3-4), 2014.

[12] Martin Franke, **Christoph Seidl**, and Thomas Schlegel. A Seamless Integration, Semantic Middleware for Cyber-Physical Systems. In *Proceedings of the 10th International Conference on Networking, Sensing and Control (ICNSC)*, ICNSC'13, 2013.

Contents

List of Abbreviations

AOP	Aspect-Oriented Programming	32
API	Application Programming Interface	38
AST	Abstract Syntax Tree	45
ATL	Atlas Transformation Language	46
CFD	Component Fault Diagram	14
ChOP	Change-Oriented Programming	92
CL	Checklist	14
CLP	Constraint Logic Programming	65
CMOF	Complete Meta-Object Facility (MOF)	43
CSP	Constraint Satisfaction Problem	65
CVL	Common Variability Language	24
CVS	Concurrent Versions System	94
DSPL	Dynamic Software Product Line	225
EMF	Eclipse Modeling Framework	43
EMOF	Essential MOF	43
ETL	Epsilon Transformation Language	46
FOP	Feature Oriented Programming	32
FST	Feature Structure Tree	114
GEF	Graphical Editing Framework	45
GMF	Graphical Modeling Framework	45
GSN	Goal Structuring Notation	14
HFM	Hyper-Feature Model	4
Hismo	History Metamodel	91
IDE	Integrated Development Environment	2
ISC	Invasive Software Composition	34
JAR	Java Archive	13
LOC	Lines of Code	112
MDSD	Model-Driven Software Development	43
MOF	Meta-Object Facility	43
OCL	Object Constraint Language	193
OBR	OSGi Bundle Repository	94

OMG	Object Management Group	43
OSGi	Open Services Gateway Initiative	40
OVM	Orthogonal Variability Model	24
PDF	Portable Document Format	1
QVT	Query View Transformation	46
SECO	Software Ecosystem	3
SFT	Software Fault Tree	14
SPL	Software Product Line	36
SVN	Subversion	3
Tecmo	Technical Ecosystem Modeling	91
URI	Uniform Resource Identifier	108
VML*	Variability Modeling Language Family	139
VSpec	Variability Specification	24
XML	Extensible Markup Language	13

Need for Integrated Management of Variability in Space and Time in Software Families

A software family encompasses a set of individual software systems that have large commonalities but still are different. An example of a software family is Microsoft Office with individual products such as Word, Excel or PowerPoint. Despite the different intended usage areas of the individual applications, their common parts can be observed in functionality such as support for different fonts or the export to Portable Document Format (PDF). Figure 0.1 depicts Microsoft Office as example of a software family.

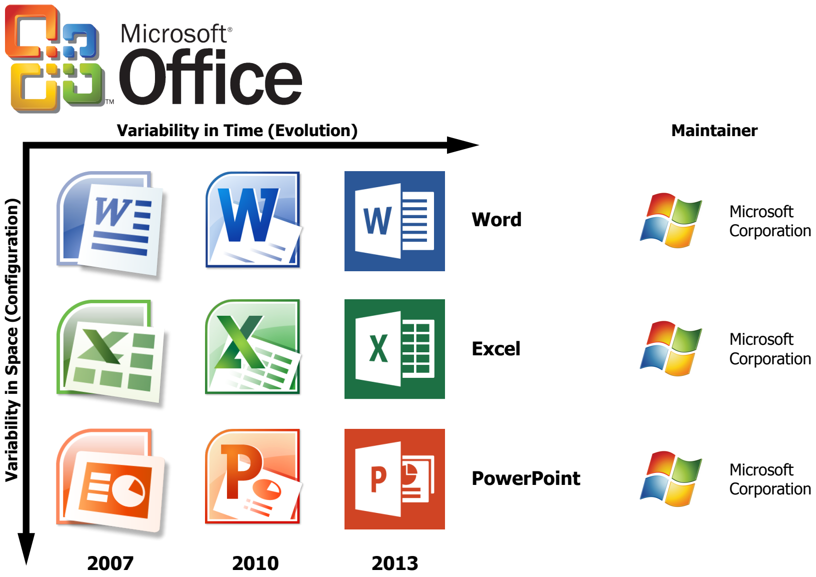


Figure 0.1.: Microsoft Office as example of a software family subjected to variability in space and time where both dimensions can be handled in isolation.

When employing a structured reuse mechanism, individual parts of a software family may be captured within reusable assets that can be combined with the functionality common to all products. Configuration rules govern which of the possible combinations are considered valid, e.g., due to constraints imposed by economic or technological concerns. A *configuration* procedure is used to create individual products of the software family, e.g., Word, Excel or PowerPoint of the software family Microsoft Office. The entirety of all possible applications created by combining functionality described by variable assets creates the dimension of *variability in space*.

Over the course of time, software families and their variable assets have to change to meet new requirements or fix defects as part of software evolution [Leh80] yielding new versions. Hence, not only the dimension of variability in space (configuration) but also the dimension for *variability in time* [PBvdL05, BFG⁺02] (evolution) has to be handled. In the case of Microsoft Office, the effects of evolution manifest in the individual products creating versions such as Word 2007, Word 2010 and Word 2013.

Problem

In the example of Microsoft Office, the dimensions of variability in space and time may be separated so that they can be treated (mostly) in isolation by software developers. This is due to the fact that the foundation as a software family on the software engineering level is transparent to end-users as they are solely concerned with individual products. As a consequence, end-users may principally combine different *products* and different *versions of products* but they cannot combine different *variable functionality* of products or different *versions of variable functionality*. Hence, it is possible to decide to use Word and Excel but not PowerPoint as well as to use Word 2010 with Excel 2013, but it is not possible to use the spreadsheet calculation functionality of Excel (directly) within Word or the revised version of the PDF export of a later version with the base functionality of an older version.

For the Microsoft Office software family, this appears to be a sensible compromise between maintenance effort of software developers and flexibility of customers with regard to combination of different functionality in different versions. This is especially valid due to the fact that the individual products of Microsoft Office can be regarded as commodity software that is supposed to meet the requirements of all potential users. However, there are other software families that permit greater flexibility in terms of configurations that more closely meet individual needs.

For example, the Eclipse Integrated Development Environment (IDE)[1], the Android mobile application operating system[2] and its apps as well as the kernel of the Linux operating system[3] each form a software family that exposes (a significant part of) the configuration process to end-users. This gives end-users increased flexibility in tailoring the resulting products to individual user needs [SSA13, BPT$^+$14]. For the Eclipse IDE, a common platform may be extended by installing different plug-ins and pre-bundled releases provide default configurations as base products for different use cases (e.g., software development with the Java language). For Android, the operating system itself may be customized by vendors (or even end-users) before it serves as base platform for various apps that extend or alter functionality of the respective product to meet individual requirements. For the Linux kernel, it is possible to select from a set of reusable modules to incorporate different functionality or to support various hardware platforms within the resulting products.

Each of these approaches utilizes a concept for reusable variable functionality (i.e., plug-ins, apps or kernel modules, respectively) that imposes constraints on possible configurations (e.g., when use of one plug-in demands presence of another one). The individual variable assets may be maintained by different vendors [BPT$^+$14, McG09a] and do not necessarily have synchronized evolution cycles [BBS10a, McG09a, SA13]. Hence, new versions of the common functionality and its extensions may be created independently and, in part, even without explicit knowledge of either one of the parties (e.g., the maintainers of the common functionality may not be aware of each new version of each extension).

However, new versions of variable assets may introduce dependencies and incompatibilities that have to be integrated into the configuration knowledge. In this case, a strict separation of variability in space and variability in time is *not* possible as they are intertwined: Evolution may be performed on the level of variable assets and the change of these variable assets may change options for configuration. For example, in the Eclipse IDE, a plug-in may be developed further so that the new version of the plug-in has additional dependencies (e.g., a new requirement for another plug-in) or incompatibilities (e.g., a dependency on a new version of the base

[1] http://eclispe.org
[2] http://android.com
[3] https://kernel.org

platform). Hence, when trying to select the new *version* of the plug-in, the configuration might have to be adapted for the variable assets as well.

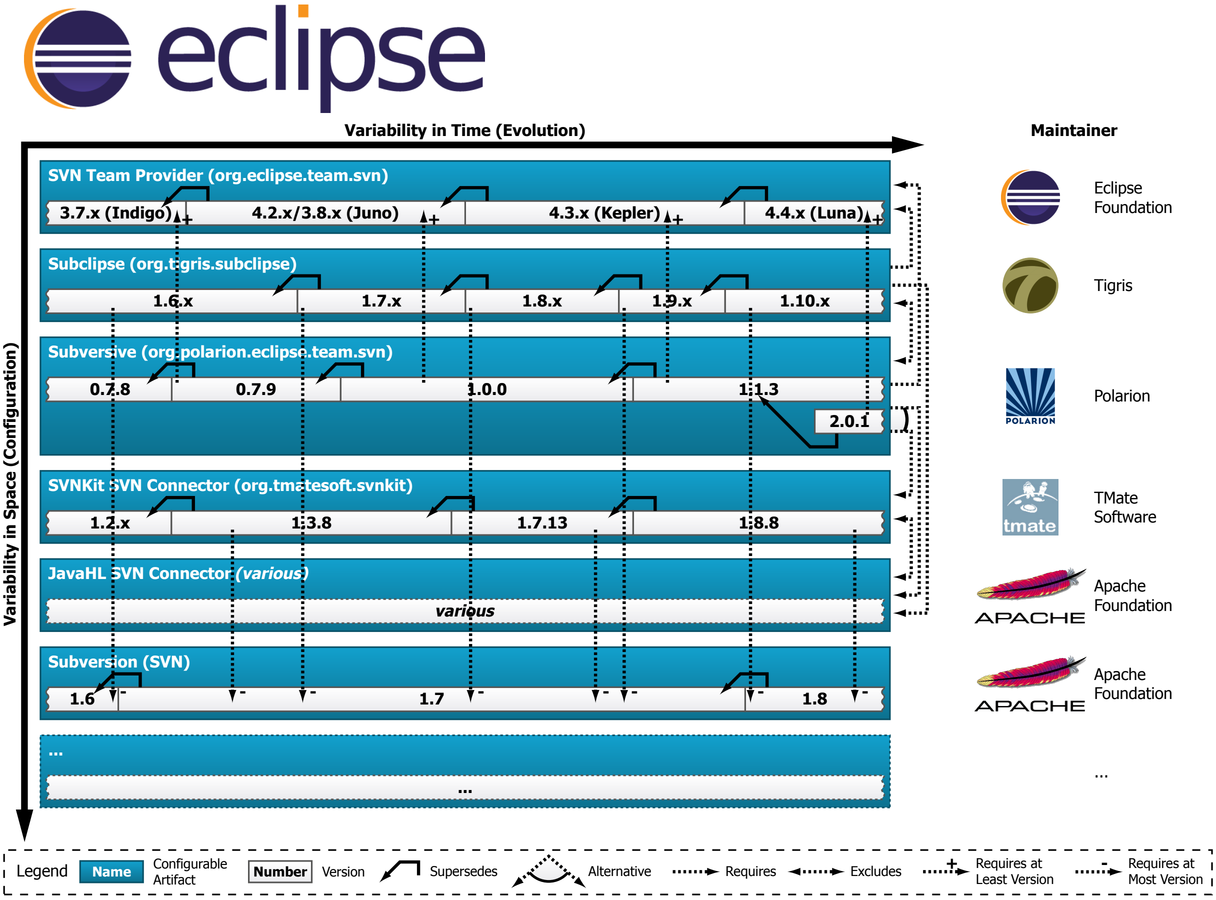


Figure 0.2.: Part of the Eclipse SECO as example of a software family subjected to variability in space and time where both dimensions cannot be handled in isolation.

Figure 0.2 shows an excerpt of the Eclipse Software Ecosystem (SECO) as example of a software family where variability in space and time have an influence on each other and, thus, cannot be treated in isolation. The example illustrates the *Subversive* and *Subclipse* extensions, which are used to connect to repositories of the Subversion (SVN) source code management system, as well as (part of) their dependencies. Both these extensions depend on the *SVN Team Provider* of the Eclipse platform and an SVN connector, such as the *SVNKit SVN Connector* or the *JavaHL SVN Connector*. Various different vendors maintain the individual extensions and release new versions at different, largely unsynchronized release cycles. Dependencies of versions of configurable artifacts on other artifacts or their versions change with evolution. Furthermore, end-users of Eclipse are empowered to perform configuration of artifacts and their versions to retrieve a concrete product. Hence, the Eclipse SECO constitutes an example of a software family where variability in space and time *cannot* be treated in isolation. However, no approach has been established on how to incorporate the notion of evolution and different versions into the variant derivation process of a software family.

Envisaged Solution

To remedy the aforementioned problems, this thesis proposes an integrated approach for handling variability in space and time in software families. The basis for this approach is formed by established concepts for handling variability in space within software families. Feature models [KCH$^+$90, CHE05] are used as basis for conceptual modeling of configuration knowledge and delta modeling [Sch10, CHS10] as basis for manifesting changes in realization assets.

These concepts are extended for integrated handling of variability in time by contributions in the three key areas of a variability model, a variability realization mechanism and a variant derivation procedure. Figure 0.3 visualizes the contributions in their respective areas. As a variability model for variability in space and time, feature models are extended to Hyper-Feature Models (HFMs) that introduce a version concept for individual features and specify chronological relations along development lines. Furthermore, a version-aware constraint language is introduced to specify interdependencies and incompatibilities of feature versions and version ranges. As a variability realization mechanism, delta modeling is extended by a dedicated concept for evolution delta modules encapsulating changes associated with variability in time. Furthermore, a delta language creation infrastructure is introduced that utilizes these concepts and enables creation of custom delta languages for a wide variety of different and potentially changing source languages. As part of a variant derivation procedure, HFMs and the extension to delta modeling are combined using a delta module association. A suitable application order for the required delta modules is derived mostly automatically from the employed HFM. Furthermore, a procedure is provided that allows automatic detection of suitable version constellations for a selection of features. Hence, the effort of configuring products containing both variability in space and time with HFMs is reduced to a similar level as configuring only variability in space using feature models.

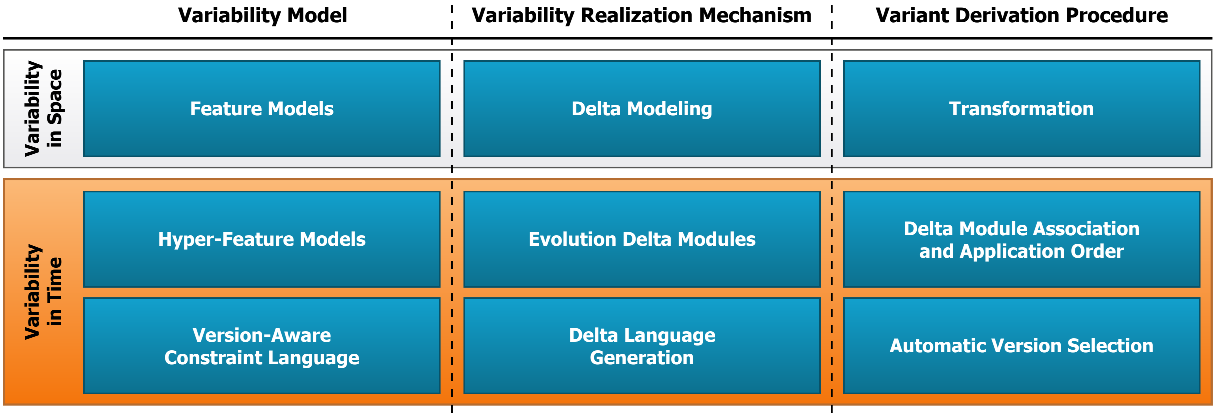

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Hyper-Feature Models
Evolution Delta Modules
Delta Module Association and Application Order
Version-Aware Constraint Language
Delta Language Generation
Automatic Version Selection

Figure 0.3.: Overview of the contributions of the thesis in the area of integrated management of variability in space and time in software families.

This approach has certain benefits for both software developers and end-users of the software family and its products. For software developers, only one set of technologies is required to cope with both variability in space and time instead of having to utilize different approaches and tools. The approach fosters decentralized development of individual variable assets of a software family by different vendors with individual and potentially unsynchronized release cycles. Furthermore, the approach explicitly captures dependencies and incompatibilities of both variable assets and their versions on a conceptual level enabling procedures such as various analyses or an automatic version configuration. In addition, the approach supports the maintenance of version histories for variable extensions and their associated configuration rules with regard to the software family. Due to the connection of the conceptual and the realization level of both variability in space and time within a variant derivation procedure, it is principally possible to trace the presence of an element in a realization asset of a product back to both its feature and the respective version of the feature. This is essential for all generative model-based development when models and the generated source code have to be synchronized and may further be utilized for the realization of safety-critical software (see Section 9.2).

For end-users, the approach allows a fine-grained control over selection of both functionality and revisions of variable assets within products of a software family. Due to this reason, highly customized products are possible that respect individual customer needs regarding functionality as well as preferences towards certain versions of individual assets (e.g., due to integration into an internal infrastructure that requires a particular version). In consequence, the approach allows a more flexible behavior of end-users regarding updates as with software families such as the aforementioned Microsoft Office: Instead of being able to choose between mere versions of entire products, constellations of individual features in different versions are permitted. Nevertheless, the approach can be utilized while exposing versioning to end-users only on a product level as with the products of Microsoft Office.

Outline

The thesis is structured into three parts: Part I describes context and preliminaries of the thesis consisting of 3 chapters. Chapter 1 introduces a configurable driver software for the domestic service robot TurtleBot used as running example for all concepts throughout the thesis. Chapter 2 defines terminology of software families and explains existing notations for handling variability in space. Chapter 3 reasons for a selection of these notations that is most suitable as basis for an extension to handle variability in time in software families and explains them in detail.

Part II presents an integrated approach for managing variability in space and time as contribution of the thesis within 3 chapters. Chapter 4 introduces Hyper-Feature Models (HFMs) as a variability model for integrated handling of variability in space and time and introduces the version-aware constraint language. Chapter 5 makes the variability realization mechanism delta modeling suitable for the use with variability in time by introducing explicit evolution delta modules and providing concepts for generating delta languages for arbitrary types of potentially changing source languages. Chapter 6 combines HFMs with this extension to delta modeling by deriving an application order of delta modules from an HFM and a defining procedure for automatic version selection in order to allow derivation of variants with aspects of variability in space and time.

Part III presents the realization and application of the developed concepts in order to validate the feasibility of the integrated approach for managing variability in space and time within 3 chapters. Chapter 7 presents the model-based realization of all concepts of the thesis as a tool suite called DeltaEcore. Chapter 8 applies the tool suite to realize three individual case studies on software families whose artifacts were subject to evolution and, thus, encompass aspects of variability in space and time in order to evaluate the concepts of the thesis. Chapter 9 closes on the thesis by summarizing the main contributions, discussing design decisions within the concepts and giving an outlook on areas for possible future application and integration.

Part I.

Context and Preliminaries

1. The Configurable TurtleBot Driver as Running Example

The software technology group at TU Dresden has developed a driver software for the TurtleBot domestic robot. Due to the different hardware configurations of the TurtleBot and the limited resources on the robot (e.g., CPU and battery life), the driver was designed to be configurable in order to derive custom-tailored applications to be deployed to individual robots. The driver has been applied by project groups that each made custom configurable extensions to the driver with individual release cycles. Thus, the characteristics of the driver align with the general challenges addressed in this thesis so that the driver software is used as a running example for many of the concepts described in this thesis. The following sections elaborate on the TurtleBot robot as well as the driver software and the process of its development.

1.1. TurtleBot: A Domestic Service Robot

Currently, small domestic robots aiding with routine tasks in the personal household are at the verge of becoming an end-user friendly mass product. The TurtleBot[1] depicted in Figure 1.1 is one of these domestic robots used mainly to collect and deliver small items. Thus, the TurtleBot may, e.g., assist people with impaired mobility by delivering items such as syringes or pills.

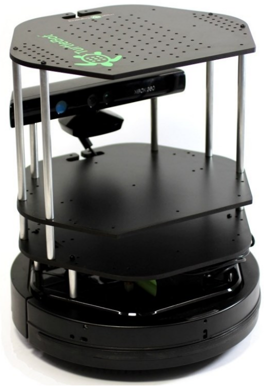


Figure 1.1.: The TurtleBot domestic service robot in revision 2.0.

In its current revision 2.0, the TurtleBot can be purchased pre-assembled or as individual components. In essence, the robot consists of a wheel drive engine as actuator enabling movement of the robot and a camera that may be used for position and person recognition. In addition, the robot features multiple sensors recording the internal state of the robot, e.g., wheel turn speed or remaining battery life, as well as data on the surroundings of the robot, e.g., using a bump sensor that triggers on collision with an object.

[1] http://turtlebot.com

Actual processing of sensor data and issuing commands to actuators is performed by a laptop assembled on the robot. The laptop may further be used to establish a WiFi connection to other computers or mobile devices that handle part of the control of the robot. To not restrain the free movement of the robot, both the engine as well as the laptop are battery powered during operation and need to be recharged. In practice, esp. the limited battery capacity of the laptop hinders constant use. Furthermore, the CPU power is a limiting factor as it is insufficient for more complex tasks, such as real time visual obstacle recognition, as well as combinations of basic tasks.

Apart from the common general setup, there are multiple different configurations for the TurtleBot's hardware setup. For example, there are different configuration options for the camera of the robot to use either a Microsoft Kinect[2] or an ASUS Xtion[3]. Furthermore, there have been two revisions of the overall TurtleBot system using different types of base platforms containing the wheel drive and the sensors of the robot. The first revision uses a retrofitted vacuum cleaning robot of the make iRobotCreate[4], whereas the second revision uses a wheel drive of the make iClebo Kobuki[5] dedicated to robot development.

Finally, there may be custom extensions to individual TurtleBots not available to all users of the robot. For example, the software technology group at TU Dresden extended individual TurtleBots to have additional means of detecting obstacles using a combination of infrared and ultrasound sensors. Both these types of sensors improve over the bump sensor as they do not depend on an impact occurring to register an obstacle. Furthermore, the ultrasound sensor may detect obstacles at a finer resolution than the infrared sensor. However, detection by infrared may include obstacles to the rear of the robot, which is relevant when reversing, as an additional sensor was installed pointing backwards. Hence, each of the obstacle detection mechanisms individually as well as combinations thereof may be feasible depending on the intended use case.

To operate the robot using end-user applications, a driver for the robot is required. The software technology group at TU Dresden developed a driver software for the TurtleBot that provides high level access to the robot's functionality (see Section 1.2). As the driver software has to cope with different configurations in a multitude of different artifacts, such as source code, project setup or documentation material (see Section 1.3) and their further development, it is fitting as running example to illustrate many concepts of this thesis.

As the software driver controls mechanical functions of the TurtleBot when it interacts with the environment, the combination of the TurtleBot's hardware and the driver software may further be considered safety critical. This yields many requirements for production use of the driver software regarding reliability and potential liability in case of accidents. However, these issues are out of scope of this thesis. Yet, the safety-critical nature of the driver yields a need for additional realization artifacts when considering safety certification that may need to be configured and developed further together with the rest of the realization artifacts. The following section explains the driver software and its configurable constituents in detail.

1.2. Configurable Driver Functionality

Due to different hardware configurations and the limited resources on the robot, it seems plausible to not provide the driver as monolithic software, but instead supply a custom-tailored variant for particular a hardware configuration and the intended usage scenario. For this

[2] http://microsoft.com/en-us/kinectforwindows

[3] http://asus.com/Multimedia/Xtion_PRO

[4] http://irobot.com/create

[5] http://kobuki.yujinrobot.com

purpose, the driver was split up into various functional components that, in part, may by selected or deselected depending on individual needs.

At its most recent state of development, the driver software of the TurtleBot encompasses the constituents depicted in Figure 1.2. The engine serves the purpose of locomotion and its respective software counter-part contains the protocol to issue movement commands regarding direction and velocity from a high-level programming interface.

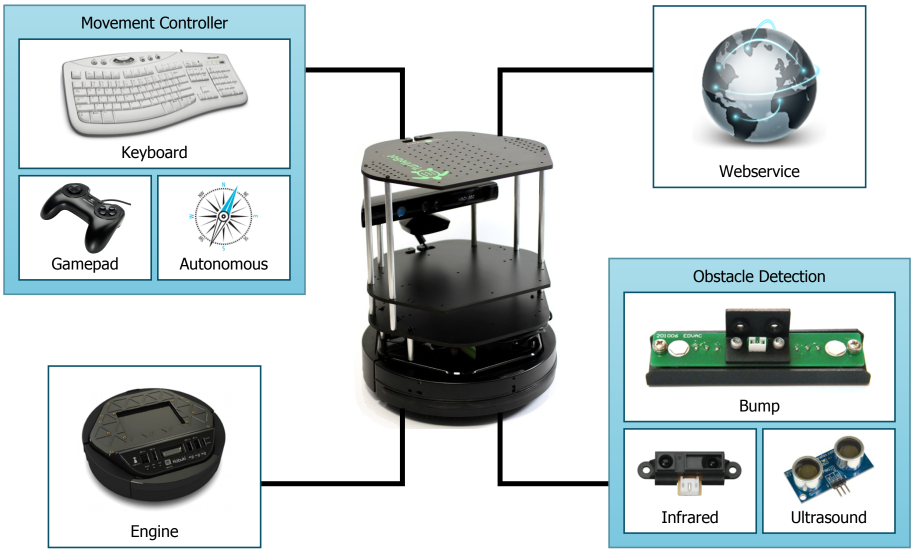


Figure 1.2.: The functional components of the TurtleBot driver software.

Besides this technical handling of movement, the group of movement controllers permits issuing logical commands regarding path or target of movement. The autonomous movement controller allows for marking a target on a virtual map representing the inner model of the surroundings of the TurtleBot, where the robot attempts to automatically plan a route and drive to the designated goal. Both keyboard and gamepad movement controllers permit remote control over the robot's movement by issuing direct commands for the rotation and acceleration to the engine. To avoid conflicts of commands from various sources, only one of the movement controllers may be used at any given point in time.

The webservice is an optional software component of the driver permitting transparent use of driver functionality over network. Hence, commands can be issued without wired connection when using WiFi. However, this may entail reduced reaction times, e.g., due to latency of the employed network.

The obstacle detection mechanisms serve the purpose of recognizing objects blocking the path of the robot when moving. The built-in bump sensor is available in all configurations and triggers when colliding with an obstacle. Furthermore, infrared sensors were installed on one of the robots pointing to the front and the rear of the TurtleBot to allow detection of obstacles in a distance of approximately half a meter. On another robot, an ultrasound sensor was installed to allow detection of obstacles at approximately the same distance but with a wider angle, so that obstacles to the side of the TurtleBot may also be detected[6]. Logically, a combination of these obstacle detection mechanisms is feasible depending on the installed hardware. In case of using multiple sensors with the driver software, triggering of a single sensor is treated as a recognized obstacle.

[6] Optical obstacle recognition by using the camera of the robot was not included in the driver due to the complexity of the calculation resulting in heavy stress on both the CPU and battery life of the robot in practical application.

When using the autonomous movement controller, the use of at least one concrete obstacle detection mechanism is mandatory in order to avoid accidents harming the robot or its surroundings. Likewise, using a remote movement controller by either keyboard or gamepad requires a wireless connection to the robot, which depends on the webservice being enabled.

However, not all of the components are essential for the robot's functioning, such as the webservice when using the autonomous movement controller. Furthermore, some components may be used as alternative to one another, such as the different obstacle detection mechanisms through different sensors. Hence, even for a given hardware setup, a number of configurations of the TurtleBot driver are possible to tailor its functionality to the respective needs. Figure 1.3 shows three examples of valid configurations. Through these configuration options, it is, in fact, not a single driver software that was developed but instead a *family of driver software* (see Section 2) with multiple members in the form of closely related, yet distinctly different software applications.

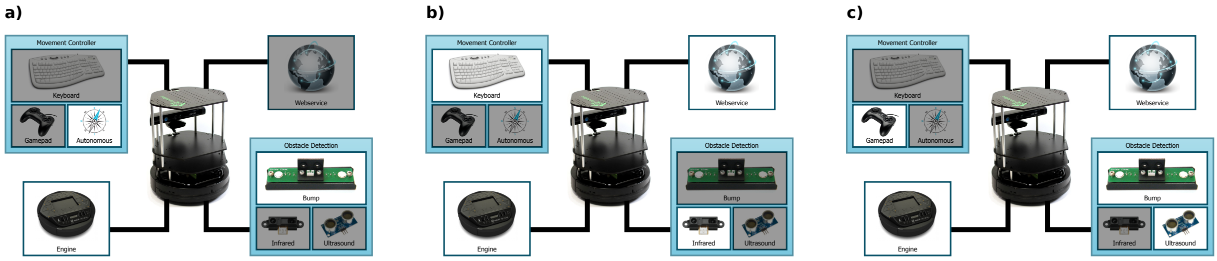


Figure 1.3.: Example of three different valid configurations of the TurtleBot driver software.

Besides the central functional configurable components, needs of various projects and individual applications have led to development of additional components not part of the centrally managed components. For example, an extension was created by non-core developers to playback a pre-recorded audio message when the bump sensor triggers. This functionality has been used during official receptions where the TurtleBot was presented to ask people to move aside when they blocked the way of the robot. However, this component does not take the configuration options of different obstacle detection mechanisms into account, but is hardwired to the bump sensor. Hence, the component implicitly requires presence of the bump sensor as an obstacle detection mechanism in the configuration and, thus, limits configuration options when employed. The component is not part of the TurtleBot driver, as it has not reached an adequate level of maturity.

1.3. Software Realization Artifacts

The realization of the TurtleBot driver consists of various artifacts in different notations. Elements of these artifacts as well as the their interconnections are subject to change with the different configurations of functionality. Furthermore, changes over time may also be the reason for altering the respective artifacts. Nevertheless, different configurations and versions of the artifacts need to be accessible to allow tailoring of the TurtleBot driver. In the following, an overview of the relevant notations is presented as basis for future elaborations.

Java source code is used to implement the logic of the TurtleBot driver in an imperative object-oriented programming language. Central functionality of the various functional components is captured by equivalently named and further auxiliary classes. Listing 1.1 shows an excerpt of a Java class realizing part of the functionality for the gamepad movement controller as example.

```java
package eu.vicci.turtlebot;

public class Gamepad extends Movement {
  public void accelerate() {
    //...
  }

  //...
}
```

Listing 1.1: Example of a Java class for the gamepad movement controller of the TurtleBot driver.

Eclipse projects are used to structure the aforementioned Java source code and realization artifacts in other notations during development with the Eclipse[7] IDE. Various different projects are used to group artifacts by the level of cohesion and references between the projects allow usage of artifacts across project boundaries. Furthermore, usage of external libraries, e.g., in Java Archive (JAR) files, may be configured with the project setup. Listing 1.2 and Listing 1.3 show examples of `.project` and `.classpath` Extensible Markup Language (XML) files resulting from setting up an Eclipse project in the Eclipse IDE.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>TurtleBot</name>
  <comment></comment>
  <projects>
  </projects>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
      <arguments>
      </arguments>
    </buildCommand>
    <!-- ... -->
  </buildSpec>
  <natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
    <nature>org.deltaecore.feature.constraint.resource.constraint_text.nature</nature>
    <nature>org.deltaecore.feature.constraint.resource.constraints.nature</nature>
    <nature>org.deltaecore.core.decore.resource.decore.nature</nature>
    <nature>org.deltaecore.feature.mapping.resource.mapping.nature</nature>
  </natures>
</projectDescription>
```

Listing 1.2: Example of the `.project` XML file of an Eclipse project setup.

DocBook[8] markup is used to represent documentation material in an XML dialect. Manuals for developers and users of the driver software are provided in this format, which may later be exported to PDF files. Depending on the functionality present in the respective driver, the documentation material has to be altered analogously. Listing 1.4 shows an example of DocBook as used within the TurtleBot driver software.

Besides these types of artifacts, there are further elements constituting the realization of the driver software. Due to the safety-critical nature of the robot's operation, a safety certification is beneficial before operation in a domestic environment. This procedure provides guarantees for

[7] `http://eclipse.org`

[8] `http://docbook.org`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path="src"/>
  <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER/
    org.eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.7"/>
  <classpathentry kind="lib" path="lib/lwjgl2.8.5/jar/jinput.jar"/>
  <classpathentry kind="lib" path="lib/lwjgl2.8.5/jar/lwjgl_util.jar"/>
  <classpathentry kind="lib" path="lib/lwjgl2.8.5/jar/lwjgl.jar"/>
  <classpathentry kind="output" path="bin"/>
</classpath>
```

Listing 1.3: Example of the `.classpath` XML file of an Eclipse project setup.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book xml:id="BookTurtleBotDriverManual" xmlns="http://docbook.org/ns/docbook" version="5.0">
  <title>TurtleBot Driver Manual</title>
  <chapter xml:id="ChapterEngine">
    <title>Engine</title>
    <para>The <emphasis>engine</emphasis>feature is responsible for ...</para>
    <!-- ... -->
  </chapter>
  <chapter xml:id="ChapterMovement">
    <title>Movement</title>
    <para>On a logical level, the movement of the robot can be controlled ...</para>
    <!-- ... -->
  </chapter>
</book>
```

Listing 1.4: Example of DocBook markup used as basis for documentation material of the driver software.

certain behavior and excludes liability for unsupported behavior of the robot. Safety certification of devices is performed by an impartial certification agency [Her00], which usually means an institution external to the company developing a particular device or software for it.

For the certification agency to assess whether a software system is considered sufficiently safe with regard to individual scenarios in a certain environment, a number of documents listing properties and behavior of the system are required. Among the most widely used notations for these documents are Software Fault Trees (SFTs) [Lev95, Her00], Component Fault Diagrams (CFDs) [KLM03], Checklists (CLs) [O'C04, Lev95, Rus92] and the Goal Structuring Notation (GSN) [KW04]. The TurtleBot driver software contains artifacts of all these notations.

Software Fault Trees (SFTs) [Lev95, Her00] are applied in safety-critical software to successively decompose a root fault into logical combinations of its constituent faults in order to determine causes for the root fault's appearance. An an SFT is a tree consisting of *gates* representing logical and/or operations as well as *intermediate faults*, which are refined further, and *basic faults*, which are considered atomic. Basic faults are assigned an individual probability of occurrence, which can be used to derive metrics for the likelihood of more complex faults activating. The procedure of creating an SFT is strictly top-down [Lev95, Her00, SSA13] starting with the root fault and gradually decomposing it into its (conceptual) causes. Hence, the structure of the SFT is aligned solely with the conceptual error propagation but does not take the realization of a system in individual implementation units and the respective error propagation therein into account. Figure 1.4 shows an example SFT where a root fault of a service robot, such as the TurtleBot, causing a collision is successively decomposed into its causal constituents.

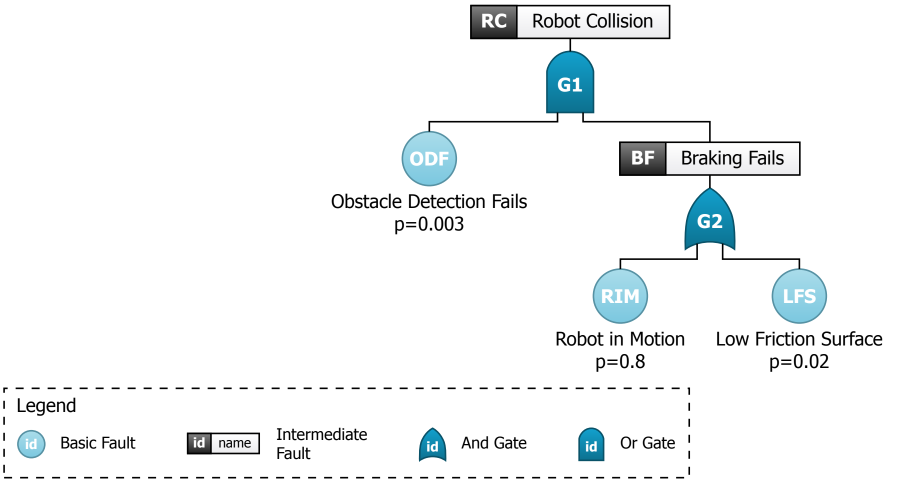


Figure 1.4.: Example of an SFT describing potential combinations of causes for a robot collision.

Component Fault Diagrams (CFDs) [SSA13, KLM03] are an extension of SFTs that can further model the structuring of a system into implementation units and the respective error propagation. For this purpose, intermediate and basic faults may be encapsulated into *components* that have in-ports and out-ports for error propagation. Components may be used and reused within CFDs as black boxes without knowledge of the respective internal structure. To create CFDs a mixture of top-down and bottom-up approach may be used starting with the conceptual error propagation paths similar to SFTs, but placing existing components with their error propagation ports at suitable places. Within this thesis, the name Component Fault Diagram is used instead of the original term “Component Fault Tree”, as the structure of the respective artifacts generally is not a tree but a graph. Figure 1.5 shows an example of a CFD with fault propagation paths similar to those of the SFT presented in Figure 1.4. However, both the `Obstacle Detector` (`OD`) and the `Braking System (BS)` are modeled as reusable black-box components with internal fault propagation paths unknown to the designer of the CFD. Furthermore, the contents of Figure 1.5 in their entirety are wrapped into a reusable component as well.

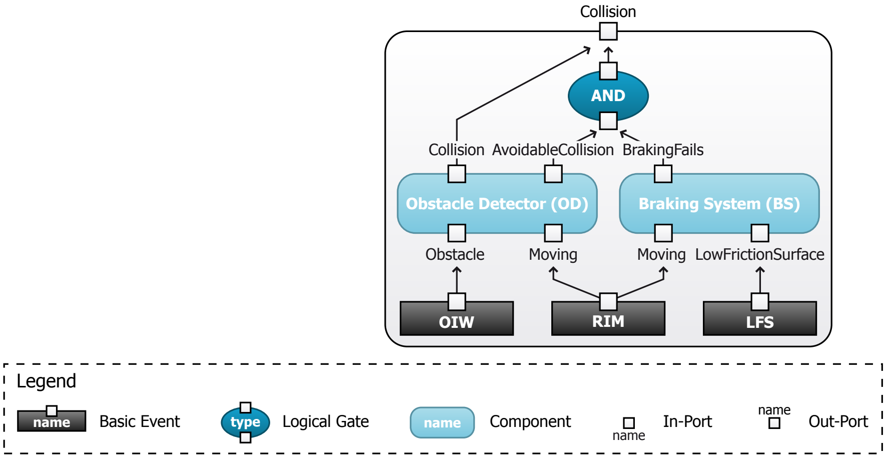


Figure 1.5.: Example of a CFD describing potential combinations of causes for a robot collision as a reusable component.

Checklists (CLs) [O'C04, Lev95, Rus92] contain a sequence of items that have to be considered when performing a certain procedure. Individual steps are enumerated as items on the list that have to be checked once completed. Within safety-critical systems, CLs are used for various processes, such as structured enumeration of all possible faults for a system, to ensure ordered and reproducible design processes, guidance of quality assurance or to check the completeness of certification material [O'C04, Lev95, Rus92]. Listing 1.5 shows an example of a CL describing different surface types and varying speed levels that have to be tested for the braking system.

```
checklist "Test Braking System"

group "Surface Type"
  F1 "Wooden Floor"
x F2 "Carpet"
x F3 "Concrete"
  F4 "Wet Floor"

group "Speed Level"
x S1 "Low Speed"
  S2 "Regular Speed"
  S3 "High Speed"
```

Listing 1.5: Example of a CL representing different combinations of conditions to inspect during test of a robot's braking system.

The **Goal Structuring Notation (GSN)** [KW04] is a semi-formal notation for arguing a safety case, i.e., which hazards exist for a system, how they are avoided or mitigated and which (other) certification material documents these measures. Hence, the GSN references documents in notations such as SFTs, CFDs or CLs. Artifacts of the GSN are graphical diagrams with different geometrical shapes for elements representing *Context*, *Goal*, *Solution*, *Strategy*, *Assumption* or *Justification* within a safety case. Furthermore, there are different connection types for *SolvedBy* and *InContextOf* relations. Artifacts of the GSN alleviate the procedure of gathering and putting into context all relevant certification documents for a safety case [KW04]. Figure 1.6 depicts an example of the GSN as semi-formal line of argumentation on how correct operation of a robot's collision avoidance mechanism is ensured.

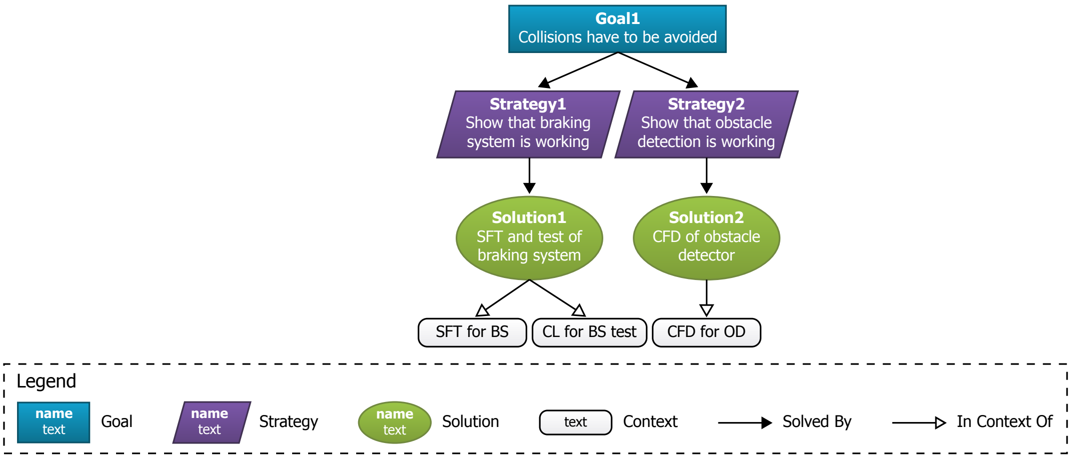


Figure 1.6.: Example of the GSN representing a semi-formal line of argumentation on the safety of a robot's collision avoidance mechanism.

1.4. Development History of the Driver Software

Section 1.2 described the most recent state of development of the TurtleBot driver and the functional components available at that time. However, this state is the result of development efforts of approximately 1.5 years, which added multiple new functional components and adapted existing ones to changed requirements.

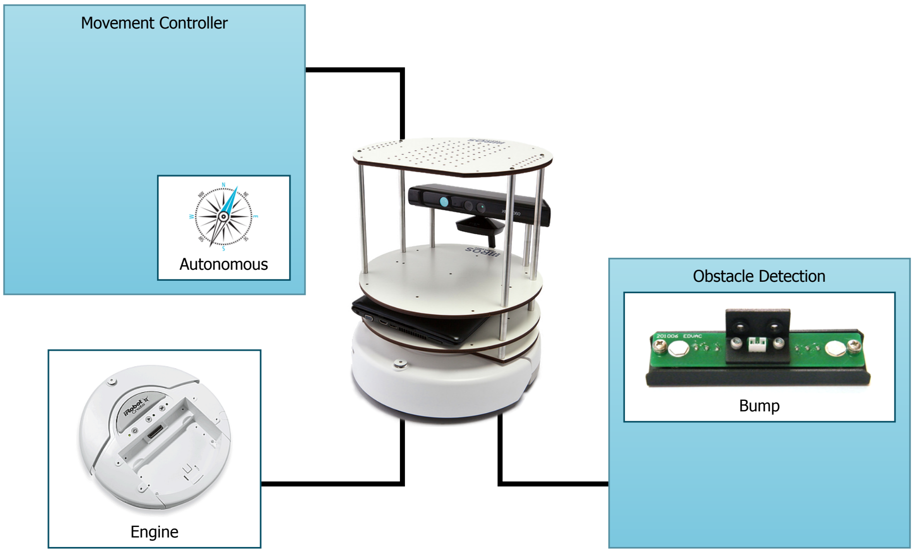


Figure 1.7.: The functional components of the TurtleBot driver software at the initial stage of development.

In the initial revision depicted in Figure 1.7, the TurtleBot driver had less functionality and was aiming at substantially different hardware: Before the current hardware revision 2.0 of the TurtleBot depicted in Figure 1.1, there was a previous revision 1.0 as depicted in Figure 1.7. The most significant difference is that the old revision employs a different engine from the current revision, which requires a different protocol to operate it. Furthermore, the custom hardware modifications of the robot employing the infrared or the ultrasound sensors had not been performed at that time. Finally, the initial revision of the driver focused autonomous operation of the robot and did not provide options for remote control so that neither the keyboard or gamepad movement controller nor the webservice used to operate them were available at that time.

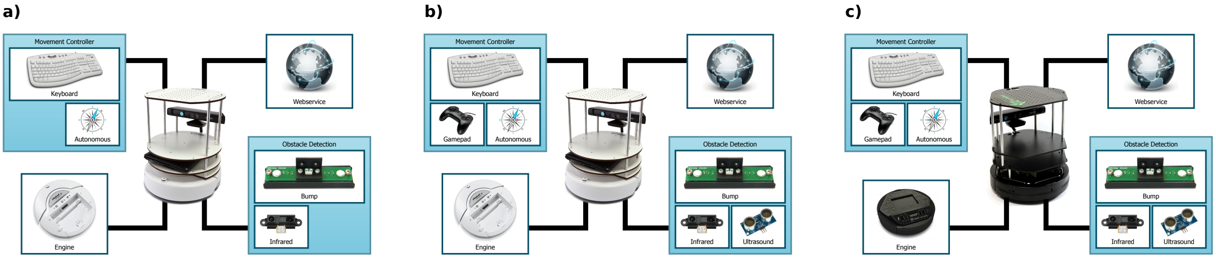
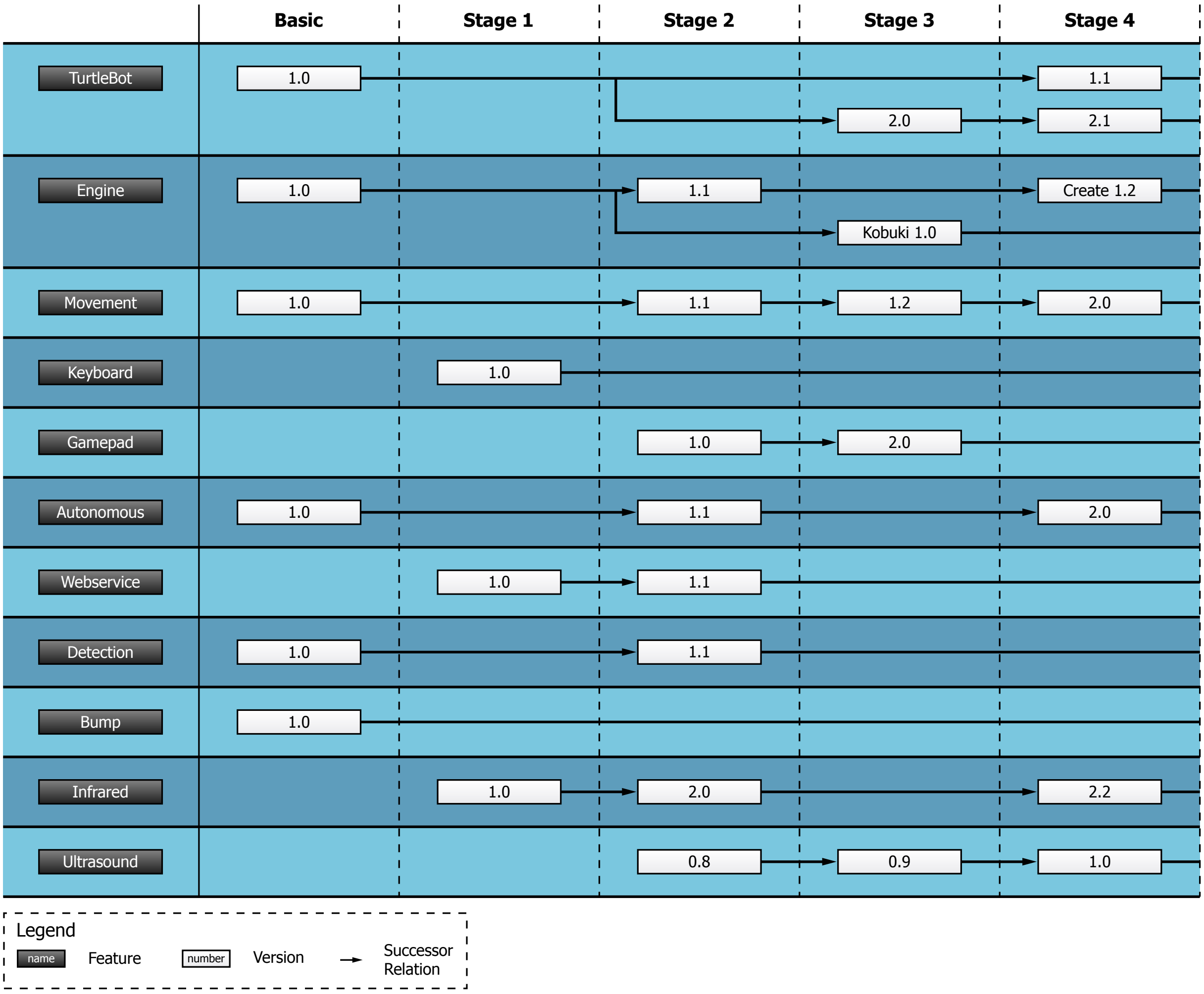


Figure 1.8.: Change of the functional components of the TurtleBot driver software over the course of approximately 1.5 years of development: a) after first development stage, b) after second development stage and c) after third development stage.

Over the course of approximately 1.5 years, the TurtleBot driver was developed further to the stage depicted in Figure 1.2. The changes performed to the driver can roughly be grouped into evolution stages capturing recurring intensified efforts in development. However, these stages were not explicitly planned or centrally coordinated so that, even within a single stage, the changes may have been performed at different times and by different independent developers. The grouping into stages was performed after the fact to more easily elaborate on the changes performed. Figure 1.8 depicts addition of new functional components for the first three stages of development.



	Basic	Stage 1	Stage 2	Stage 3	Stage 4
TurtleBot	1.0			2.0	1.1 2.1
Engine	1.0		1.1	Kobuki 1.0	Create 1.2
Movement	1.0		1.1	1.2	2.0
Keyboard		1.0			
Gamepad			1.0	2.0	
Autonomous	1.0		1.1		2.0
Webservice		1.0	1.1		
Detection	1.0		1.1		
Bump	1.0				
Infrared		1.0	2.0		2.2
Ultrasound			0.8	0.9	1.0

Table 1.1.: The development history of the individual features of the TurtleBot driver software over the 4 development stages.

In particular, the following addition of functional components was performed: In the first stage, remote movement control using a keyboard was introduced along with the webservice to allow remote connection to the robot. Furthermore, an infrared sensor was installed on one of the robots and a respective obstacle detection mechanism utilizing this sensor was added to the driver. In the second stage, a further movement controller was added to the TurtleBot that allows using a gamepad to remotely issue movement commands. Furthermore, an ultrasound sensor was installed on a further robot and the respective obstacle detection mechanism was realized as functional component. In the third stage, the TurtleBot's hardware revision 2.0 was introduced so that the core functionality and the functional component for the engine had to be adapted to accommodate for the changes in the protocols used to operate the different hardware.

In addition to adding entirely new functional components as configuration options of the TurtleBot driver software, there also were changes to existing components during the development. Reasons for these changes were fixing of defects as well as slight modifications of existing functionality, e.g., improvements in the path finding procedure utilized by the autonomous movement controller. These changes were performed as part of *updates* of existing functionality and yielded new versions of the respective components. Table 1.1 lists the resulting version history of the functional components over the course of the aforementioned development stages[9]. For example, the functional component for the engine was part of the original TurtleBot driver and was developed further in the second stage. With the introduction of the TurtleBot revision 2.0, starting in stage 3, there were two different development lines implementing the protocol for the new Kobuki engine as well as maintaining that for the old Create engine. All these versions as well as the different development lines need to be maintained as configuration options, as the different robots utilizing the driver software are not updated simultaneously or completely. Furthermore, the choice of particular versions also influences configuration options as, e.g., the versions of the engine component targeting the new engine cannot be combined with the core functionality of the TurtleBot component for the robot's old revision.

[9] In the fourth development stage, only updates of existing functional components were performed, but no new functionality was added. Hence, this stage was not included in Figure 1.8.

2. Families of Variable Software Systems

Commodity software assumes that a single software application can satisfy the needs of a wide range of potential users so that no individual developments are required, which keeps development efforts at a minimum. In contrast, software development for individual customers addresses individual concerns by developing an application for a specific use case. While both of these methodologies are valid in their respective domains due to the mentioned benefits, they also entail significant drawbacks: Commodity software cannot take requirements and demands of smaller groups of stakeholders into account. Individual software development is associated with a high development effort and, thus, high monetary cost. Hence, a solution that capitalizes on the benefits of both these approaches but reduces their drawbacks seems beneficial.

Software mass customization [PBvdL05] aims at combining benefits of both individual software development and commodity software in that it provides possibilities for configuration of a specific product but keeps development cost low by capitalizing on the similarities of the respective products. To customize a software system to a wide range of customer needs, it may be configurable in certain parts of its functionality. When the configurable options affect a wide range of a system's functionality, it may no longer be possible to handle a single software system but, instead, a set of closely related software systems. This set of systems then consists of assets shared by all members, regarded as *commonalities*, and assets used only by a subset of all members, regarded as *variabilities*. This general property of software to appear in different shapes is referred to as *variability* (see Section 2.1). Pohl et al. [PBvdL05] define commonality as follows: "Commonality denotes features that are part of each application in exactly the same form". Regarding variability they state: "Variability [...] is modelled to enable the development of customised applications by reusing predefined, adjustable artefacts". Configuration rules govern which combinations of variabilities are deemed valid when used together with the commonalities.

If the set of software systems has a significant common core functionality and is further subject to a software engineering process handling variability, one may speak of a *software family* or a product line (in the general sense). Parnas [Par76] formulates the underlying assumption to justify software family engineering as "Software family engineering assumes that there exists more commonality than variability in a family of software systems". Parnas does not distinguish the terms "software family" and "program family". Consequentially, the definition of software families used within this thesis uses that of "program families" by Parnas:

Definition 1: Software Family

"Program families are defined [...] as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." [Par76]

Pohl et al. [PBvdL05] make the following observation regarding the term "software family": "Concerning the terminology, there is an almost synonymous use of the terms 'software product family' and 'software product line'. Whereas in Europe the term software product family is used more often, in North America the term software product line is used more frequently” [PBvdL05]. However, within this work, the term *software family* is used in a different way, as an umbrella term for variable software systems that contain more than a single product and where individual products may be derived by using a valid subset of the configurable elements. In particular, it is meant as a superclass of the concrete concepts of Software Product Lines and Software Ecosystems explained in the following sections.

2.1. Variability

Software systems appear in multiple different shapes regarding their functionality, targeted platform or state of development etc. This quality is referred to as *variability* and different dimensions can be distinguished.

2.1.1. Variability in Space and Time

One scheme of distinguishing different types of variability is using the dimensions of space and time. The term *variability in space* encloses all changes associated with altering the configuration space. Hence, changes associated with variability in space are those that alter functionality of software–commonly referred to as *configuration.* Within this work, the definition of Pohl et al. is used for variability in space:

Definition 2: Variability in Space (Configuration)

“Variability in space is the existence of an artefact in different shapes at the same time.” [PBvdL05]

Regardless of whether a software system is subject to variability in space, over the course of time, each software system has to undergo changes to meet new requirements or fix defects in order to stay both usable and useful with regard to an intended application scenario. This procedure is referred to as *software evolution* [Leh80]. The result of software evolution are new versions of the affected software artifacts making them appear in different shapes. Hence, software evolution is a further dimension of variability of software systems. To distinguish it from variability in space, the variability dimension relating to software evolution is referred to as *variability in time*. Within this work, the respective definition by Pohl et al. is used for variability in time:

Definition 3: Variability in Time (Evolution)

“Variability in time is the existence of different versions of an artefact that are valid at different times.” [PBvdL05]

Even though variability in space and variability in time may seem orthogonal at first glance, they may, in fact, be interdependent: For one, the change of artifacts through evolution may directly affect configuration options. In addition, the existence of certain configurations with their dependencies regarding configurable functionality may restrain or put demands on further evolution. Pohl et al. acknowledge this interconnection as well: “Development artefacts vary in time as well as in space” [PBvdL05]. Other authors also distinguish variability in space and time such as [EBLSP10, SPBL12, BFG$^+$02].

Both types of variability can be found in the running example of the TurtleBot driver described in Chapter 1: As the driver software’s variable artifacts are subject to evolution, the software family of the example encompasses aspects of both variability in space and time that have to be captured.

2.1.2. Internal and External Variability

In addition to the distinction of variability in space and time, it is also possible to use an orthogonal classification into *internal variability* and *external variability*. Internal variability denotes all variable options that are used exclusively by maintainers of a software family but are never visible to customers of individual products. Pohl et al. define it as follows:

Definition 4: Internal Variability

"Internal variability is the variability of domain artefacts that is hidden from customers." [PBvdL05]

In contrast, external variability denotes those variable options that are explicitly visible to customers and, thus, can be used for product configuration. Pohl et al. define it as follows:

Definition 5: External Variability

"External variability is the variability of domain artefacts that is visible to customers." [PBvdL05]

The distinction of internal and external variability is orthogonal to that of variability in space and in time so that one may form categories such as "external variability in time" etc. Within this work, primarily, the dimensions of variability in space and in time are used, but the distinction of internal and external variability is essential for the decision of when to make new versions of artifacts publicly available as will be discussed in Section 4.4.

2.2. Manifestations of Configuration Knowledge

To manage variability in software families, configuration rules govern which combinations of variable and common artifacts are valid. The entirety of all these rules constitutes the *configuration knowledge*. Constraints on which constellations of variable assets are valid may emerge from realization and conceptual concerns. On the realization level, technical incompatibilities or limitations of the underlying technological platform may prevent a certain combination of variable assets from being valid. On a conceptual level, economical and logical concerns may restrain configuration options, e.g., when a group of variable assets should always be sold in conjunction or when at most one of the variable assets may be selected as they represent similar functionality with different characteristics such as performance.

These different levels of variability are reflected in software families by an explicit distinction of two concern spaces: the *problem space* and the *solution space* [CE00]. The problem space contains conceptual elements specific to the domain, such as requirements or the conceptual configuration options captured in a *variability model*. The solution space contains realization artifacts for all possible software systems of the software family such as source code, design models or documentation and certification material. In addition, a *variability realization mechanism* is used to create an individual product of the software family by assembling the realization artifacts for a particular conceptual configuration. Figure 2.1 illustrates the problem and solution space of software families as well as their interconnection by a variability realization mechanism.

The elements of the problem space are most suitable for non-technical stakeholders, such as managers or end-customers, to get an overview of configuration options or perform configurations on a conceptual level without having to know about the realization. In con-

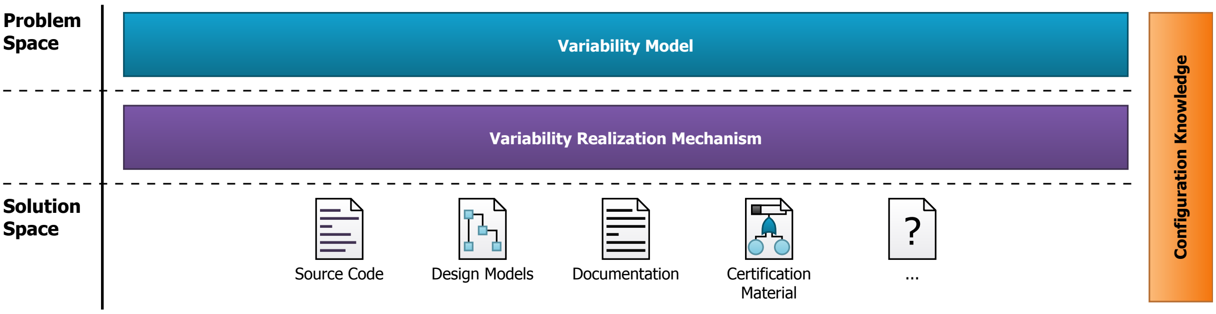


Figure 2.1.: The problem space of a software family contains domain knowledge and the solution space contains realization artifacts. A variability realization mechanism bridges the gap between both spaces. Configuration knowledge is present throughout all conceptual spaces.

trast, the elements of the solution space are most suitable for technical stakeholders concerned with the realization of the software system and are required in order to assemble executable software systems from the software family.

The configuration knowledge of the software family spans both these spaces. However, it is represented differently in the respective spaces. On a conceptual level, it is captured within a *variability model* in the problem space and, on a realization level, it manifests as customization of the realization assets in the solution space. To transform a configuration of the conceptual variability model to a concrete realization of a software system, a *variability realization mechanism* is employed within a *variant derivation process*. These constituents of software family engineering are described in detail in the following sections.

2.2.1. Variability Models

Variability models capture the conceptual configuration knowledge independently of a potential realization in software. They specify which combinations of variable assets are valid and which are invalid. The reasons for certain configurations being infeasible may be technical in nature but may also be, e.g., economical if two configuration options should not be sold in conjunction. The intent of variability models is to capture the entirety of these constraints in a compact notation without particular focus on technical details of a realization in software. Nevertheless, the configuration knowledge represented in variability models needs to be complete in the sense that no invalid configurations are permissible and all valid configurations can be derived with regard to the configuration rules of the variability model. Hence, technical incompatibilities originating from realization artifacts (solution space) have to also be represented in the variability model (problem space). This is necessary to not suggest validity of configurations that are, in fact, invalid as the respective software systems cannot be created.

A number of different types of variability models exist. Among the most prominent are *feature models* [KCH⁺90], *decision models* [MA02], *Orthogonal Variability Models (OVMs)* [PBvdL05] and Variability Specifications (VSpecs) of the *Common Variability Language (CVL)* [HMPO⁺08]. Each of these notations focuses slightly different concerns.

Feature models [KCH⁺90, CE00] capture variable assets as a hierarchy of *features* each encompassing a set of user-visible requirements. In addition, *cross-tree constraints* spanning across the tree, such as requires or excludes, may be expressed using propositional formulas over features. Hence, implicit and explicit constraints among features describe all possible configurations of the software family (see Section 3.2.1). For example, the configuration options of the TurtleBot driver software from Chapter 1 may be represented in a feature model as presented in Figure 2.2.

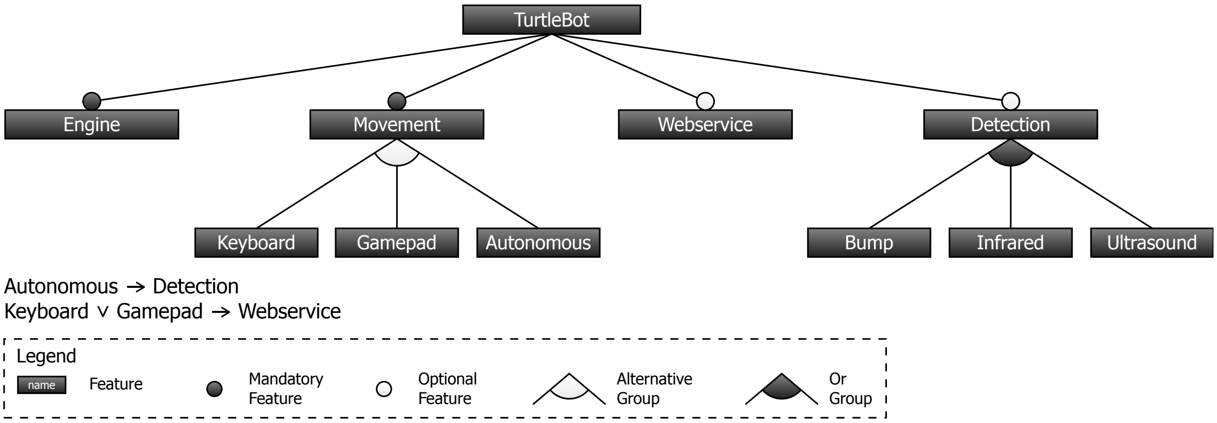


Figure 2.2.: Example of a feature model for the configurable TurtleBot driver software from Chapter 1 with two cross-tree constraints.

Decision models [MA02] capture configuration knowledge in a set of interconnected questions that have to be answered by deciding for one of a particular set of possible values. Hence, decision modeling has a very strong focus on the process of configuration. The expressive power of decision models is equivalent to those of feature models [ESDS12] so that similar configuration knowledge may be represented. For example, Table 2.1 represents the configuration options of the TurtleBot driver software from Chapter 1 represented as decision model.

Decision Name	Description	Type	Range	Cardinality	Rule

Movement	Which type of movement controller do you want to use?	Enum	Keyboard |
			Gamepad |
			Autonomous
				1 : 1	**IF** Movement.contains("Keyboard") **OR** Movement.contains("Gamepad") **THEN** Webservice.setValue(true);
					IF Movement.contains("Autonomous") **THEN** Detection.setValue(true) **AND** Detection_Mechanism.setVisible(true);

Webservice	Do you want to use a webservice for remote control?	Boolean	true | false		
Detection	Do you want to use an obstacle detection mechanism?	Boolean	true | false		
Detection_ Mechanism	Which obstacle detection mechanism(s) do you want to use?	Enum	Bump | Infrared | Ultrasound	1 : 3	

Table 2.1.: Example of a decision model for the configurable TurtleBot driver software from Chapter 1 represented as table.

Orthogonal Variability Models (OVMs) [PBvdL05] assume that there are different dimensions of (potentially orthogonal) concerns involved in variability. For example, concerns regarding the functionality of a software family may be orthogonal to those regarding the platform (e.g., Windows or Linux). In contrast to the aforementioned approaches to variability modeling, OVM uses *variation points* as top level elements that specify independent places for variation. A variation point lists all possible configuration options for that point. The sum of all variation points and their options comprises the variability model. Within OVM, each concrete configuration option is called a *variant*, which is in contrast to the meaning of the term employed within this thesis as a software product conforming to a valid configuration of the software family (see Definition 7). Figure 2.3 shows an example of an OVM of the TurtleBot driver software from Chapter 1. In contrast to a feature model, which captures both commonalities and variabilities, an OVM only contains variabilities.

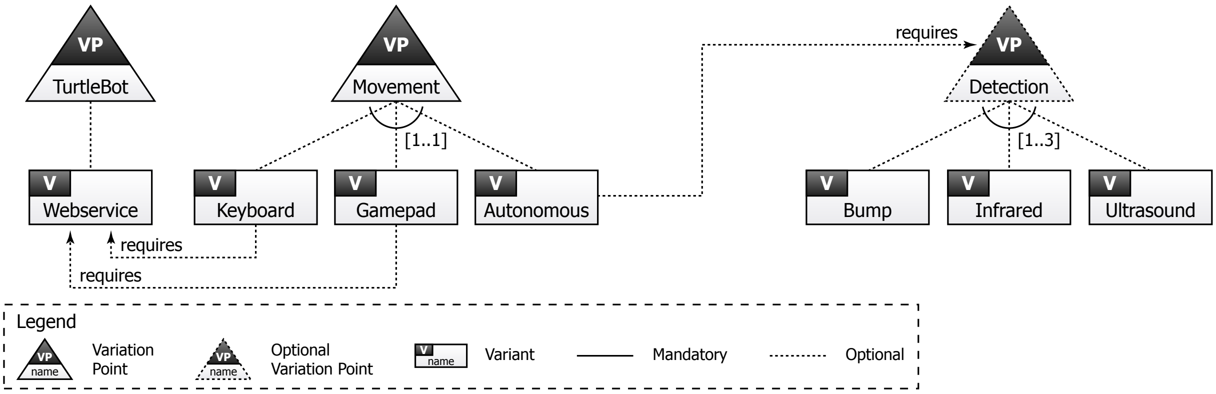


Figure 2.3.: Example of an OVM for the configurable TurtleBot driver software from Chapter 1.

The **Common Variability Language (CVL)** [HMPO+08] is an attempt at adding standardized variability to arbitrary modeling notations by overlaying variability information over realization assets of a software family. CVL uses Variability Specifications (VSpecs) to describe variability on an abstract level, where each VSpec describes possibilities to bind variability at one variation point, e.g., by choice from a fixed set of options or by setting the value of a variable. The sum of all VSpecs comprises a variability model that may be assembled along the structure of a tree–called a *VSpec tree*. Furthermore, CVL supports the use of propositional logic to specify further constraints. Figure 2.4 illustrates an example of a CVL VSpec tree with configuration knowledge of the TurtleBot driver software from Chapter 1.

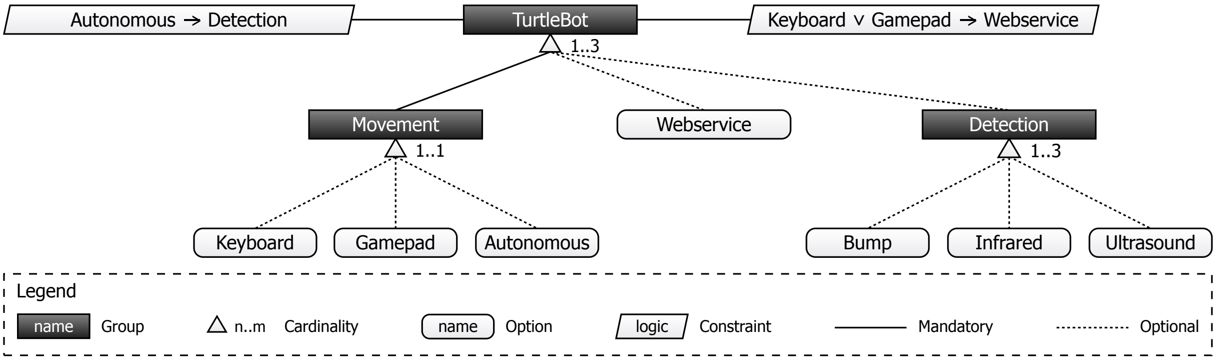


Figure 2.4.: Example of a CVL VSpec tree for the configurable TurtleBot driver software from Chapter 1.

Each of the presented variability models encompasses the configuration knowledge of a software family in a prescriptive way. In all these notations, variable assets are modeled explicitly, which is in contrast to other approaches to reuse-in-the-large, such as frameworks. Hence, when using the variability model as canonical source of configuration knowledge[1], this means that *all* the described configurations are valid but also that *no other* configurations are valid (closed world assumption).

With the compact and often visual representation of variability models, they are very suitable to get an overview of all possible configuration options. Furthermore, variability models may be employed in the variant derivation process to specify a concrete product of the software family on a conceptual level by means of a *configuration*. Within this thesis, a configuration is defined as described in Definition 6.

Definition 6: Configuration

A subset of all configurable assets on conceptual level in terms of entities from the variability model that is valid with regard to the configuration knowledge.

Depending on the concrete notation for the variability model, applying this definition yields different artifacts for a configuration: For feature models, a configuration is a subset of features (see Section 3.2.1); for decision models, a sequence of answers to the questions posed in the decision table; for OVMs, a selection of variation points and variants; and for CVL VSpecs, a concrete selection from the provided options and values for the specified variables (called a *resolution*)[2]. However, in either case, a configuration defines a product of the software family merely on the conceptual level.

2.2.2. Variability Realization Mechanisms

To assemble an executable software system from a conceptual configuration, a *variability realization mechanism* [SRC+12] has to be employed to manifest configuration knowledge on a realization level. The result of this procedure is one concrete member of the software family in the form of a *variant* as defined in Definition 7.

Definition 7: Variant

A variant of a software family denotes the realization of one member of the set of software systems encompassed by the family that is valid with regard to the configuration rules governing variability.

Hence, a variant is the realization-level counterpart of a conceptual configuration. Various publications use similar definitions of the term *variant* [AK09, HKM+13, PSC09, RSPA11, SRG11, HHK+13, SRS13]. Furthermore, the term *product* of a software family is used synonymously to the above definition of variant in these sources. However, there is no universally accepted definition for the term *variant*, so that, in some publications, it is defined as one concrete *option at a variation point* [PBvdL05, AKM+10, ZSS+10, GA01, GBS01, BB01].

To create a variant of a software family, a configuration of a variability model, such as a feature model, is provided as input to the variability realization mechanism. The variability realization

[1] This should be the case in a software family that employs variability models. However, studies on industrial practice [ZBP+13, HSB+14] have revealed that, in some cases, there are additional or even conflicting constraints on the configuration knowledge stemming from realization assets. As explained in Section 3.4.3, within this thesis, it is assumed that the variability model is the canonical artifact for representing the complete configuration knowledge.

[2] Within this thesis, the *process* of defining a configuration is perceived as a timeless, transactional procedure. Staged configuration [CHE04] allowing individual configuration operations in multiple steps and potentially by various stakeholders at different times is not considered.

mechanism has to be aware of how to retrieve all realization assets relevant for building a particular variant. It then assembles a variant from all implementation assets related to the elements of the configuration and their respective realization assets. Figure 2.5 illustrates this general procedure.

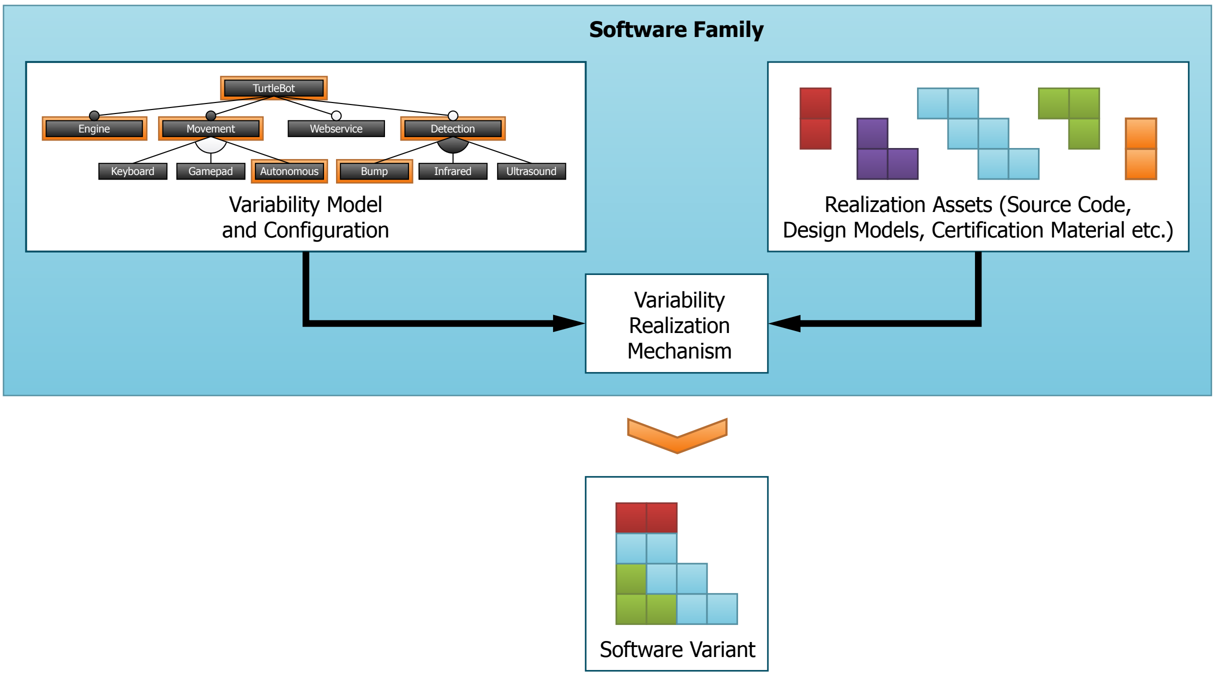


Figure 2.5.: A variability realization mechanism transforms a conceptual configuration into a variant of the software family.

The concrete procedure of building a variant from the realization assets and even the form in which the realization assets of the software family are available to the variability realization mechanism depends on the concrete type of mechanism [SRC$^+$12, Bat04, KAK08, SBB$^+$10]. Three principle types of variability realization mechanisms can be distinguished: *annotative*, *compositional* and *transformational*. The following sections elaborate on each of these types as well as their implications and requirements.

2.2.2.1. Annotative Variability Realization Mechanisms

Annotative variability realization mechanisms[3] [KAK08, SRC$^+$12] assemble all possible variations of a realization asset within a single artifact. Hence, the characteristics of all possible variants of the software family are contained within the artifact provided that it is affected by the respective configurable elements. An example of such an artifact would be a Java class with a method that contains application logic for all possible configurations within an if-cascade checking whether certain options are included in a configuration. As, in the general case, such an artifact contains more than the elements required by any one variant, it is often referred to as a *150% model*.

The connection between the conceptual variability model and the respective parts of the 150% model of the realization assets is established using *annotations*. Annotations may be either *internal* or *external* with regard to the realization artifact. Internal annotations are placed inside

[3] Annotative variability realization mechanisms are sometimes also referred to as *subtractive* or *negative* variability realization mechanisms.

the affected realization artifact, e.g., using constructs available in the language of the artifact or additional languages integrated into that of the realization artifact. For example, cascades of conditional control flow branching (e.g., `if...then` statements) may annotate sections of source code attributed to a particular feature or a combination of features. In the C/C++ programming languages, the use of conditional compilation with preprocessor directives (e.g., `#ifdef`) is a common practice for internal annotation [KAK08]. Likewise, special comments may serve a similar purpose, e.g., using the Antenna[4] preprocessor for Java. Naming conventions may be employed to relate elements from the variability model to those used in the internal annotations. Listing 2.1, Listing 2.2 and Listing 2.3 illustrate the use of if cascades and comments for the Antenna preprocessor in Java as well as conditional compilation with preprocessor directives for C/C++ to realize annotation, respectively.

```
//Configuration
boolean featureA = false;
boolean featureB = true;
boolean featureC = false;


//...

//Implementation
if (featureA) {
  //...
} else if (featureB) {
  //...
} else if (featureC) {
  //...
}
```

Listing 2.1: Example of an annotative variability realization mechanism with if cascades in Java.

```
//Configuration
//#define [FEATURE_B true]

//...

//Implementation
//#ifdef FEATURE_A
  //...
//#elifdef FEATURE_B
  //...
//#elifdef FEATURE_C
  //...
//#endif
```

Listing 2.2: Example of an annotative variability realization mechanism with Antenna preprocessor directives for Java formulated as comments.

External annotation references elements from outside of the affected realization artifact. In the most basic case, elements of the variability model may be related to parts of individual realization artifacts they are associated with. For example, this may be performed through referencing the affected parts of the realization artifact by name, e.g., a field of a Java class that is associated with a certain configuration option may be referenced by an identifier consisting

[4] `http://antenna.sourceforge.net/wtkpreprocess.php`

```
//Configuration
#define FEATURE_B true

//...

//Implementation
#ifdef FEATURE_A
  //...
#else
#ifdef FEATURE_B
  //...
#else
#ifdef FEATURE_C
  //...
#endif
#endif
#endif
```

Listing 2.3: Example of an annotative variability realization mechanism with conditional compilation using preprocessor directives in C/C++.

of `<QualifiedPackageName>.<ClassName>.<FieldName>`. More complex annotations may be formulated when associating a logical expression over configurable elements with that identifier. Another example of an external annotation mechanism is the explicit mapping model utilized by the tool FeatureMapper [HKW08] where logical expressions over features are related to arbitrary parts of realization assets as illustrated in Figure 2.6. The added benefit of an external annotation is that the realization assets need not be aware of the use within a software family. This leads to a cleaner separation of concerns (functionality vs. variability in space) at the cost of having additional artifacts containing the annotations.

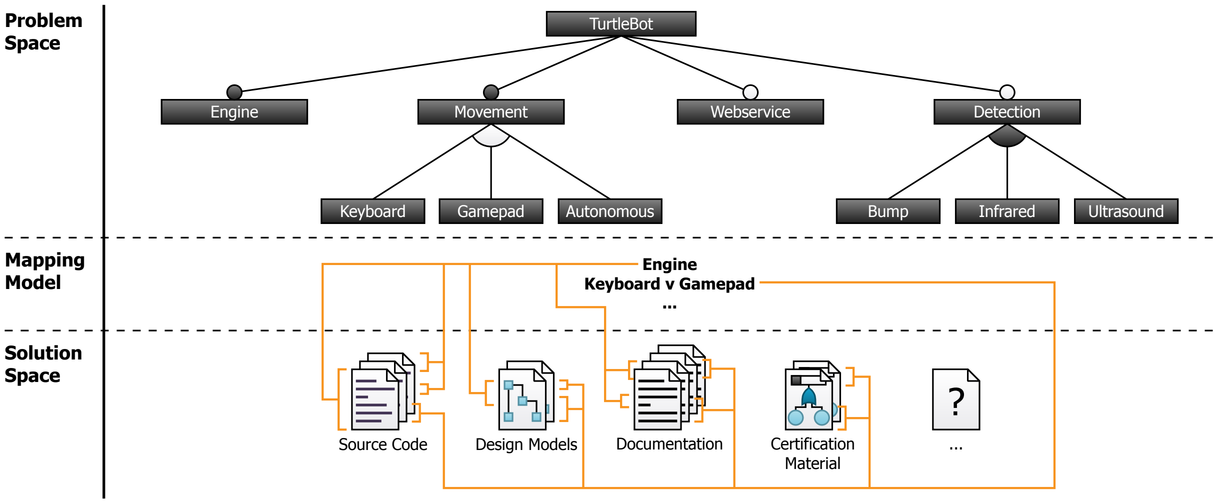


Figure 2.6.: Example of external annotation using a mapping model to relate logical expressions over features to individual parts of various realization assets.

To derive a concrete variant, conceptual configuration elements from the variability model included in the input configuration are resolved to the respective elements from the required realization assets. The respective variant of the software family is then assembled by reducing the 150% model of each required realization artifact: All those parts of the model that are guarded by annotations but are *not* a required part of the configuration for the realization asset are *removed*. Figure 2.7 illustrates the general procedure of variant derivation using an annotative variabiltiy realization mechanism.

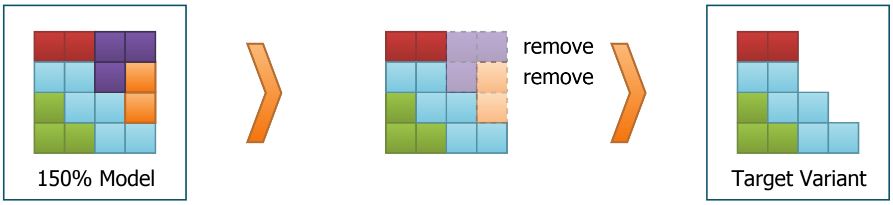


Figure 2.7.: Annotative variability realization mechanisms assemble all possible variations of a realization asset in a "150%" model for the software family and remove the parts that are not required by a particular configuration.

Many tools for the handling of software families utilize an annotative variability realization mechanism: BigLever's Gears[5] [Kru08] and pure-system's pure::variants[6] [Beu12] are industrial tools for software families. FeatureIDE[7] [TKB+14], Clafer[8] [BCW11] and FeatureMapper[9] [HKW08] are tools for software families stemming from academia.

An annotative variability realization mechanism poses no specific requirements on the structure of realization assets and allows the specification of variations for a realization asset in the same language as the asset itself. However, depending on the annotation mechanism employed, it may be problematic to express some variations (e.g., for alternative configuration options) while still maintaining validity with regard to syntax and well-formedness of the realization artifact. For example, when presence of one configuration option results in a Java field having a particular type and its absence in another type, this information cannot be captured in a Java class that is valid with regard to syntax and static semantics when, e.g., comments are used as annotation mechanism. Even when lifting the annotation to the level of the field declaration (as opposed to just its type), there would still be two fields with similar name within that class. This violates well-formedness rules of the language and will be rejected by most tools. Furthermore, the need to assemble all possible variations of a realization asset within a single 150% model requires full knowledge of all possible configuration options so that new options can only be realized by altering the 150% model itself.

2.2.2.2. Compositional Variability Realization Mechanisms

Compositional variability realization mechanisms[10] [Bat04, KAK08, SRC+12] assemble a variant from a conceptual configuration by *adding* parts of realization assets required by the configuration (the units of composition) to a common *core model*. The core model encompasses those artifacts that are common to all possible variants of the software family. Hence, it is not necessarily a complete representation of one variant (e.g., when considering alternative features in a mandatory group). Furthermore, the realization assets of the core model are not necessarily valid with regard

[5] `http://biglever.com/solution/product.html`

[6] `http://pure-systems.com/Products.html`

[7] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide`

[8] `http://clafer.org`

[9] `http://featuremapper.org`

[10] Compositional variability realization mechanisms are sometimes also referred to as *additive* or *positive* variability realization mechanisms.

to the syntax and/or semantics of their respective language as they may only be completed to valid assets by adding further parts during the variant derivation procedure.

In an example of utilizing a compositional variability realization mechanism for a Java program, the core model may consist of a class containing various methods whose implementation includes calls to a particular method that is not present in the core. As part of two units of composition that are associated with alternative elements in the variability model, the method may be realized with different implementation logic by specifying the declaration and implementation of the method. In this example, neither the core model nor the units of composition are, in isolation, valid with regard to syntax and/or static semantics of Java.

With compositional variability realization mechanisms, a special case may exist where the core model itself is empty so that variants are built by merely combining the units of composition. However, this case should be rare in practical application as the underlying assumption of a software family is that the individual members *share* significant amounts of functionality (see Definition 1) that could be captured in a core model.

To derive a concrete variant, compositional variability realization mechanisms utilize the core model, resolve conceptual elements from the configuration to the respective units of composition and compose them with the core model. The relation between elements from the variability model and the respective units of composition may be established in various ways, e.g., by using a naming convention or an explicit mapping model. Figure 2.8 illustrates the general procedure of deriving a variant with a compositional variability realization mechanism.

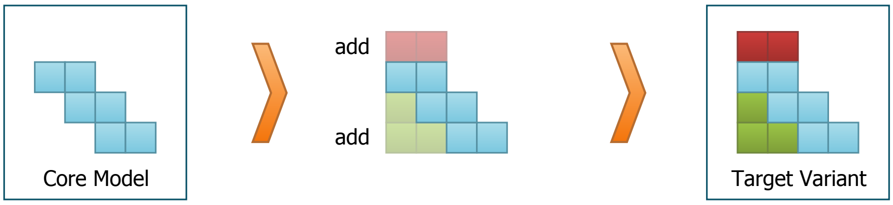


Figure 2.8.: Compositional variability realization mechanisms capture variations related to individual configuration options in separate units of composition external to the realization assets and combine them with the core model when deriving a variant.

Compositional variability realization mechanisms are employed by a variety of approaches and tools: **Feature Oriented Programming (FOP)** [Bat04] captures changes to realization artifacts associated with a single feature (see Definition 10) in individual *feature modules*. Hence, there is a one-to-one relation between features and feature modules. Furthermore, feature modules may specify interdependencies and constraints on their application order. When deriving a variant, the feature modules associated with the features in the configuration are sorted according to their application-order constraints and the respective contents are then composed with the core model of the software family. FOP is implemented in various tools, such as the Ahead Tool Suite[11] [Bat04] or FeatureHouse[12] [AK09]. Furthermore, FeatureIDE [TKB$^+$14] may be configured to use a compositional variability realization mechanism as well.

Furthermore, **Aspect-Oriented Programming (AOP)** [KLM$^+$97] may be employed as a compositional variability realization mechanism for software families. An *aspect* captures (potentially cross-cutting) concerns of a software system as additions to various significant locations of the targeted realization artifact (called *join points*). The additions of an aspect are

[11] http://cs.utexas.edu/~schwartz/ATS/fopdocs

[12] http://infosun.fim.uni-passau.de/spl/apel/fh

composed with the targeted realization artifacts by *weaving* them into the respective locations. When using individual aspects to represent the changes associated with a conceptual configurable unit from the variability model, this process constitutes a compositional variability realization mechanism as is utilized in various different approaches [AJC$^+$07, GV09, FCS$^+$08].

When compared to annotative variability realization mechanisms, compositional variability realization mechanisms have the benefit that not necessarily all possible variants of a realization asset have to be known in advance as new ones may be added as additional units of composition assuming that they can be realized by more compositions (i.e., not by removal). However, a compositional variability realization mechanism may pose requirements on the structuring of the realization assets: If validity of realization assets with regard to syntax and static semantics should be maintained even for the units of composition, the architecture of the realization assets has to be chosen accordingly. For example, for source code, this may entail a component-based software architecture [HC01, TMD09] that allows addition of further components in the process of variant derivation. Alternatively, composition is also possible without obeying a particular structure of the realization assets but might entail that the units of composition or even the common core model are, in isolation, invalid with regard to syntax and static semantics of their respective language. In addition, in comparison to annotative variability realization mechanisms, which assemble all possible variations of a realization asset within a single 150% model, compositional variability realization mechanisms lead to an increase in scattering as each variation of a realization assets is represented in its individual unit of composition, which may increase maintenance effort.

2.2.2.3. Transformational Variability Realization Mechanisms

Transformational variability realization mechanisms create variants of a software family by applying transformation operations of different complexity to a software system. At an atomic level, these transformation operations may be described as *add*, *modify* and *remove* of individual elements of the respective underlying language. However, more complex operations may be synthesized by composing the basic operations. Individual operations may be grouped into *transformation modules* regarding the cohesion of the intended transformations. It is principally possible to include control flow logic into the transformation modules, e.g., by employing conditional branching and performing different transformations on each branch. However, in the general case, a transformation module encompasses a mere list of transformation operations that is to be applied sequentially if the transformation module is selected as part of the variant derivation procedure.

Each individual transformation module may realize as much as the functionality entailed by a single feature of the software family. However, transformation modules may also be more fine-grained so that they only represent a fraction of a feature. This has benefits, e.g., when different features, in part, require identical transformations to be performed whose specification should not be duplicated. In such a case, it is beneficial to specify the common transformations in a separate transformation module that is referenced by the transformation modules realizing the individual changes required by the individual features. Hence, for a valid variant of the software family, a subset or all of the specified transformation modules have to be applied. It is possible to relate (combinations of) the entities of a variability model to sets of transformation modules. For a concrete configuration, this relation may then be used to determine the transformation modules required to realize a variant.

When applying transformation modules to create a variant of a software family, the source of transformation is one valid variant of the software family, referred to as the *base variant*. In contrast, the resulting variant is referred to as the *target variant*. Figure 2.9 illustrates the general procedure to create a variant of a software family using a transformational variability realization mechanism. In a concrete example, part of a software variant may be reflected within a Java class. To create the Java class of this variant through transformation, fields rendered redundant in the variant may be removed, new methods may be added and statement lists may be modified to call the newly added methods.

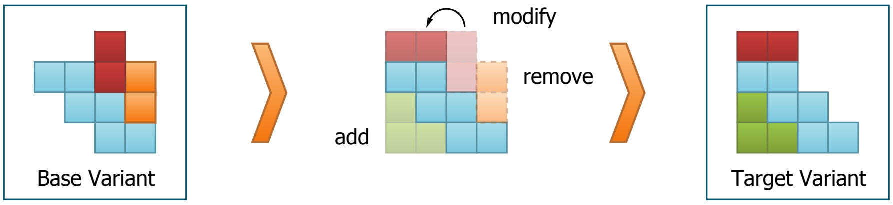


Figure 2.9.: Transformational variability realization mechanisms transform a base variant of the software family to a target software system by adding, modifying and removing parts.

Determining which variant is to be used as base variant is an essentially arbitrary choice. However, there are various aspects to consider: For one, chronology of development may lead to the variant being developed first (possibly as a standalone software system) serving as basis for further variants that are specified by transformation modules to be applied. Furthermore, a set of pre-existing closely related software systems may be consolidated to a software family by using the variant with the most significant overlap with the rest of the software systems as base variant and specifying the remaining variants using transformation modules. Finally, it is also possible to select the variant that is most suitable for transformation as the base variant. For instance, for Java, referencing classes, methods and fields by their respective qualified names is more stable than referencing individual statements in a method body by their line number so that a variant may be more suitable as base variant if it entails variability above statement level.

Different approaches may be used as transformational variability realization mechanisms. For one **delta modeling** [SBB$^+$10, SD10, CHS10] (see Section 3.2.2) uses target language specific *delta languages* that provide *delta operations* to transform a base variant by adding, modifying or removing elements of a base variant. *Delta modules* are used as transformation modules to group cohesive sequences of invoking delta operations. To create a variant, a subset of all delta modules is selected that is applied to transform a base variant to the intended target variant.

To employ **model transformation** as transformational variability realization mechanism, it is possible to perceive all realization assets as models that are instances of metamodels of their respective languages (see Section 3.1). Individual transformations may be specified by model transformation operations. Within *general purpose model transformation* [Sch06, SK03], these operations target entities of the notation used for metamodeling. Hence, the operations are generic and can be used for artifacts of all languages representable by metamodels. However, this lack of relation to the concrete language of the artifacts being modified may make transformations hard to specify and comprehend. As a remedy, *domain-specific model transformation* [RW11] creates transformation operations that utilize the constructs of the languages for modified artifacts. Individual model transformation scripts may be perceived as transformation modules of a variability realization mechanism that have to be applied in order to create a variant of a software family.

Furthermore, **Invasive Software Composition (ISC)** [Aßm03] may be perceived as a transformational variability realization mechanism when employed to create variants of a software family. ISC is a fragment-based gray-box composition technique that uses transformation to adapt and extend *components* at specific *hooks*. Components are represented by *fragment containers*, which may be classes, packages, software components etc. each containing a certain number of hooks. Hooks are the points of composition within ISC. A hook may be implicit (e.g., the entry/exit point of a method) or explicit in the form of a *declared hook* (realized through, e.g., new syntax elements/naming conventions/special comments). A *composer* transforms unbound hooks to bound hooks by performing a set of transformation operations. In ISC, the most essential transformation operation is additive composition to extend existing functionality. However, despite its name, ISC also allows modifications such as renamings and subtractive operations such as deleting individual elements which makes it a transformational approach. ISC may be used as a transformational variability realization mechanism if the sum of all relevant fragment containers forms a valid variant of the software family that may be regarded as the base variant. Hence, contained hooks have to be either implicit or specified with a language construct of the realization artifact's language. Furthermore, the represented functionality has to be that associated with the configuration of the base variant.

The transformation from one concrete variant to another variant is in contrast to both annotative and compositional variability realization mechanisms that each use a special family-aware representation of their realization assets. Annotative variability realization mechanisms form a 150% model. This may result in conflicting information, such as multiple super classes for a Java file, which is correct for the software family but not for a concrete artifact of the realization language. Furthermore, compositional approaches split functionality into relatively small composable units. In isolation, the composable units may also not constitute valid artifacts of the respective realization language. For example, FOP uses feature modules containing only fragments of assets in the target language. This problem hinders use of standard tools, which may not be able to deal with these conflicts. Furthermore, it may lead to problems regarding comprehension of the respective artifacts for developers.

In contrast, a transformational variability realization mechanism uses one concrete software system as source of transformation that constitutes a valid variant of the software family and transforms it into another variant conforming to a different configuration. This procedure has various benefits: For one, provided that the configuration knowledge is sound, both the input and the output of the variability realization mechanism consist of artifacts valid with regard to the syntax of their respective languages so that they can be inspected using standard tools. Furthermore, the general process of retrieving additional variants, by starting out with one concrete software system and transforming it, aligns well with the practice of companies of first developing individual software solutions on request and, later on, transforming them into configurable software families that are sold as customized products off the shelf. Finally, the general notion of using transformations is flexible (as compared to mere removal/addition of annotational/compositional variability realization mechanisms) as it does not necessarily require all features to be known in advance as well as that it principally allows for new features to be added or existing ones to be altered.

2.2.3. Variability in Realization Assets

Whether and how configuration knowledge manifests within realization assets greatly depends on the concrete choice of variability realization mechanism. When utilizing an annotative variability realization mechanism with internal annotation, such as preprocessor directives in C/C++, a significant part of the configuration knowledge is present within the realization assets. Even though information on the principle configuration rules for various configuration options may be specified within a variability model, the parts affected by these configuration options and possibly even the method of building variations of the respective asset is contained within the realization asset itself.

With an annotative variability realization mechanism with external annotation, information on which parts are affected by a configuration option and how to build variations of the respective realization assets is externalized (e.g., to a mapping model and the concrete implementation of the variability realization mechanism). However, due to the nature of the utilized 150% model, different variations of the realization assets and, thus, a part of the configuration knowledge are still part of the realization asset.

In contrast, compositional variability realization mechanisms maintain configuration knowledge external to the realization asset in the sense that even the concrete changes to be performed are captured in separate units of composition (e.g., feature modules of FOP). Hence, knowledge with regard to building variations of a realization asset is not part of the realization asset itself. However, the core model of a realization asset may, in isolation, be invalid with regard to syntax and static semantics of its respective language. This stems from the fact that a valid variation of the asset may only be created by adding certain unit(s) of composition. In consequence, the underlying realization asset is itself aware of its use within a software family and, thus, its configurable nature, even though it does not contain the concrete configuration knowledge.

Finally, transformational variability realization assets completely externalize configuration knowledge from realization assets: The basis for building a particular variation of a realization asset is the variation asset as present in the base variant of the software family. The base variant constitutes a valid member of the software family in the form of an executable software system. Information on how to create a different variation according to a particular configuration option is captured within a unit of transformation external to the realization asset. Hence, a realization asset itself does neither contain configuration knowledge nor is it aware of its use within a software family and its configurable nature.

2.3. Types of Software Families

With the umbrella term software family, a number of different types of interrelated sets of software systems may be described. Within this thesis, two particular representatives are of interest–namely Software Product Lines (SPLs) and Software Ecosystems (SECOs). The following sections describe both these concrete types of software families and highlight their similarities as well as their differences using the following characteristics:

1. Maintainer: Number, type and roles of vendors contributing to the software family.
2. Perception of End Users: Which parts of the software family are perceivable by end users of the software family.
3. Configuration Knowledge: Representation of the configuration rules determining validity of combinations of variable assets within a software family.
4. Variant Space: Knowledge of the entirety of all possible members of a software family at specific points in time.
5. Variability Realization Mechanism: Applicability of different types of variability realization mechanisms for the software family.
6. Change of Configuration Knowledge: General process, frequency, synchronization and control over the evolution of configuration knowledge within the software family.

2.3.1. Software Product Lines

A Software Product Line (SPL) is one particular form of software family encompassing a set of closely related software systems. The members of the software family can be distinguished by their functionality and/or other user-visible characteristics such as the technical platform they may be executed on. SPLs capitalize on the similarity of the members of the software family by reusing their shared parts. This makes SPL engineering an approach to software reuse-in-the-large with a shared technological platform. However, in contrast to other approaches to reuse-in-the-large, such as, e.g., frameworks [GS03], SPLs also explicitly model parts specific to only a subset of all family members and, thus, make them available for reuse in different software systems with partially similar requirements. Explicitly formulated configuration knowledge governs which of the possible combinations of variable assets with the common functionality are regarded valid configurations and, thus, variants of the software family. Clements and Northrop [CN02] define an SPL as follows:

Definition 8: Software Product Line (SPL)

"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 [. . .] and that are developed from a common set of core assets in a prescribed way." [CN02]

Pohl et al. [PBvdL05] use a slightly different definition: "Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation." [PBvdL05]. According to Bosch, an SPL is "a set of systems which share a common software architecture and set of reusable components" [Bos00]. However, both these definitions apply to largely similar systems as the one by Clements and Northrop, which is employed within this work. This definition yields concrete characteristics of SPLs:

Maintainer: An SPL is usually maintained by a single company or consortium having control over which combination of variable assets are considered valid variants [PBvdL05, Bos09]. Contributors to the realization assets stem from these vendors or are at least known explicitly as subcontractors. However, the owning company of an SPL may be structured into various divisions or even be aligned with a structure most suitable for the development of the SPL [Bos01].

Perception of End Users: End users of SPLs acquire and employ individual variants of the software family in the form of independent products [Bos00, PBvdL05, CN02]. The configurable nature of the SPL and, thus, its foundation as a software family is generally not visible for end users. However, in some cases, (parts of) the configuration process may be made public to end users, e.g., when the participation in choosing concrete options serves a business value for the owner of the SPL.

Configuration Knowledge: Within an SPL, configuration knowledge is present not only within realization assets but is also represented on a conceptual level. Usually, this configuration knowledge is captured within a variability model [KCH+90, CE00, MA02, PBvdL05, HMPO+08], such as the ones described in Section 2.2.1. The described rules govern validity of configurations regarding the combination of variable assets. Hence, configuration knowledge within a variability model of an SPL is considered prescriptive and the canonical source for configuration rules [SRC+12].

Variant Space: An essential characteristic of SPLs is that, besides the commonalities, also the variabilities of all members of the software family are modeled explicitly [PBvdL05]. With a central owner of the SPL having control over the configuration options, this yields that, theoretically, all possible valid configurations of the SPLs are known at any given time. Thus, the configuration knowledge can be viewed as both prescriptive and complete resulting in a closed variant space.

Variability Realization Mechanism: All members of a software family encompassed by an SPL share a common base functionality and, thus, their products have a similar software architecture. This choice of software architecture may influence the feasibility of employing different types of variability realization mechanisms [SRC+12] (e.g., compositional variability realization mechanisms greatly benefit from encapsulation of functionality such as when using a component-based architecture). However, the complete knowledge of all possible variations of realization assets makes use of all types of variability realization mechanisms presented in Section 2.2.2 as 150% models may be assembled (annotative variability realization mechanism), a core model may be extended (compositional variability realization mechanism) or a base variant may be transformed to a target variant (transformational variability realization mechanism).

Change of Configuration Knowledge: Within an SPL, evolution may lead to a change of configuration knowledge [BFG$^+$02, Bos00], e.g., when modifying interdependencies of variable assets. The control over this process lies with the owner of the SPL [SB99, Bos00]. In the literature, evolution is mostly treated as a phenomenon happening outside the regular development of an SPL. Furthermore, the process of evolution is often perceived as timeless and transactional. As a result, evolution of an SPL is often described as a process that updates the software family in its entirety to a new version. As of that point, customers are perceived as immediately and exclusive utilizing products consisting solely of variable assets of the new SPL. However, in practice, older versions of the SPL or even combinations of variable assets of older and newer versions may have to be maintained to support end users that did not update immediately or completely.

The running example of the configurable TurtleBot driver introduced in Chapter 1 has distinct characteristics of an SPL. The driver software is modeled in the sense of a software family so that is allows creating a number of individual variants. The validity of different configurations and the respective variants is governed by an explicitly specified variability model, in particular, a feature model. The variability model specifies a variant space that is closed (with regard to variability in space) so that all possible variants are theoretically known in advance. Finally, end users of the TurtleBot driver software employ a custom-tailored variant instead of a generic software system.

2.3.2. Software Ecosystems

A Software Ecosystem (SECO) [MS03, Bos09] is another form of software family. Similarly to an SPL, it encompasses a family of closely related and similar yet different software systems. In the center of the SECO is a shared technological platform that may be combined with various extensions to form multiple similar yet different variants of the software family. The validity of different configurations is determined by explicitly stated configuration knowledge. Prominent examples of SECOs are the Eclipse[13] IDE and its extensions, the Android[14] platform and its apps for mobile devices as well as the Linux kernel[15] with its user-contributed extensions [BPT$^+$14].

Hence, SECOs are similar to SPLs in many aspects but both types of software families have distinct differences. Bosch [Bos09] describes steps necessary to move from an SPL to a SECO. Most notably, he states that it is necessary to “open up” the software family to external developers and a community building around the technological platform. On a technical level, this may be achieved by establishing Application Programming Interfaces (APIs) and extension points within the platform and providing documentation for further extension. On a social level, the former product line owner has to take on the role of the *platform leader* responsible for ensuring flourishing of the SECO mostly by gathering developers for extensions, planning future directions of development and ensuring economical soundness for the participation of all possible contributors. Hence, the majority of the economical success of a SECO depends on the platform leader.

For the platform leader, there are a number of valid reasons to establish a SECO instead of maintaining an SPL: For one, it may be the case that there are too many requests for individual

[13] `http://eclipse.org`

[14] `http://android.com`

[15] `https://kernel.org`

features than can be served by the original vendor of the technological platform so that further developers may be obtained by involving the community. Furthermore, for one particular vendor, it may not be feasible to serve all special interests by providing extensions so that developers specialized on a particular niche may be required. In addition, an increase in number and diversity of developers from the community may lead to a subsequent increase in innovation and, thus, interest in the platform by customers. Finally, establishing a shared platform at the center of a SECO may serve a long-term economic success strategy by having not only end customers but also extension developers use and depend on the provided technological platform.

Hence, a SECO encompasses technological as well as social and economical aspects. In the literature, there is no uniform agreement on a definition of *Software Ecosystem*. The following is an excerpt of competing attempts at a definition:

- Bosch: "…the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions." [Bos09]
- Yu et al.: "…a collection of software products that have […] symbiotic relationships [and] may interact through non-software elements, such as customers, users, developers, and markets." [YRB07]
- Bosch and Bosch-Sijtsema: "…a software platform, a set of internal and external developers and a community of domain experts … " [BBS10b]
- McGregor: "…the complete set of entities with which [an organization] interacts to satisfy its goals." [McG09b]
- Boucharas et al.: "…a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them." [BJB09]

Many works discuss social [MG11, SFF^{+}06, vdBJL10] and economical [BCPR09, BvSJ09, JFB09, ILV06, PV04, WGBE12] aspects of SECOs. However, within this thesis, the focus is on technical aspects of SECOs. As a result, *Software Ecosystem* is defined according to Definition 9.

Definition 9: Software Ecosystem (SECO)

A Software Ecosystem (SECO) is a family of closely related software systems with significant commonalities but distinctly different functionality of the individual members realized through contributions to a shared technological platform. The technological platform is provided and maintained by a single vendor or a consortium of multiple vendors. Contributions to the platform are provided and maintained by multiple vendors with largely autonomous responsibilities and usually without a central prescriptive steering mechanism.

This definition entails a number of implications for the general characteristics of a SECO:

Maintainer: Through the explicit opening of the software family to developers from the community, the SECO in its entirety is maintained by a number of vendors. The platform is maintained by a single vendor or multiple, closely related vendors forming a consortium that acts as platform leader [Bos09, McG09a, McG09b]. Success greatly depends on a strong platform leader having the ability to unite the community and mature the shared technological platform, e.g., through creation of suitable variation points. In addition, multiple vendors create and maintain contributions to the shared platform using a distributed and decentralized development approach [Han10].

Perception of End Users: End users of a SECO use products created from the shared technological platform and a number of compatible contributions. However, in contrast to SPLs, in SECOs, end users are aware of the configurable nature of the software family [BBS10b, McG09a, McG09b]. In consequence, configuration of individual products may be within the responsibilities of end users. For example, the Eclipse, Android and Linux SECOs each support configuration of individual products from a selection of contributions [BR09, BR10]. However, as an alternative, ready-assembled products are available in the form of distributions serving different needs.

Configuration Knowledge: In contrast to SPLs, configuration knowledge is usually not represented within one central variability model on a conceptual level [BPT$^+$14]. Instead, the rules governing compatibility of contributions and, thus, the rules for valid configurations are part of realization assets [LL07, Lun09]. For example, the Eclipse ecosystem uses bundles based on Open Services Gateway Initiative (OSGi) components as atomic units of extension. OSGi bundles each provide a manifest file that contains information regarding the dependencies and incompatibilities of the individual extension. The relevant portion of the configuration knowledge is inspected when creating a configuration and attempting to resolve its respective realization assets. The variable elements within the Android SECO have a similar approach. However, central repositories for these extensions, such as the Google Play Store[16], attempt to assemble configuration rules of their respective variable elements within a central basis of configuration knowledge.

Variant Space: Through the large autonomy of vendors with regard to maintaining existing contributions (and their interdependencies) and creating new contributions as well as the lack of a central variability model, the complete variant space of a SECO is inherently unknown in the general case [Bos09]. Hence, a SECO has an open variant space with regard to the available extensions and their potential combinations. However, in some cases, this may not be the case if the platform leader is particularly restrictive with regard to the release policy of contributions. For example, the SECO surrounding Apple's iOS platform[17] with apps for the iPhone and iPad etc. is strictly controlled by its platform leader so that new contributions and even new versions of existing contributions have to be approved before they are released. In this case, the configuration knowledge can be considered complete, which results in a closed variant space.

Variability Realization Mechanism: The general nature of SECOs having an open variant space impacts the possible use of variability realization mechanisms [BPT$^+$14]. Compositional variability realization mechanisms combine a common core model with a selection of possible extensions which naturally aligns well with the largely similar technological structure of the SECO. Transformational variability realization mechanisms may extend the shared technological platform in a similar way but can also be used to further add or modify elements without the need for explicit variation points (see Section 3.2.2.4). However, annotative variability realization mechanisms assemble a family model containing all possible variations of a realization artifact, which depends on complete knowledge of the variant space. Hence, annotative variability realization mechanisms are not feasible for the realization of variants within a SECO.

Change of Configuration Knowledge: In a SECO, a number of different vendors contribute and maintain various contributions to the shared technological platform. Due to the general lack of a prescriptive central steering entity and the resulting autonomy of contribution developers, release cycles do not generally use a common time table for all contributions which leads to largely unsynchronized evolution cycles [JFB09]. Furthermore, within a SECO, minor increments in functionality of contributions and the need to fix defects along with the potential to deploy changes rapidly lead to rather frequent releases of contributions [BBS10b]. To provide guidance for end users regarding the ever evolving SECO, Eclipse utilizes release trains aligned with the major releases of the shared technological platform where releases of most contributions are

[16] https://play.google.com

[17] http://store.apple.com

synchronized. However, in between these release dates, contribution developers have full autonomy regarding frequency and timing of their respective releases. Apart from the functionality provided by a contribution, with new releases, the dependencies and incompatibilities of contributions may evolve as well which leads to changes in the configuration knowledge.

The running example of the TurtleBot driver software presented in Chapter 1 encompasses the characteristics of a SECO with regard to Definition 9. For one, not all features (i.e., the contributions to the shared technological platform) are developed by the same group of people. Instead, some of the features are developed by individual teams operating largely independently and also releasing new versions of their contributions at unsynchronized points in time[18]. Furthermore, some of the contributions explicitly serve a particular niche, such as the `Infrared` and `Ultrasound` features, which are provided only for those TurtleBot robots having the respective custom hardware extensions. In addition, end users of the driver are fully aware of the driver's nature as a software family as they utilize the configuration knowledge to configure products for individual use, which is in contrast to the general SPL approach of specifically providing ready-configured products.

2.3.3. Comparison of Software Product Lines and Software Ecosystems

From the elaborations of the previous two sections, a list of characteristics for both SPLs and SECOs can be assembled. Table 2.2 contrasts these characteristics for both types of software families in order to highlight their similarities and differences.

		Software Product Line (SPL)	Software Ecosystem (SECO)
Similarities		Family of highly configurable software systems	
		Shared technological platform	
		Multiple similar yet different variants	
		Configuration knowledge governs validity of individual variants	
Differences	**Maintainer** [PBvdL05, Bos09]	One or Few	Multiple
	Perception of End Users [BBS09, McG09a]	Individual Products	Individual Products, Technological Platform
	Configuration Knowledge [KCH+90, PBvdL05, LL07]	Explicit in Variability Model	Usually Internal to Realization Assets
	Variant Space [Bos09]	Closed (All products theoretically known in advance.)	Open (Not necessarily all products known in advance.)
	Variability Realization Mechanism [SRC+12, BPT+14]	Annotative, Compositional, Transformational	Compositional, Transformational
	Change of Configuration Knowledge [SB99, Bos00, JFB09, BBS09]	External and Explicitly Controlled Process	Frequent and Largely Unsynchronized Process

Table 2.2.: Comparison of the main characteristics of SPLs and SECOs.

[18] As mentioned before, the evolution stages of the TurtleBot driver described in Section 1.4 and displayed in Table 1.1 were established after the fact. The stages group versions of features that, in some cases, had significant amounts of time between their individual releases.

3. Fundamental Approaches and Technologies of the Thesis

The work in this thesis builds upon a number of established approaches and technologies. The following sections explain these approaches in detail and give reasons as to why they are suitable as foundation for the work presented in the following chapters.

3.1. Model-Driven Software Development

“A model is an abstraction of reality or a representation of a real object or situation.”[1] In software engineering, models are utilized to form the special field of Model-Driven Software Development (MDSD) [GS03]. Within this context, a model can appear in many shapes, e.g., as an ontology, a database scheme or a virtual representation of the physical world. However, within this thesis, the term model is used in the scope of *metamodel*-based software development.

3.1.1. Metamodeling Levels

A metamodel is a further abstraction from a model. It defines the types of model entities and their relations. In metamodel-based software development, each model conforms to a certain metamodel. Hence, the concepts of a model have a corresponding concept in the metamodel they instantiate. This general relation is captured by the Object Management Group (OMG) in the Meta-Object Facility (MOF) as a standard for metamodel-based software development. MOF defines 4 layers M0 to M3, as depicted in Figure 3.1, where each layer instantiates the layer one level above.

Layer M0 is used to describe real-world objects. Layer M1 defines model representations that capture the essential characteristics of the real-world objects from level M0 with regard to a particular use case–in MDSD, this means development of a certain software for a particular domain. Layer M2 represents the metamodel, which defines all possible characteristics of the models on the M1 layer, such as concrete classes with specific attributes. Finally, the M3 layer represents the MOF metametamodeling notation used to define metamodels of the M2 layer. It defines entities such as (meta) classes, attributes and associations as depicted in Figure 3.2. Concepts of the M3 level are defined with the notation of the M3 level itself in order to avoid infinitely many metalevels.

3.1.2. Utilizing Models in Generative Approaches

Besides the comprehensive Complete MOF (CMOF) standard, the OMG has further released the more compact Essential MOF (EMOF) standard. It reduces MOF to the elementary concepts required to define metamodels of the M2 level. A technology that closely resembles the EMOF standard is the metametamodeling notation *Ecore* of the Eclipse Modeling Framework (EMF). The EMF offers a wide range of different tools and technologies to leverage model-based development (see below) and is, thus, used as basis for the implementation of the concepts within this thesis.

[1] `http://www.encyclopedia.com/topic/model_and_modeling.aspx`

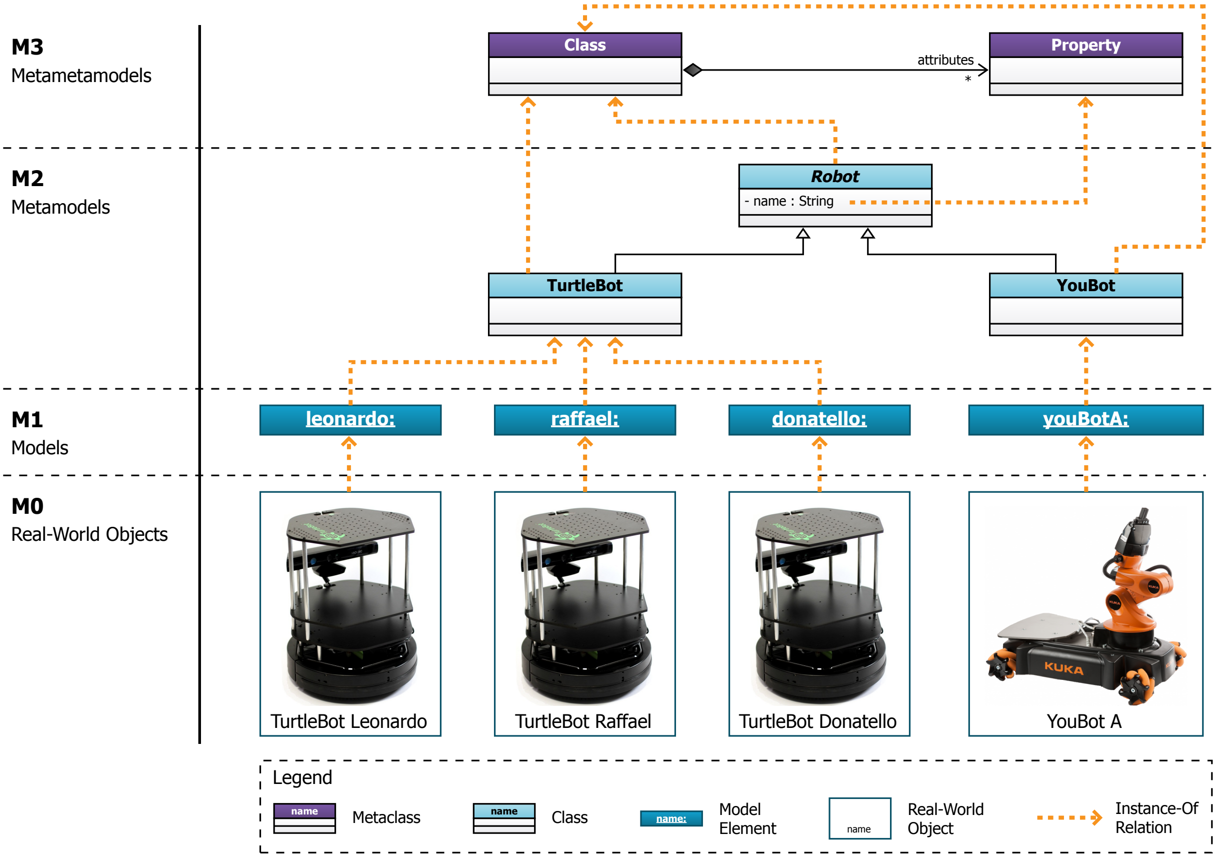


Figure 3.1.: Layers of the MOF where each layer is an instance of the layer one level above.

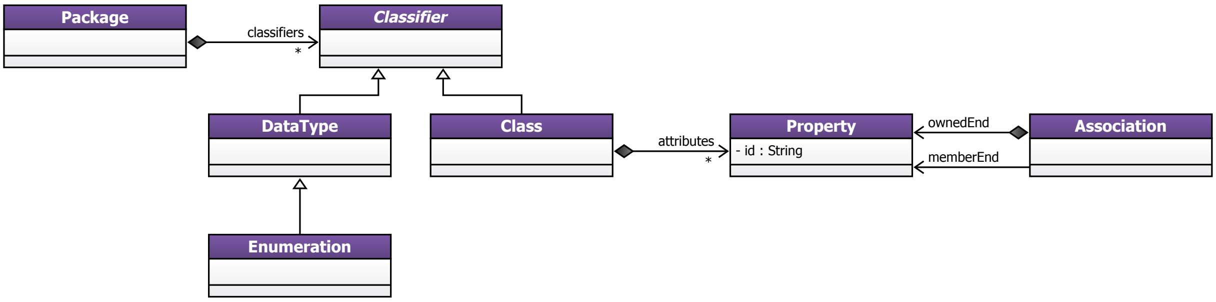


Figure 3.2.: The essential entities of the MOF metametamodel used to define metamodels.

Due to their formalization of concepts, models and metamodels of MDSD may be used as input to generative programming techniques [CE00]. For example, EMF includes a source-code generator for Java. When using generative techniques, models are augmented with additional information and are then used with templates to generate further assets for the models. In the case of EMF, this means that concrete `EClass`es of a metamodel defined using Ecore are used to generate corresponding classes in Java source code.

3.1.3. Representation of Languages using Metamodels

Models in MDSD may be used to represent data in a structured format. However, the method of acquisition of the represented data is an orthogonal issue. For example, data may be acquired from measurements, from persistent storage such as a database or from manual input by users. Furthermore, it is possible to use dedicated languages to define the data represented in models. In particular, it is possible to perceive a metamodel as an abstract syntax of a textual language when it is augmented with a concrete syntax that defines how language constructs are mapped to metamodel elements, e.g., in the form of an attribute grammar [Knu68, Knu71]. For this purpose, the containment hierarchy of elements imposed by the metamodel is exploited to form an Abstract Syntax Tree (AST). Using this approach, artifacts specified in that particular textual language may be perceived as models of the respective metamodel. Generative techniques may be used to derive tools, such as parsers and editors, for the respective languages. The EMF provides facilities to support the definition of textual languages by means of metamodels such as Xtext[2] or EMFText[3]. These tools may be used to define metamodel-based domain-specific languages[4] as well as general purpose languages, such as the Java programming language[5].

Similarly to textual languages, graphical languages may equally be defined with a metamodel formalizing the represented data. In this case, information on the graphical representation has to be supplied along with the metamodel. Generative techniques can again be utilized to create tools, such as graphical viewers and editors, for the languages. EMF provides the Graphical Editing Framework (GEF)[6] as a basis for this procedure. On top of GEF, other techniques are defined that ease definition of graphical languages: The Graphical Modeling Framework (GMF)[7] uses models to define the specific graphical representation of model elements. Epsilon EuGENia[8] generates these models from annotations in the metamodels. Furthermore, Graphiti[9] provides sensible defaults for the generation of a graphical tool infrastructure and Spray[10] defines a textual language to create Graphiti-based tools for graphical languages.

Hence, both textual and graphical languages can be defined on the basis of metamodels and the respective language artifacts can be perceived as models instantiating these metamodels. In consequence, all realization artifacts presented in Section 1.3 can uniformly be represented as models when a suitable metamodel for the type of artifact is provided. Hence, the terms "language" and "model" may be used interchangeably.

To illustrate usage of MDSD and representations on the different MOF layers, Software Fault Trees (SFTs) are used as an example. As described in Chapter 1, SFTs represent

[2] `http://eclipse.org/Xtext`

[3] `http://emftext.org`

[4] `http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo`

[5] `http://jamopp.org/index.php/JaMoPP`

[6] `http://eclipse.org/gef`

[7] `http://eclipse.org/modeling/gmp`

[8] `http://eclipse.org/epsilon/doc/eugenia`

[9] `http://eclipse.org/graphiti`

[10] `https://code.google.com/a/eclipselabs.org/p/spray`

logical combinations of causal error propagation paths, which lead to the fault of a system, in the form of a tree. The example of Figure 1.4 from Chapter 1 represents an artifact of the language for SFTs. Within MDSD, this constitutes an artifact of the M1, i.e., a model, in this case in a graphical representation. To incorporate artifacts of this type into the MDSD process, a suitable metamodel of the M2 level has to be provided. Figure 3.3 shows one possible metamodel for SFTs. `SFTFault`, `SFTGate` etc. are metaclasses and `SFTGateType` is an enumeration data type. Furthermore, relations are defined as references between metaclasses, such as the relation `faults` from `SFTGate` to `SFTFault`.

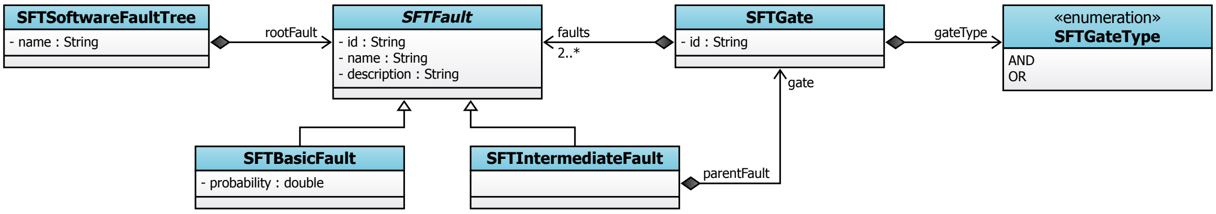


Figure 3.3.: Metamodel for SFTs used to instantiate SFT models.

The metaclasses defined in the metamodel are instances of the type `Class` of the MOF metametamodeling notation of the M3 level as depicted in Figure 3.2. Furthermore, the defined references are instances of the type `Association` of the MOF metametamodeling notation.

Using the metamodel for SFTs as defined in Figure 3.3, it is also possible to support model representations other than the graphical form of Figure 1.4. For example, it is possible to define a concrete syntax that relates textual language constructs to the metamodel entities, e.g., as attribute grammar. A generated parser can then transform the textual representation into an instance of the metamodel. One possible concrete syntax for SFTs as specified with the tool EMFText is depicted in Listing 3.1. Furthermore, an example SFT in textual form is represented in Listing 3.2.

Using the generative facilities of EMF, the respective metamodel may be used to create a structure of classes capable of holding and manipulating data of the respective models. Hence, for the example, it would further be possible to create or load, manipulate and store SFT models conforming to the metamodel in Figure 3.3 using Java source code.

3.1.4. Changing the Model-Representation of Artifacts

For one particular language, different metamodels may exist, e.g., if created by different vendors. Furthermore, it is possible that artifacts of different languages represent (partially) similar information, with each language having its own metamodel. In these cases, it may be necessary to transform models conforming to one metamodel to entities conforming to another metamodel. As far as possible and sensible, the content of the source model should be maintained in the target model. For these use cases, in MDSD, the field of *model transformation* exists.

The respective approaches permit specification of individual operations that alter a source model representation to create a target model representation. Tools for model transformation within the EMF, e.g., are the Atlas Transformation Language (ATL)[11] or the Epsilon Transformation Language (ETL)[12] using MOF Query View Transformation (QVT)[13]. These tools permit specification of rules on how to treat elements of certain metaclasses during transformation.

[11] http://eclipse.org/atl

[12] http://eclipse.org/epsilon/doc/etl

[13] http://omg.org/spec/QVT

```
SYNTAXDEF sft_text
FOR <http://vicci.eu/sft/1.0>
START SFTSoftwareFaultTree

TOKENS {
  DEFINE PROBABILITY $('0''.'('0'..'9')+)$;
}

RULES {
  SFTSoftwareFaultTree ::= "softwareFaultTree" name['"','"'] "{"
      rootFault
    "}";
  SFTBasicFault ::= "basicFault" "{"
      "id" ":" id[TEXT]
      "name" ":" name['"','"']
      ("description" ":" description['"','"'])?
      "probability" ":" probability[PROBABILITY]
    "}";
  SFTIntermediateFault ::= "intermediateFault" "{"
      "id" ":" id[TEXT]
      "name" ":" name['"','"']
      ("description" ":" description['"','"'])?
      gate
    "}";
  SFTGate ::= "gate" "{"
      "id" ":" id[TEXT]
      "gateType" ":" gateType[AND : "AND", OR : "OR"]
      faults (faults)+
    "}";
}
```

Listing 3.1: Concrete syntax for the metamodel of SFTs from Figure 3.3 as specified with EMFText where name matching is used to map syntax rules to metaclasses.

In addition to targeting *types* of elements, it is further possible to target *particular* elements. For this purpose, constraints may be specified that need to match for the particular elements, such as an attribute value being within a certain range. As these tools are defined on the metametalevel (M3; Ecore) and use information on the metalevel (M2) as provided with the transformation rules, they are generally applicable to all metamodels defined with the language of the M3 level–hence the name *general purpose model transformation*. Despite the benefits of the wide area of application, these approaches also have the drawback of not having a direct relation to the model representation (M1) associated with the metamodel so that the specification of transformations may be tedious if a syntax is used that is inherently different from that for the respective models. *Domain-specific model transformation* [RW11] addresses this problem by supplying individual transformation languages for each metamodel that closely resemble the syntax of the respective model. Hence, concepts of the metamodel may directly be employed to specify transformation rules. Figure 3.4 shows an example of a model transformation where an SFT is transformed to an artifact of the more expressive language of CFDs. Model transformation approaches are usually applied on the M1 level to transform concrete artifacts but may also be applied to the M2 level to transform metamodels, e.g., when performing an update of a metamodel before transforming concrete artifacts to conform to that update.

3.1.5. Suitability of Model-Driven Software Development

As most software systems consist of multiple artifacts for different purposes (e.g., design models, source code, documentation, certification material etc.), a variety of languages has to be made

```
softwareFaultTree "RobotCollision" {
  intermediateFault {
    id: RC
    name: "RobotCollision"

    gate {
      id: G1
      gateType: AND

      basicFault {
        id: ODF
        name: "Obstacle Detection Fails"
        probability: 0.003
      }

      intermediateFault {
        id: BF
        name: "Braking Fails"

        gate {
          id: G2
          gateType: OR

          basicFault {
            id: RIM
            name: "Robot in Motion"
            probability: 0.8
          }

          basicFault {
            id: LFS
            name: "Low Friction Surface"
            probability: 0.02
          }
        }
      }
    }
  }
}
```

Listing 3.2: Example of the SFT from Figure 1.4 in a textual representation using the concrete syntax of Listing 3.1.

subject to variability in software families. A model-based representation is possible for artifacts of a wide variety of different languages, e.g., in textual or graphical representation. Hence, with suitable metamodels for the respective languages, all the realization artifacts can uniformly be regarded as models. The explicit structure of artifacts inherited from their metamodels provides a concise definition. Furthermore, generative approaches and facilities for various applications (e.g., modifications, analyses etc.) provide benefits when employing a model-based development approach. With EMF being a widely supported realization of the EMOF standard, EMF Ecore is used as basis for the representation of realization assets as well as the implementation of formalisms and algorithms presented in this thesis.

3.2. Fundamental Variability Management Techniques of the Thesis

In this section, from the aforementioned notations for managing variability, the most suitable ones for the challenges addressed in this thesis are selected. Namely, feature models are used as a basis for an extension to a variability model capturing variability in time (see Chapter 4) and

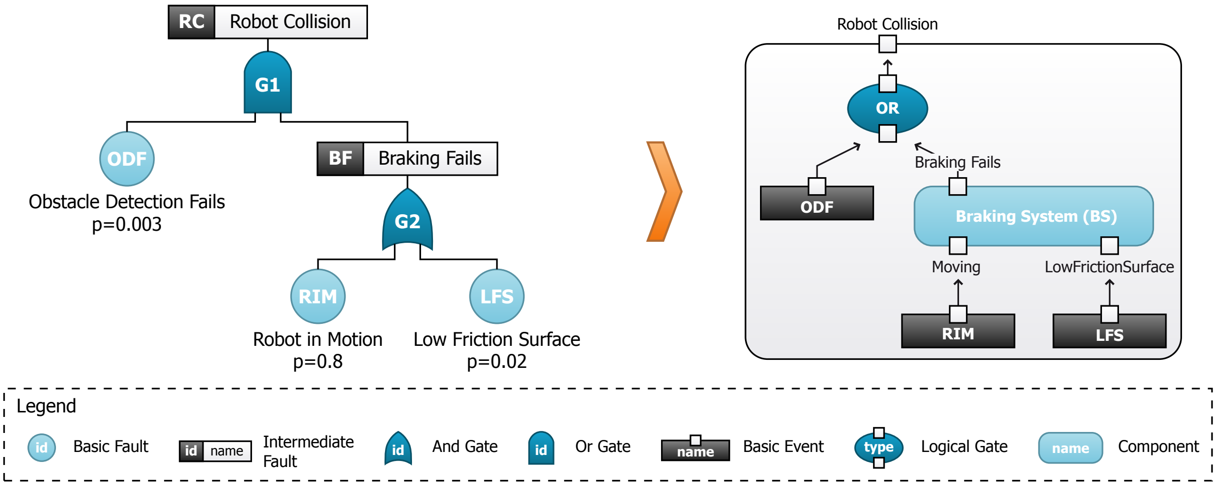


Figure 3.4.: Example of a model transformation procedure transforming an SFT into a corresponding CFD that encapsulates the error propagation paths of the braking system in a separate component.

delta modeling is used as basis for a variability realization using transformations to cope with variability in time (see Chapter 6). The following sections elaborate on these approaches in detail.

3.2.1. Feature Models as Variability Models

Within feature models, a feature is the atomic (most fine-grained) unit of configuration. To model a software family using features, a definition of the term is required. However, there is no general agreement on one particular definition within the literature and multiple slightly different definitions exist [AK09]. The following is an excerpt of alternative definitions:

- Kang et al.: “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.” [KCH$^+$90].
- Batory: “A feature is a product characteristic that is used in distinguishing programs within a family of related programs.” [BSR04].
- Chen et al.: “A feature describes a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements.” [CZZM05].
- Czarnecki and Eisenecker: “A feature is a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.” [CE00].

Within this thesis, the definition of *feature* by Bosch is used for the following explanations as presented in Definition 10.

Definition 10: Feature

“[A feature is] a logical unit of behaviour specified by a set of functional and non-functional requirements.” [Bos00]

3.2.1.1. Feature Model Syntax and Semantics

To explain syntactical elements and the semantics of feature models, Figure 3.5 recapitulates the feature model of the running example of the TurtleBot driver for the sake of easier reading as already presented in Figure 2.2.

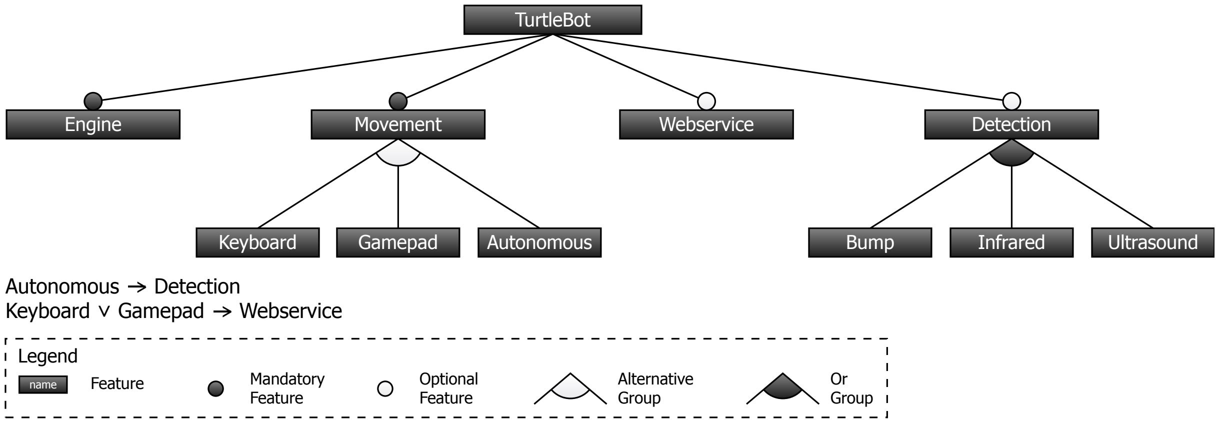


Figure 3.5.: The feature model of the configurable TurtleBot driver software.

The example uses the notation presented by [CE00]. Within the example, *intermediate features* may be distinguished from *leaf features*, where the prior have further child features and the latter have none. Apart from the representation of individual features, a feature model uses distinct syntactical elements to define the *variation type* of both features and feature groups. The variation type specifies the rules for selection for the affected feature or the features contained within an affected group, respectively. For features, the following two variation types are distinguished with their individual graphical representations:

1. *Optional:* The feature may be selected or deselected. This is represented by a hollow dot above the feature (e.g., `Webservice` in Figure 3.5).
2. *Mandatory:* The feature must be selected. This is represented by a filled dot above the feature (e.g., `Engine` in Figure 3.5).

In addition, groups may be used to structure features. The variation type of groups is used to govern selection rules for the members of the group. For a single feature, it is possible to define multiple groups if it is an intermediate feature or no groups at all if it is a leaf feature. The following 3 variation types of groups can be distinguished where only the first two have an explicit graphical representation:

1. *Alternative (XOR):* The group permits selection of exactly one of the contained features. It is represented by a hollow arc spanning all members of the group and located beneath the feature defining the group (e.g., the group beneath `Movement` in Figure 3.5).
2. *Or:* The group allows selection of at least one and up to all of the contained features. It is represented by a filled arc spanning all members of the group and located beneath the feature defining the group (e.g., the group beneath `Detection` in Figure 3.5).
3. *And:* The group permits selection of features by obeying the individual variation types of the contained features. It is implicit in the graphical representation when none of the other group variation types is specified explicitly (not present in Figure 3.5).

Besides the configuration rules imposed by the variation types of both features and groups, the structure of the feature model further defines configuration rules. In particular, the following rules can be derived from the structure:

1. The root feature of a feature model is considered to implicitly have mandatory variation type (e.g., `TurtleBot` in Figure 3.5)
2. Selection of a particular feature requires the respective parent feature to be selected as well. This rule is transitive in the sense that it is applied to the parent feature as well so that further ancestors of the originally selected feature have to be selected as well (e.g., when selecting `Autonomous` in Figure 3.5 then `Movement` has to be selected as well as, transitively, `TurtleBot`).

Applying definition Definition 6 of a configuration to feature models yields that a configuration consists of a subset of or potentially all features defined in the feature model provided that the subset fulfills the configuration logic of the feature model. Figure 3.6 shows an example of a valid configuration of a feature model.

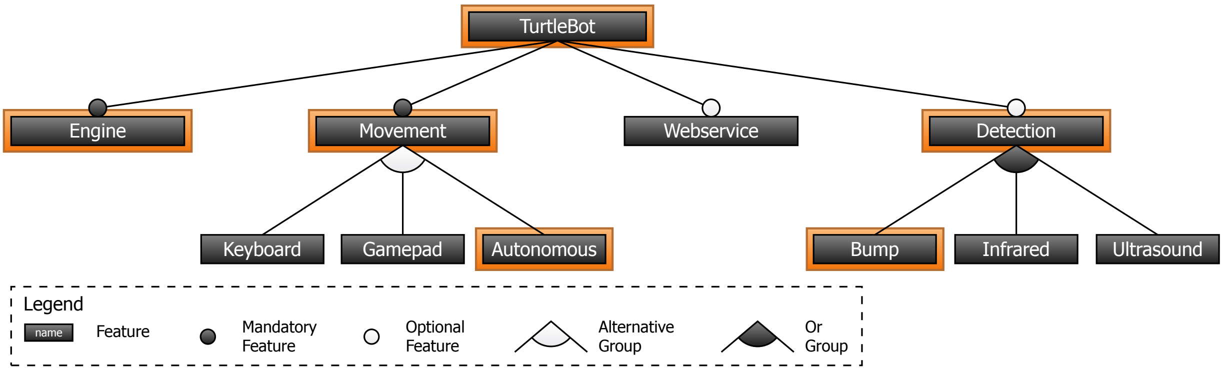


Figure 3.6.: Example configuration of the feature model for the TurtleBot driver.

Through the specified configuration rules, principally, all valid configurations defined by the feature model can be determined explicitly. In consequence, the semantics of a feature model is defined in terms of the type of valid configurations it describes.

3.2.1.2. Cross-Tree Constraints

In addition to the configuration rules inherent to the structure of a feature model and expressed using variation types of both features and groups, it is possible to specify additional *cross-tree constraints* to further restrain the set of valid configurations. This may be necessary when features are not completely isolated from one another so that the choice of one feature may influence options for the selection of other features. A common case are technical incompatibilities of the implementation of two or more features scattered over different branches in the feature model that prohibit selection of certain combinations of the respective features. In addition, it is also possible that economical reasons prohibit or demand certain combinations of features, e.g., if they should be sold only individually or in combination, respectively.

In feature modeling, different approaches exist to specify cross-tree constraints. For one, additional edges may be added to the tree-structure of the feature model essentially transforming it to a graph [CW07]. These edges are used to express *requires* and *excludes* relations. The semantics of a requires relation from a feature `A` to a feature `B` demand that, whenever feature

A is part of a configuration, feature B also has to be selected. In contrast, for an excludes relation from feature A to feature B, the semantics demand that the two features must not be part of a configuration at the same time. Figure 3.7 illustrates the specification of cross-tree constraints as additional edges on the feature model for the TurtleBot driver.

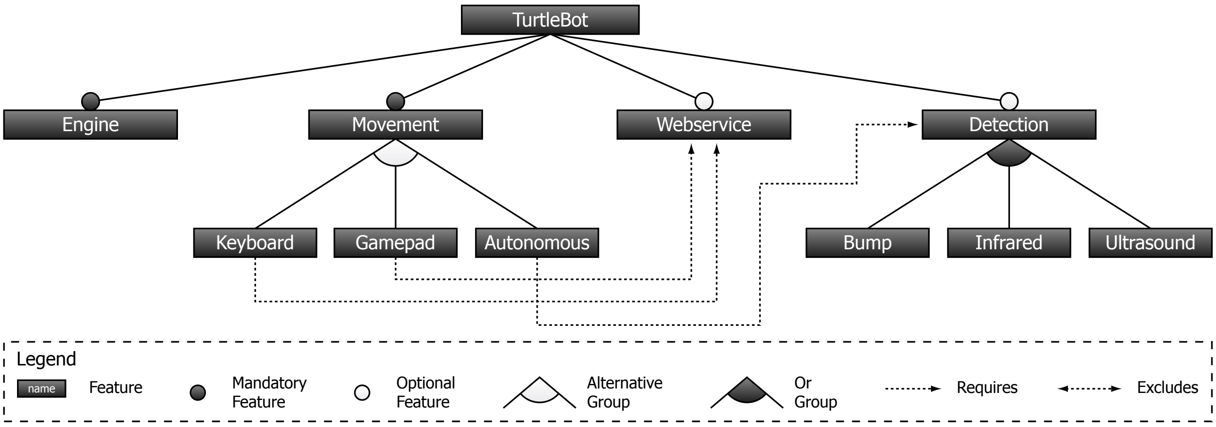


Figure 3.7.: Example of the feature model for the TurtleBot driver with requires constraints specified as additional edges.

Alternatively, it is also possible to use propositional logic over features to express cross-tree constraints [Bat05]. Expressions of propositional logic using feature names as Boolean variables that represent the presence of the respective feature in a configuration are referred to as *feature expressions*. The logical operators ∧, ∨, ¬, → or ↔ may be used to express conjunctions, disjunctions, negations, implications or equivalences, respectively. With these operators, complex conditions may be superimposed on the structure of the feature model. Different notations for cross-tree constraints may use different subsets of these operators but, usually, the selection suffices as a logical basis. Figure 3.8 shows the feature model from Figure 3.5 with cross-tree constraints in propositional logic.

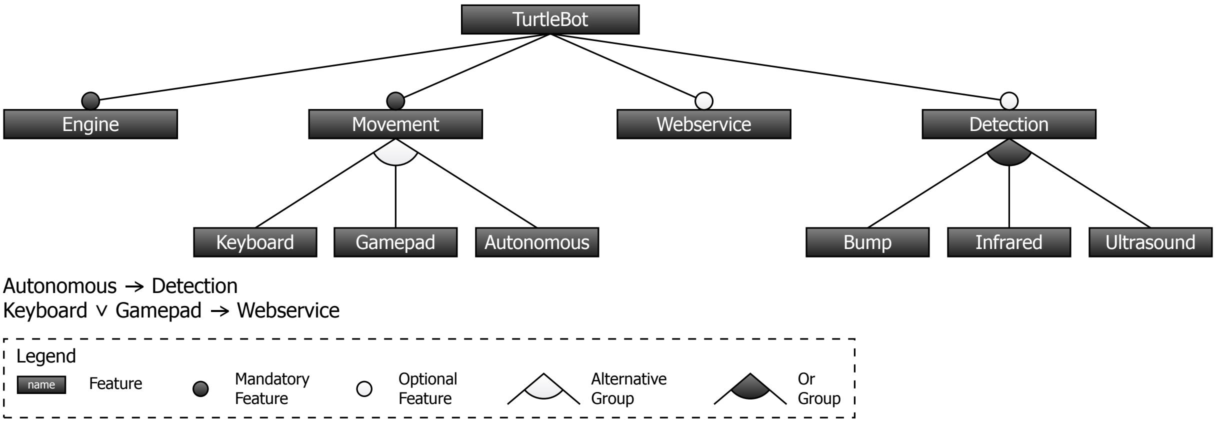


Figure 3.8.: Example of the feature model for the TurtleBot driver with cross-tree constraints specified as propositional logic.

Furthermore, using propositional logic, the requires and excludes edges between features A and B mentioned before may be viewed as A → B and ¬(A ∧ B), respectively. Additionally, it is possible to transform the configuration rules specified in a feature model entirely to propositional logic [CW07]. This may be beneficial when determining validity of configurations or performing analyses, such as finding features that can never be selected [BRCT05].

Configurations defined for a feature model also have to fulfill the additional configuration rules defined by the cross-tree constraints in order to be regarded valid. Hence, the semantics of a feature model are, in part, defined by its cross-tree constraints.

3.2.1.3. Further Feature Modeling Notations

Feature models were first introduced under the name *feature diagrams* by Kang et al. in [KCH$^+$90]. The original notation contained syntactical elements for optional and mandatory features as well as alternative groups. Czarnecki and Eisenecker [CE00] added a first-class syntactical element for or groups and specified the graphical representation used as basis for the example in Figure 3.5. In addition, further feature modeling notations exist that provide alternative and more expressive representations. To verbally discriminate references to feature models using the notation of [CE00] when making an explicit distinction, the expression “common feature models” will be used throughout the thesis. The following presents two extensions to feature models that are relevant for the work in this thesis.

Cardinality-Based Feature Models [CHE05] represent the variation types of features and groups using a minimum and maximum cardinality. For optional features, this means a cardinality of (0..1) and for mandatory features one of (1..1). For alternative groups, a cardinality of (1..1) is used and for or groups one of (1..n) assuming that there are n features in the group. Hence, unbounded groups allowing selection of an arbitrary number of features are modeled by setting the maximum cardinality of a group to the number of child features. And groups do not put restraints on selections within the group other than the variation type of the contained feature. Their cardinality may be modeled as (0..n) to reflect this.

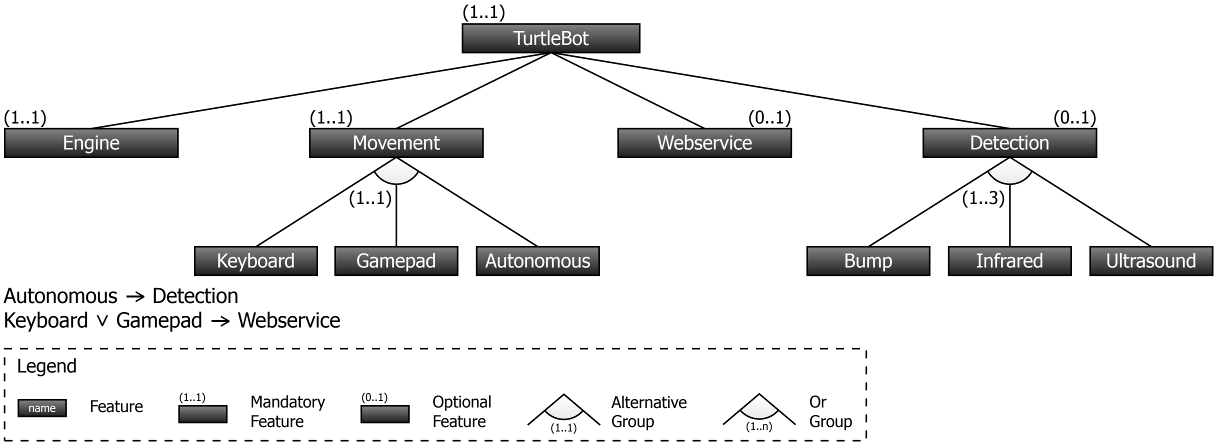


Figure 3.9.: The cardinality-based feature model of the configurable TurtleBot driver software.

Cardinality-based feature models may be used as basis for a graphical representation as well [CHE05]. Figure 3.9 shows a cardinality-based feature model for the TurtleBot example of Chapter 1. The represented diagram is semantically equivalent to that in Figure 3.5. Generally, the cardinality-based representation of feature models subsumes that of using explicit variation types because cardinalities may be used to model all variation types and, additionally, further constraints, e.g., a group that permits selection of 3 to 5 out of 7 contained features. Cross-tree constraints may be applied similarly to common feature models.

For the realization of the concepts in this thesis, internally, cardinality-based feature models are used (see Chapter 7). However, examples throughout the thesis are illustrated in the graphical representation of [CE00] without using cardinalities.

Attributed Feature Models [CHE05] extend common feature models by a first-class syntactic element for attributes. An attribute is a variable with a name and an associated type. Types may be integers, floating point numbers, strings etc. or explicit (ordinal or nominal) enumerations of constant values as well as other user defined types. Regarding the type of an attribute, the domain of possible values differs as well as the cardinality of the set of values from the respective domain. For example, enumerated types and string types have a domain with a cardinality of a constant number, where the latter is usually significantly greater. Furthermore, unbounded integer types have a theoretically infinite cardinality and floating point numbers even have the cardinality of the continuum (uncountably infinite). The cardinality of the employed domain of attributes has a direct effect on whether analyses and calculations on the basis of the defining attributed feature model can be solved or are considered unsolvable due to infinite domains of attributes [BSRC10]. However, cross-tree constraints may be used to restrain the possible configuration options making a subclass of calculations on attributes with types of infinite domains feasible. Figure 3.10 shows an example of an attributed feature model for the TurtleBot driver software where the options for the movement controller were modeled as an attribute of a custom-defined enumerated type. The respective cross-tree constraints were reformulated to obey the notation employing attributes.

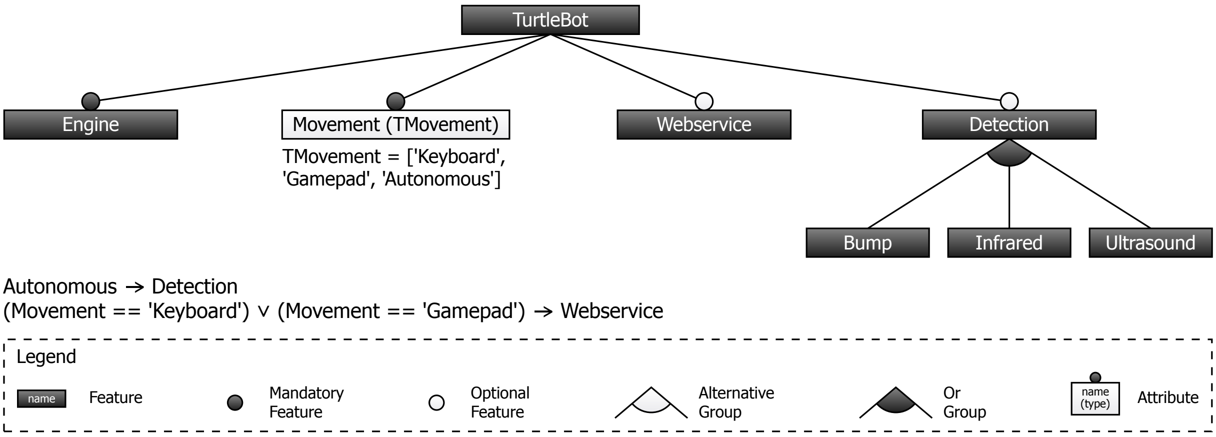


Figure 3.10.: An attributed feature model of the configurable TurtleBot driver software modeling options for the movement controller as attribute.

For attributed feature models, the notion of configuration is extended to not only contain a selection of features but also assignments of concrete values to all relevant attributes so that the constraints are satisfied.

3.2.1.4. Suitability of Feature Modeling as Basic Variability Model

Section 2.2.1 introduced a number of different variability models, which may be used to express configuration options within a software family on a conceptual level. All these notations focus the dimension of variability in space representing various configurations. However, none of the notations is capable of representing variability in time in the form of information regarding evolution of the configuration options. Hence, to meet the requirements posed in Section 3.4.2 with regard to a variability model for variability in space and time, a new type of variability model has to be established. However, feature models form a suitable basis for this new variability model.

Various works compare different variability modeling approaches with regard to their expressive power and application areas [CGR+12, ESDS12, CBA09]. It has been shown that decision models and feature models have the same expressive power [ESDS12]. Hence, configuration knowledge can be transformed from one notation to the other. Similarly, there are approaches that transform OVMs to feature models and vice versa [RFBRC09]. Even though there has been no formal inspection of the relation of feature models and CVL, due to the large similarities of CVL with OVM and also CVL VSpec trees with feature models, a similar expressiveness of CVL and feature models seems plausible (e.g., when considering attributed feature models).

However, decision models, OVM and CVL focus on representing only the variable parts of a software family. In contrast, feature models represent both variabilities and commonalities. With regard to a future integration of variability in time, this quality of feature models is beneficial as also the common parts of a software family are subject to evolution. Hence, it is necessary to capture information regarding variability in time of the respective elements. In consequence, commonalities have to be part of the variability model. Furthermore, according to Berger et al. [BRN+13], feature models are by far the most commonly used type of variability model in practice.

From the different representations of cross-tree constraints, one based on propositional logic seems most suitable due to its capacity to represent even complex constraints in a relatively compact form. A visual representation of requires and excludes constraints may lead to cluttering of the graphical form of a feature model due to the possibly high number of cross-tree constraints in practical application. Furthermore, even though requires and excludes relations form a logical basis, they can only represent constraints of arbitrary complexity if more complex operands than mere presence conditions for a single feature are permitted. However, this is not the case within the graphical representation of constraints.

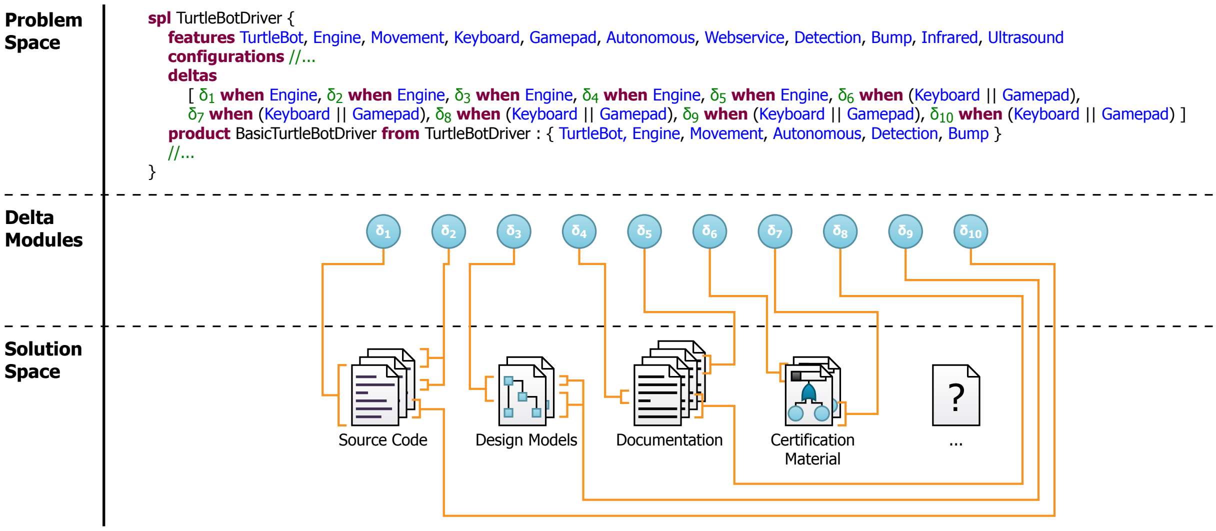
As a consequence, feature models with constraints in propositional logic are used as basis for the conceptual representation of variability in space and in time within this work (see Chapter 4).

3.2.2. Delta Modeling as Variability Realization Mechanism

Delta modeling is an approach to manage variability of software families based on transformations [SBB+10, CHS10]. By means of adding, modifying and removing elements, an existing variant is transformed into another variant of the software family. Within this work, the original variant serving as source of transformation for all possible variants is referred to as *base variant*. The variant resulting from a transformation for a valid configuration is referred to as *target variant*.

Delta modeling encompasses aspects of both a variability realization mechanism and a variability model. Variability realization is performed by transformation of artifacts in accordance to a particular configuration of conceptual variable units. The set of variable units and their interrelation may be specified as part of delta modeling in a *product line declaration* [KHS+14] as illustrated in Figure 3.11. Alternatively, it is also possible to discard the product line declaration in favor of an explicit variability model, such as a feature model, as illustrated in Figure 3.12. Within this work, an extension of feature models is employed as explicit variability model (see Section 4). Hence, the following explanations focus on using delta modeling as variability realization mechanism in conjunction with feature models.

Delta modeling encompasses various constituents to represent and manifest variability in space. These constituents are explained in the following sections.



```
spl TurtleBotDriver {
  features TurtleBot, Engine, Movement, Keyboard, Gamepad, Autonomous, Webservice, Detection, Bump, Infrared, Ultrasound
  configurations //...
  deltas
    [ δ1 when Engine, δ2 when Engine, δ3 when Engine, δ4 when Engine, δ5 when Engine, δ6 when (Keyboard || Gamepad),
    δ7 when (Keyboard || Gamepad), δ8 when (Keyboard || Gamepad), δ9 when (Keyboard || Gamepad), δ10 when (Keyboard || Gamepad) ]
  product BasicTurtleBotDriver from TurtleBotDriver : { TurtleBot, Engine, Movement, Autonomous, Detection, Bump }
  //...
}
```

Figure 3.11.: Example of using delta modeling without explicit variability model.

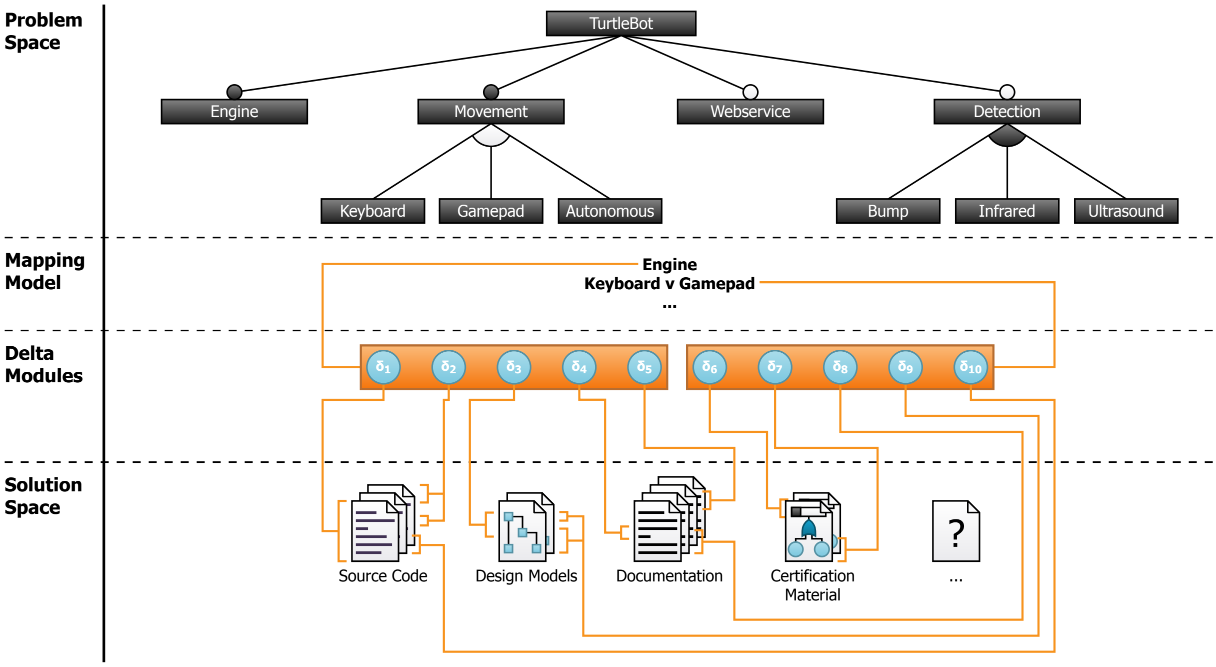


Figure 3.12.: Example of using delta modeling with a feature model as explicit variability model.

3.2.2.1. Delta Languages and Delta Operations

To function as variability realization mechanism, realization artifacts have to be altered in accordance with a particular configuration. To perform the respective changes, delta modeling uses a dedicated domain-specific language-dependent transformation language. On such transformation language is defined for each individual language of a realization asset and is referred to as a *delta language*, e.g., DeltaJava [SBB^{+}10, KHS^{+}14] as delta language for Java. However, the general concepts of delta modeling are language independent [SBB^{+}10, CHS10, DS11]. To disambiguate the language of a realization artifact and the delta language in terminology, within this work, the artifact language is referred to as *source language* when accompanied by a delta language.

A delta language specifies a number of domain-specific transformations available to variability engineers to alter a specific realization artifact in the respective source language for the purposes of configuration. These transformation operations are referred to as *delta operations*. In the most general form, delta operations add, modify or remove individual elements of the source language. However, more complex operations consisting of multiple modifications may be defined in order to capture complex procedures for configuration. The intent of formulating a dedicated delta operation is to enable variability engineers to specify how to enable or disable functionality during variant derivation with as little unintended side effects as possible. Hence, the expressiveness of delta languages is explicitly limited to only address configuration concerns and, thus, reduce the likelihood of damaging system integrity during variant derivation. For example, identifiers of source language assets are usually perceived as being immutable [SBB^{+}10].

A delta language and its delta operations may be defined in accordance with the concrete or abstract syntax of the source language. When using the concrete syntax, keywords of the delta language are interspersed with the source language's formal grammar. For example, the declaration statement for an attribute within a class in Java may be extended with a set of possible preceding keywords to add, modify or remove the respective declaration as illustrated in Listing 3.3. When using the abstract syntax, the respective transformations are performed on the structure underlying the source language but do not necessarily emulate its concrete syntax. Instead, delta operations may use arbitrary names. Listing 3.4 illustrates delta operations based on the abstract syntax of Java.

```
modifies eu.vicci.turtlebot.Engine {
  //Add
  adds private int currentSpeed;

  //Modify
  modifies currentSpeed {double};

  //Remove
  removes currentSpeed;
}
```

Listing 3.3: Example of delta operations to add, modify and remove an attribute in Java using the concrete syntax of Java.

Basing a delta language on the concrete syntax has the benefit of providing only a small gap between the source and target language to improve comprehensibility. However, complexity of the transformation operations is limited by the concrete syntax of the source language so that, e.g., the operations performed on attribute declarations in Java may only have a single parameter (the declaration itself). Basing a delta language on the abstract syntax has the benefit that more complex operations are possible that operate on multiple parameters. This is

```
 1 //Add
 2 Class containingClass = <class::eu.vicci.turtlebot.Engine>;
 3 Type type = new Int();
 4 List<Modifier> modifiers = [new Private()];
 5 createField("currentSpeed", type, modifiers, containingClass);
 6
 7 //Modify
 8 Field field = <field::eu.vicci.turtlebot.Engine#currentSpeed>;
 9 Type newType = new Double();
10 setTypeOfField(newType, field);
11
12 //Remove
13 removeField(field);
```

Listing 3.4: Example of delta operations to add, modify and remove an attribute in Java using the abstract syntax of Java.

especially relevant for the work in this thesis as delta operations handling variability in time may be more complex than those used to handle variability in space. For example, when using a delta operation to move an attribute along the inheritance hierarchy, the destination class or interface for the operation has to be specified in addition to the affected attribute.

To apply delta operations of a delta language to realization artifacts, it is necessary to know at least the affected part of the internal structure of the realization artifact. Hence, delta modeling is a form of white-box composition [Aßm03]. To address individual elements as operands of delta operations, there are two principle approaches in delta modeling: *hierarchical addressing* or *direct addressing*. In hierarchical addressing, the structure imposed by the language of the artifact is navigated to the addressed element by imitating the hierarchy with modification operations. For example, Listing 3.3 illustrates how an attribute is removed (Line 9) that is addressed by navigating its containing class using the `modifies` keyword (Line 1).

With direct addressing, the respective element is addressed by an identifier that can uniquely be resolved to that element, such as an explicit ID. However, it is also possible to have compound identifiers resembling the hierarchical structure of the modified artifact, e.g., as with qualified names for attributes of classes in Java consisting of the fully qualified package name, the class name and the attribute name. Listing 3.4 demonstrates the removal of a field (Line 13) directly identified by its fully qualified name (Line 8). Most commonly, hierarchical addressing is used with delta languages operating on the concrete syntax of a language and direct addressing is used with delta languages operating on the abstract syntax, but it is possible to use different constellations.

3.2.2.2. Manifesting Variability in Delta Modules

Changes associated with one conceptual unit of variability, such as a feature, are represented as a sequence of calls to delta operations. Cohesive calls to delta operations are grouped into *delta modules*. When a delta module is applied, the contained delta operations are executed sequentially to transform the targeted realization artifact.

A delta module may directly relate to one feature, but it is also possible that a delta module is only relevant, when a certain combination of features is part of a configuration. Due to this reason, a delta module is subject to an *application condition* stating under which circumstances it has to be applied. The application condition is an expression in propositional logic over all available features. Besides simple feature presence, it is, thus, possible that a delta module should only be applied when a feature is not present (logical *not*), when multiple features are selected simultaneously (logical *and*), when at least one of a group of features is selected (logical *or*), or when only one alternative of a group of features is selected (logical *xor*). More complex application conditions are possible when combining logical operators. When using the product line declaration of delta modeling, these application conditions are specified explicitly and internal to a delta module [SBB$^+$10]. However, it is also possible to utilize an explicit model containing a mapping of application conditions to affected delta modules for a stricter separation of concerns. Furthermore, when using a feature model along with delta modeling as in this work, the application conditions may be implicit for delta modules and derived from the explicit specification in the variability model, when a mapping between features and associated delta modules exists.

Furthermore, the order of delta modules may not necessarily be completely arbitrary, e.g., if interdependencies of the changes exist. For this purpose, delta modules may specify *application-order constraints* stating demands on the order in which the modules have to be applied. For example, it is possible to specify interdependencies between delta modules stating that certain delta modules have to be applied before the current one using the *before* constraint. Likewise, it is possible to state that a delta module can only be applied after another one using the *after* constraint. However, an application-order constraint of a delta module on another delta modules does not implicitly require the second delta module.

To relate feature models to delta modules, a relation between the conceptual configuration units and the realizing delta modules has to be established. This relation is created with an expression in propositional logic similar to the application condition to create a mapping from delta modules to one or multiple features. Alternatively, the direction of the mapping may be reversed to relate expressions over features to sets of delta modules. To improve separation of concerns, this mapping may be captured in a dedicated model in addition to the variability model and the delta modules. Figure 3.13 visualizes the relation of feature model, mapping model and delta modules. Using the mapping model, a configuration of features can be resolved to the associated set of delta modules (see Section 3.2.3).

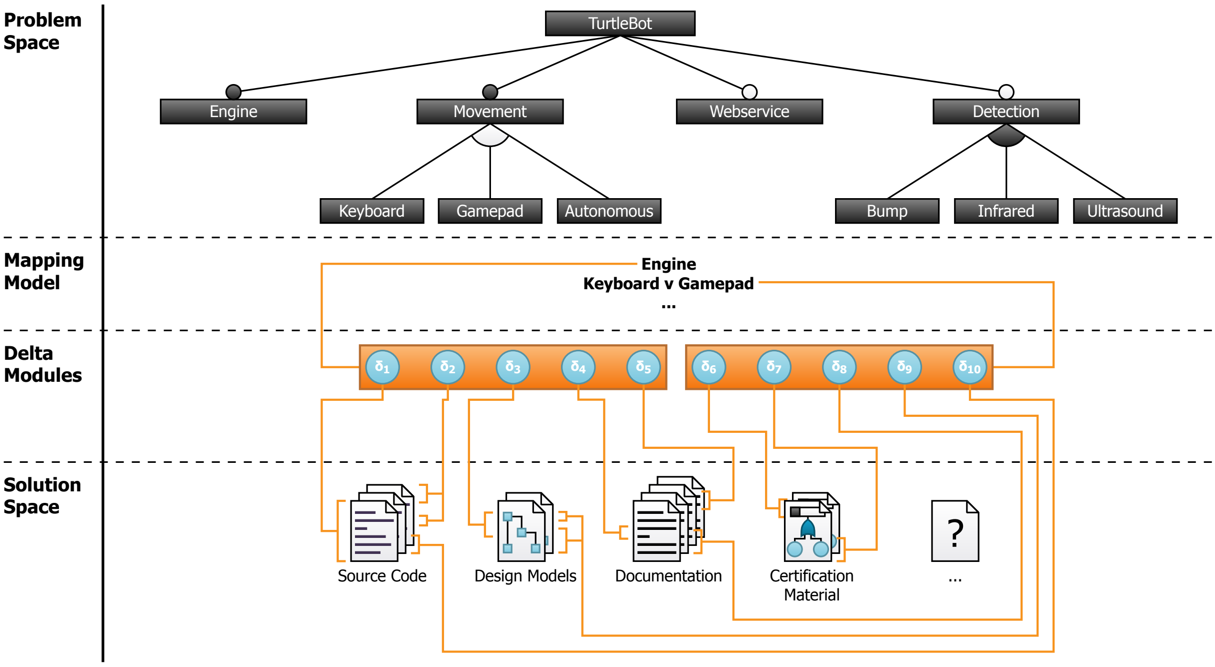


Figure 3.13.: Using a mapping model to relate a feature model to delta modules.

3.2.2.3. Characteristics of Delta Languages and Source Languages

To apply delta modeling for variability management, each individual source language whose artifacts are subject to variability needs to be accompanied by a respective delta language. To address notations commonly used in software development, a range of delta languages was defined for a number of programming and modeling languages including Java [SBB$^+$10, KHS$^+$14], Class Diagrams [Sch10] and Matlab/Simulink [HKM$^+$13]. For the remaining artifacts of a software family that conform to a language without a predefined delta language, a suitable delta language has to be created and integrated into the variant derivation procedure (see Section 5).

There is no principle limitation to the characteristics of a source language for a delta language so that it may be textual, graphical or in any other representation. *Delta-oriented programming* focuses programming languages, such as Java, but *delta modeling* may further be used with conceptual languages such as architecture or design languages [HRRS11, HKR$^+$11, Sch10, SSA13, KLL$^+$14]. Using appropriate metamodels, it is further possible to perceive artifacts of both textual and graphical languages uniformly as models (see Section 3.1). Through this procedure, the concepts of delta modeling are applicable to a wide range of artifacts.

Delta modules are usually specified using a textual language [SBB$^+$10, DS11], but there also are attempts to specify them with graphical languages [HKM$^+$13]. Hence, delta languages themselves may be based on metamodels (see Section 5).

When applied to models as instances of metamodels, delta modeling can be perceived as a specialized form of model transformation [MVG06]. However, in contrast to a general model transformation language, a delta language only provides selected modification operations required for expressing variability. Operations that should not be performed as part of variability, such as changing identifiers, are explicitly prohibited by omitting the respective delta operations. Furthermore, operations may be specified to respect the syntactical and semantical constraints of the source language, e.g., by avoiding dangling references. Finally, variability engineers are not required to learn or understand the full scope of a general model transformation engine, but only that of the reduced functionality of the delta language.

Even though the primary concern of delta modeling is to address configuration (variability in space), its foundation as transformation language makes it principally suitable to be used for evolution (variability in time) as well. Existing approaches in this area inspect evolution of software families *utilizing delta modeling* [HRRS12, SRS13, KLL$^+$14], but not the *application of delta modeling* to perform evolution. Hence, extensions to delta modeling and the respective delta languages are required (see Section 5.3).

3.2.2.4. Suitability of Delta Modeling as Basic Variability Realization Mechanism

Delta modeling as variability realization mechanism has multiple qualities that are beneficial to the goals of this thesis. For one, it natively supports *proactive*, *reactive* and *extractive* development of software family [Kru02]: In a proactive approach to software families, the notion of variability is part of the original development approach so that a managed approach to variability is utilized from the beginning. With a reactive approach to software families, configurable units are only realized if sufficient user-demand exists. Within an extractive approach to software families, multiple individual similar products have been developed, before a managed approach to variability is introduced to explicitly form the software family. Annotative variability realization mechanisms require substantial effort and restructuring to support extractive development. Compositional variability realization mechanisms principally support proactive, reactive and extractive development but require variable elements to be encapsulated as units of composition, which may entail significant restructuring effort for reactive and extractive development. Delta modeling as a transformational variability realization mechanism can cope with proactive, reactive and extractive development without restrictions on the structure of realization assets, as it transforms one variant into another variant of the software family. This is especially relevant, as extractive development of a software family is a common practice to create a software family after initial development of multiple successful individual products for various customers. Hence, utilizing delta modeling accommodates a bandwidth of practical application scenarios and natively supports an economically sound process of software family development.

In addition, the manifestation of configuration options as transformations within delta modules has many benefits (e.g., as compared to textual diffs): Many approaches exploit the knowledge of transformation steps to form variants of a software family to improve the efficiency and effectiveness of analyses when performed not for individual products but for sets of interrelated products. This is essential, as the number of individual variants stemming from the combinatorial complexity of different valid configurations can rarely be addressed by treating each system in isolation. For example, there is work increasing the efficiency of testing software families [KSS13, LLL^{+}14, LSKL12] and analyzing the performance of variants of software families [KST14]. Furthermore, there are approaches analyzing compositionality of changes performed by delta modules [CHS10, BDS13].

Moreover, when utilizing a model-based approach throughout the software family, the application of delta modeling further allows for full traceability of changes from variability model to the affected realization assets. Through this mechanism, it is possible to not only assess *how* an artifact changed but also *why* it changed with regard to the configuration of the variability model. This information may prove essential, e.g., in safety critical software systems that need certification where even minor changes to realization assets may yield the need for a full re-certification, which makes software families infeasible in many cases. By attributing certain changes of a realization asset to a feature and also knowing all further changes associated with that feature, it may be possible to pinpoint the effects of adding or removing certain functionality. This may localize the need for inspection during re-certification and, thus, reduce cost in terms of time and money [SSA13].

Furthermore, delta modeling does not depend on a closed variant space as in SPLs, but can deal with an open variant space where not necessarily all configuration options are known in advance as found in SECO [Bos09, SA13]. This is a discriminating difference to annotative variability mechanisms often used with SPLs. Adding additional functionality can be achieved by adding further delta modules to perform transformations. Even though compositional variability realization mechanisms can extend functionality by a similar principle, their composition technique may depend on the presence of explicit *hooks* to perform gray-box composition [Aßm03], e.g., as with the extension point mechanism of OSGi components utilized within Eclipse. For SECOs, this advantage of delta modeling over compositional variability realization mechanisms may be beneficial: In SECO, the success of products is essentially determined by the underlying platform and the platform leader maintaining and evolving that platform. Hence, when being dependent on suitable hooks provided by the platform in order to create customizations, vendors of extensions and products surrounding the platform are directly dependent on the capabilities of the platform leader. Especially in the early phases of platform development and regarding niche interests for customization, this dependence may lead to a decline in business value or missed business opportunities for vendors of extensions. Delta modeling may provide remedy for this problem by giving control over extensibility of the platform to the vendors of extensions as it does not depend on the presence of explicit hooks.

Finally, delta modeling can handle configuration (variability in space) and (principally) evolution (variability in time) within a single notation [SBB^{+}10, DS11] allowing both to derive products and to modify the software family in response to changed or new requirements (see Section 6).

Due to these beneficial qualities with regard to the overall goal, delta modeling was preferred over the alternatives mentioned in Section 2.2.2 and was chosen as variability realization mechanism for this work.

3.2.3. Variant Derivation Process of Delta Modeling with Feature Models

The variant derivation process transforms a conceptual configuration into an executable software system. For this purpose, all relevant realization assets have to be gathered and the variability realization mechanism has to be invoked on all realization assets subject to variability. For delta modeling, this procedure entails the following steps:

1. Select a configuration
2. Determine all relevant delta modules
3. Evaluate application-order constraints
4. Establish an application sequence of delta modules
5. Copy the base variant
6. Apply all relevant delta modules in determined sequence

Figure 3.14 visualizes this process with a feature model as explicit variability model. The following paragraphs explain the individual steps in more detail.

3.2.3.1. Select a Configuration

A configuration from a feature model is determined by successively selecting and deselecting features until all variability is bound. Each selection or deselection of a feature has to obey the configuration rules imposed by the feature model and, if present, the accompanying constraint model. Furthermore, selection and deselection of a feature must not contradict creation of a valid variant. All variability is bound at the point where exactly one configuration can be derived. The resulting set of features constitutes a valid configuration, as it obeys the configuration rules of the feature model and the constraint model (see Section 3.2.1.1).

In practice, a configuration is often defined using a software configurator. The configurator may represent a feature model graphically, e.g., as presented in Figure 2.2, or in another form suitable for configuration, e.g., as a hierarchical structure of checkboxes each representing one feature. In practice, deselection of features is often implicit in the sense that not selecting a feature is regarded as deselection. The result of the configuration procedure is a set of selected features.

3.2.3.2. Determine all Relevant Delta Modules

Depending on the selected configuration, not necessarily all realization artifacts of the base variant are subject to variability. Furthermore, the specific kind of modification on a realization artifact depends on the concrete configuration. As a consequence, not all possible delta modules used for transformation apply for each valid configuration. Thus, a subset of all delta modules has to be selected according to the provided configuration.

For this purpose, application conditions of delta modules are evaluated (see Section 3.2.2). With the use of a feature model and an explicit mapping model, as in this work, the application conditions are not directly known by the delta modules. Hence, the feature expression used as premise of

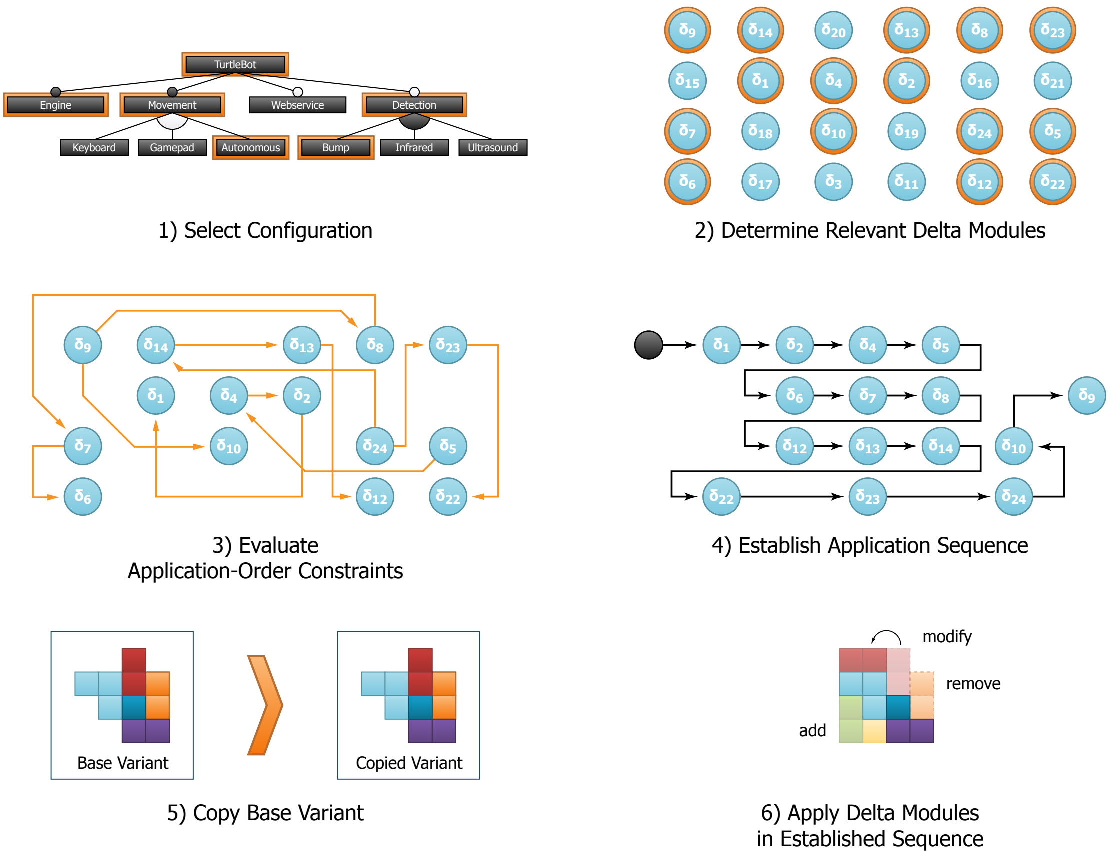


Figure 3.14.: The general process of deriving a variant from a software family using delta modeling.

the mapping is evaluated with the features in the provided configuration. If the premise is satisfied, the set of delta modules used as conclusion of the mapping is added to the overall set of relevant delta modules. This procedure is repeated for all entries in the mapping model to determine the full set of all relevant delta modules for a particular configuration as illustrated in Figure 3.15.

3.2.3.3. Evaluate Application-Order Constraints

Application-order constraints of delta modules specify demands and incompatibilities regarding the sequence of application of individual delta modules (see Section 3.2.2). Hence, application-order constraints may be interpreted as *use before* relations between delta modules, which denote that one delta module has to be applied before another one can be applied. For this purpose, the

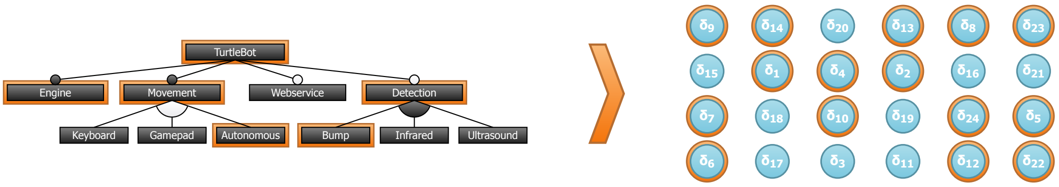


Figure 3.15.: The features selected in a configuration are resolved to delta modules using either internal application conditions of delta modules or an external mapping from feature expressions to delta modules.

before application-order constraint can be used directly and the *after* application-order constraint is interpreted with reversed operands as *use before* relation. These relations, in turn, form a partial order over the respective delta modules that serves as basis for establishing an application sequence. Figure 3.16 illustrates the partial order created by the relations of application-order constraints.

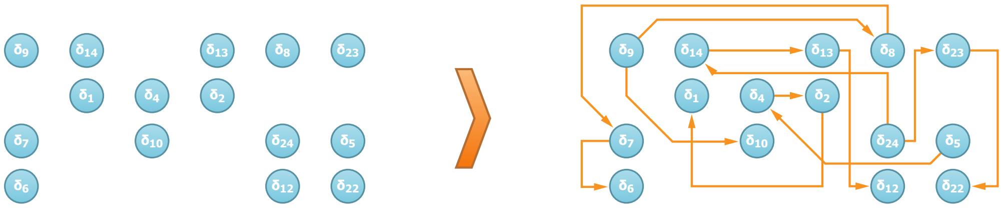


Figure 3.16.: The application-order constraints of the delta modules are evaluated to establish relations between delta modules for their demands and incompatibilities regarding an application sequence.

3.2.3.4. Establish an Application Sequence of Delta Modules

Information from the application-order constraints regarding demands and incompatibilities for the sequence of application are used as input to a topological sorting procedure. This process creates a valid sequence for the delta modules respecting the application-order constraints. As a first step, the relations spanned by the application-order constraints are perceived as a partial order of the delta modules as illustrated by Figure 3.17.

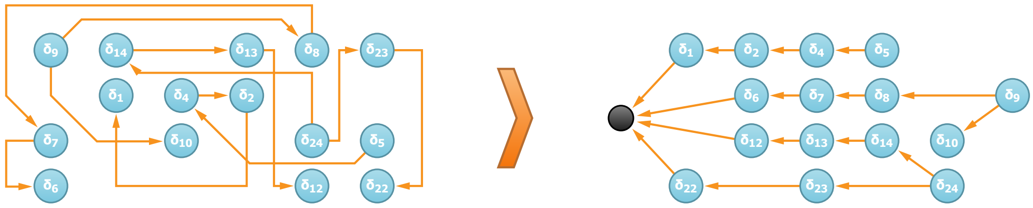


Figure 3.17.: The application-order constraints of delta modules span a partial order when interpreted as *use before* relations.

From this partial order, a concrete application sequence is established in which the delta modules are to be applied. For this purpose, one of the possible paths described by the partial order is chosen. Figure 3.18 illustrates this step.

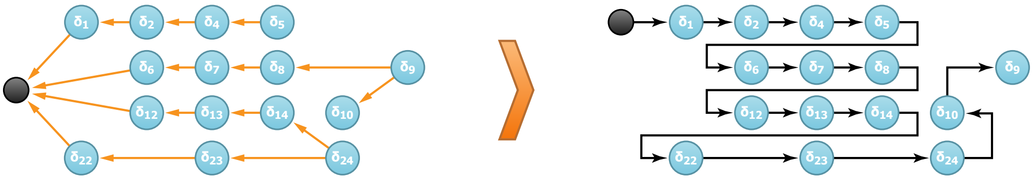


Figure 3.18.: Topological sorting is used to determine a suitable application sequence for the delta modules.

3.2.3.5. Copy the Base Variant

Delta modeling operates by transforming one variant of a software family into another variant that corresponds to the conceptual configuration provided as input. Hence, the source for the transformations specified within delta modules is the base variant of the software family (see Section 3.2.2). To not alter the actual artifacts of the base variant, which would interfere with further variant derivation, the base variant of the software family is copied. Figure 3.19 illustrates this procedure.

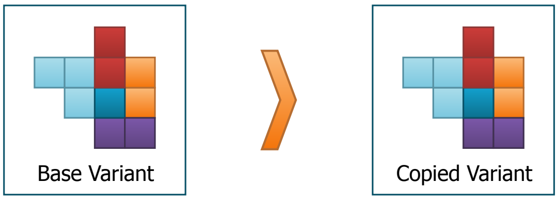


Figure 3.19.: The base variant of the software family is copied to serve as source for the transformations specified within delta modules.

3.2.3.6. Apply All Relevant Delta Modules in Determined Sequence

To perform the changes that transform the base variant to the target variant, the delta modules have to be applied. For this purpose, the previously established application sequence is employed. Due to the topological sorting procedure, it satisfies all application-order constraints posed by the delta modules. Hence, regarding the demands and incompatibilities of delta modules specified as part of the configuration knowledge, delta modules may be executed in that sequence. To persist the changes, each delta module is processed according to the established sequence, and the contained calls to delta operations are executed sequentially. Thus, the base variant is transformed by stepwise add, modify and remove operations (or compound operations thereof) in order to retrieve the target variant of the software family corresponding to the conceptual configuration provided as input to the variant derivation procedure. Figure 3.20 illustrates this general procedure.

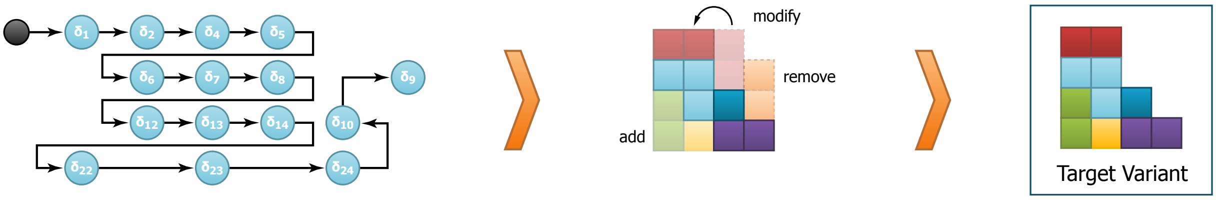


Figure 3.20.: Delta modules are applied in the established sequence and contained delta operations are executed sequentially.

3.3. Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) [Tsa93, Dec03, Lec09] describes a task where concrete values have to be assigned for a given number of *variables* so that a set of *constraints* is satisfied. Values for the variables must further stem from particular *domains* defined for each variable. In the area of Constraint Logic Programming (CLP), a constraint is defined as presented in Definition 11.

Hence, constraints define the connection between a variable and its potential values. In consequence, the sum of all constraints effectively defines all possible constellations of values for

Definition 11: Constraint

"A constraint declaratively specifies a relation between logical variables of different domains." [Coh90]

the variables. Solving a CSP means finding one (or multiple) constellations of values for the given variables so that *all* constraints are satisfied. Algorithmically, CSPs are most commonly solved by employing specialized search and constraint propagation procedures such as backtracking or the AC-3 algorithm [Tsa93, Lec09]. *CSP solvers*, such as Choco [JRL$^+$08], are tools that implement such algorithms to determine solutions to concrete CSPs presented in a suitable format. Formally, a CSP is defined as in Definition 12.

Definition 12: Constraint Satisfaction Problem (CSP)

A Constraint Satisfaction Problem (CSP) is a 3-tuple CSP = $(\mathbb{V}, \mathbb{D}, \mathbb{T})$ with

1. $\mathbb{V} = \{V_1, \dots, V_n\}$: a finite set of variables,
2. $\mathbb{D} = \{D_1, \dots, D_n\}$: a finite set of domains for the variables of $\mathbb{V}$ with $|\mathbb{V}| = |\mathbb{D}|$, and
3. $\mathbb{T} = \{T_1, \dots, T_m\}$: a finite set of constraints formulated as expressions of propositional logic over the variables of $\mathbb{V}$.

A solution of the CSP assigns to each variable $V_i \in \mathbb{V}$ a value $d_j \in D_i$ from the respective domain of the variable such that $\forall T_i \in \mathbb{T} : T_i \equiv \top$ holds.

CSPs are of relevance for software families when performing analyses on the configuration knowledge described in variability models, e.g., to determine whether a feature model permits selection of configurations or if a seemingly optional feature can actually never be deselected [BRCT05, WDS09, MSDLM11, KOD10b]. Beyond that, CSPs may be used for software families to complete partial configurations, as will be demonstrated for the contributions of the thesis in Section 6.3. In this context, multiple valid solutions may exist for one CSP. However, not all of them may be of equal quality with regard to a desired characteristic, e.g., minimizing the number of features that have to be selected.

To incorporate the notion of quality into the CSP solution process, it is possible to utilize an *objective function.* An objective function defines a method of calculating a quality value that is to be maximized (or minimized, respectively) in order to determine an optimal solution within the set of valid solutions to a particular CSP. Conceptually, an objective function may be incorporated into the solution process of a CSP as follows: Upon determining one valid solution for the CSP, the value of the objective function for this solution is calculated. Before the next iteration, an additional constraint is added to the CSP stating that subsequent solutions need to have a value for the objective function that is larger (or less, respectively) than the one just determined. If it exists, the following solution is required to surpass the currently existing one in terms of quality with regard to the objective function. However, CSP solvers provide optimized procedures to incorporate objective functions into the solution process that may not adhere to this conceptual integration. The procedure terminates when no more solutions can be determined for the CSP. The optimal solution is the solution determined most recently.

3.4. Scope

The fundamental approaches and technologies presented in this chapter form the basis for a system for integrated management of variability in space and time in software families. The following sections describe the addressed problem in detail, derive requirements for a suitable solution and state general assumptions underlying the envisaged approach.

3.4.1. Problem Statement

In the literature on evolution of software families [PGT+13, PCA+13, SHA12, PGT+13, LSB+10, SPP+13, SPBL12], variability in time is perceived as a timeless and transactional phenomenon transforming the entire software family and its assets from one state to a revised state with all stakeholders immediately utilizing exclusively the artifacts of the new state. In consequence, it is implicitly assumed that neither the old state of the software family nor combinations of variable assets in different versions need to be supported. However, in practice, this procedure is more complex especially when different vendors maintain individual parts of the software family with independent release cycles. As a result, different versions of variable assets are created that may have different requirements or incompatibilities than their previous versions regarding other variable assets or their respective versions.

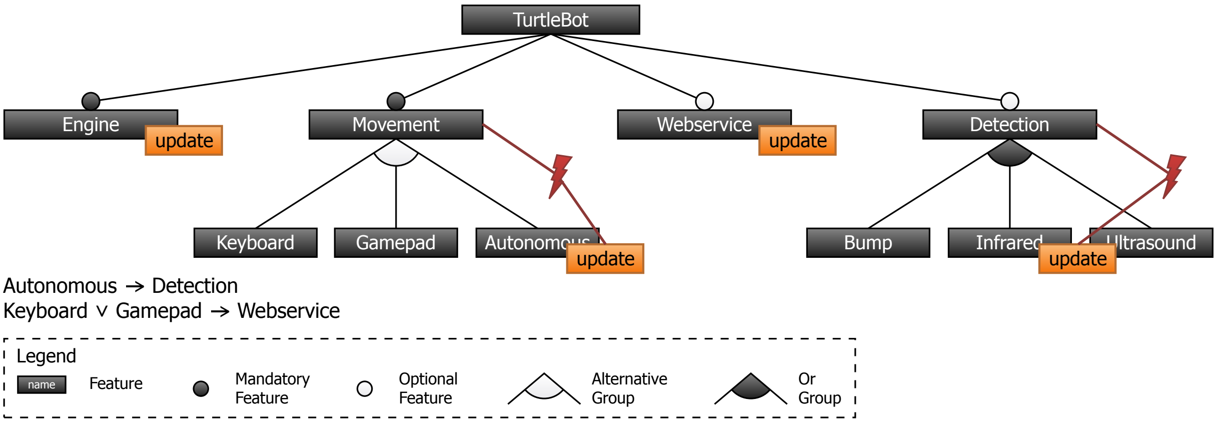


Figure 3.21.: Evolution of variable assets affects configuration knowledge intertwining variability in space and time.

Due to these interdependencies and their effect on valid configurations, variability in time, which created the versions, is a concern of configuration knowledge and, thus, intertwined with variability in space. However, currently existing approaches to manage variability in software families cannot cope with variability in time in an integrated approach. Figure 3.21 illustrates this problem for the TurtleBot driver. As a result, the following problem statement is derived that formulates the main concern of this thesis:

Problem Statement

Due to their effects on configuration knowledge, variability in space and time cannot always be separated completely, but current variability management techniques for software families cannot cope with both dimensions in an integrated approach.

To illustrate the problems in coping with variability in time using current approaches to handling variability in space for software families, Figure 3.22 defines four stereotypes of users for their behavior in updating assets of a software family:

- **Early Adopter**: Always updates to the newest version.
- **Periodic Updater**: Updates infrequently but then completely.
- **Late Adopter**: Updates rarely or not at all.
- **Selective Updater**: Updates/does not update selected features.

Figure 3.22 further illustrates problems of current approaches for variability in space when handling variability in time. With regard to the introduced stereotypes, only the "Early Adopter" can be accommodated, as it is assumed that upon evolution of the commonalities or a subset of the variable assets, the software family in its entirety is present in a new version and that users immediately utilize products encompassing only the most recent versions of all parts of the software family.

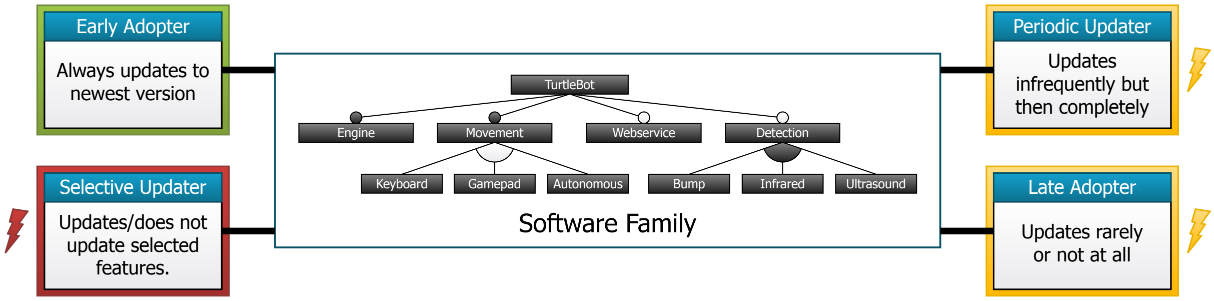


Figure 3.22.: User stereotypes for different behavior in updating assets of a software family illustrating the problems of current variability management approaches regarding variability in time.

The remaining three stereotypes each cause problems with current approaches for software families: The "Periodic Updater" updates from one version of *the entire software family* to a more recent one (possibly manifesting within a single product). However, due to the time between updates, old versions of the software family and its products still need to be accessible, e.g., for reasons of re-installation or maintenance. The prior case may be addressed by explicitly storing snapshots of all possible products (if feasible) or the entire software family and its assets. However, the latter case may not be addressed by current approaches to software families as they (implicitly) assume an old version of a software family to immediately vanish upon release of a new version and are incapable of recreating products of older versions.

Concerns of the "Late Adopter" even amplify problems caused by the "Periodic Updater". Due to possibly extensive amounts of time between updates or the complete lack of updates, support for older versions of the software family, its products and its variable assets is required for possibly a large number of revisions. Using explicit snapshots of the software family is infeasible as each of the snapshots would have to be maintained individually, which leads to tremendous efforts.

Finally, the "Selective Updater" causes the most significant problems to current approaches for software families due to the fact that updates are on a more fine-grained level than entire products of the software family. Instead, individual variable assets may be updated and others may be explicitly excluded from being updated. Evolution of software families such as Eclipse, Android or the TurtleBot driver of the running example need to provide this granularity for variability in space and still cope with variability in time. However, existing solutions only address these concerns on a realization level [SE08, MHP07, LL07, LRL10] but do not acknowledge the interdependencies between versions that restrain configuration options as part of the conceptual configuration knowledge.

3.4.2. Requirements

To remedy the illustrated problems, an integrated management of variability in space and time for software families is required that allows for handling of configuration concerns caused by variability in time on a conceptual level and permits the derivation of variants with different constellations of variable assets in different versions. It seems well-advised to base this approach upon established techniques for handling variability in space in software families. Following the line of argumentation presented in Section 3.2.1 and Section 3.2.2, feature models are the foundation for a variability model and delta modeling forms the basis for a variability realization mechanism. Furthermore, the following three top-level requirements are derived for the work in this thesis to create an integrated approach for managing variability in space and time in software families:

- R1 **Variability Model for Variability in Space and Time**: Feature models and their constraint languages have to be extended to cope with feature versions and their effects on configuration knowledge.
- R2 **Variability Realization Mechanism for Variability in Space and Time**: Delta modeling has to be extended to cope with changes associated with variability in time, and delta languages have to be provided for a wide variety of source languages.
- R3 **Variant Derivation Procedure for Variability in Space and Time**: The variability model and the variability realization mechanism have to be combined in order to allow for the derivation of products from a software family that consist of various variable assets in different versions.

These top-level requirements are addressed in Chapter 4, Chapter 5 and Chapter 6, respectively. Each of these chapters explains the addressed challenges in detail and refines the associated top-level requirements for an adequate solution. Figure 3.23 visualizes the top-level requirements. The respective chapters use a similar figure to illustrate their individual contributions in context of the thesis.

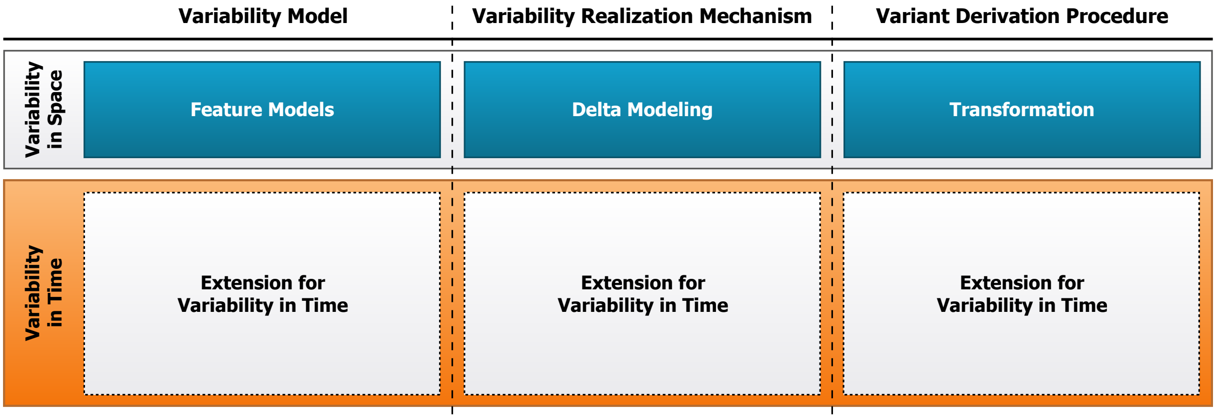

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Extension for Variability in Time
Extension for Variability in Time
Extension for Variability in Time

Figure 3.23.: Illustration of the top-level requirements of the thesis.

3.4.3. Assumptions and Boundaries

Within this work, the specification of configuration knowledge of a software family is assumed to be error and inconsistency-free. In particular, this means that the variability model is assumed to be the canonical source of information regarding configuration knowledge and that possible further sources of constraints on the configuration knowledge, e.g., technical dependencies specified only in delta modules, do not contradict this information. Furthermore, the configuration knowledge is assumed to be sound in the sense that derivation of at least one product is feasible. Hence, identifying, addressing and resolving inconsistencies in configuration knowledge that may arise as side-effect of evolution is out of scope of this thesis.

In addition, the process of configuring a product of a software family is perceived as a timeless and transactional phenomenon. This means that approaches such as staged configuration [CHE04], where multiple stakeholders participate in configuring a product with a potentially complex workflow, are out of scope of the thesis. Integrating such procedures with the concepts of this thesis is an orthogonal challenge that may be addressed as future work.

Furthermore, the effects of evolution are treated on the level of individual variable assets but not all cases of evolving the configuration knowledge itself. In particular, this means that individual features of the software family may be subject to evolution, which creates new versions of these features. The feature versions in turn may introduce new dependencies and incompatibilities with other features, their versions or entire ranges of versions. However, evolution of the configuration knowledge describing variability in space is addressed only for the case of refactorings [TBK09] (maintaining the set of valid configurations) as well as generalizations [TBK09] (permitting a super set of configurations). Specializations [TBK09] (reducing the set of valid configurations) and arbitrary edits to the configuration knowledge [TBK09] are considered out of scope. Work previous to this thesis [SSA13] has revealed that generalization encompasses the vast majority of evolution scenarios. Furthermore, it showed that different vendors maintain individual variable assets with unsynchronized evolution cycles. Hence, these cases are in focus.

Additionally, all realization artifacts are assumed to be model-based with a suitable metamodel. However, the concepts described are independent of a concrete realization as models and can be applied to other representations of realization artifacts as well.

Moreover, the origin of the many example notations for realization assets in the domain of safety-critical systems is strictly incidental. Challenges from safety-critical systems and especially modular safety certification are considered out of scope of the thesis. However, the respective challenges may be addressed on basis of some of the concepts introduced within the thesis as part of future work (see Section 9.1).

Part II.

Integrated Management of Variability in Space and Time

4. Capturing Variability in Space and Time with Hyper-Feature Models

The contents of this chapter are largely based on the work published in [SSA14a, SSA13].

Summary *SPLs and especially SECOs are subject to evolution to adapt to new or changed requirements resulting in different versions of the software family and its variable assets. These versions may have to be maintained and used for products even after they were superseded by newer versions, e.g., because not all customers upgrade immediately or completely. Feature models capture variability in space (configuration), but not variability in time (evolution), which makes it impossible to respect versions of variable assets in product definitions on a conceptual level. A suitable extension to feature models may address this problem. In this chapter, Hyper-Feature Models (HFMs) are proposed as a remedy explicitly providing feature versions as configurable units. Hence, both features and feature versions may be used for product definition. Furthermore, a version-aware constraint language to specify dependencies between features and ranges of feature versions is defined.*

4.1. Feature Models Cannot Capture Variability in Time

Software evolution yields different revisions of software systems. These revisions are represented as separate versions. This effect is also present in SPLs and SECOs, which have explicit configuration knowledge, e.g., captured in a variability model such as a feature model.

As mentioned in Section 3.2.1, feature models capture variability in space (configuration), but not variability in time (evolution). Thus, there may be only exactly one version of each variable asset available for product definition in the variability model. Even though multiple variants of the system are supported, there is merely one version of each of these products and, thus, the product family. However, explicit support of multiple versions is necessary for highly configurable systems of SPLs, e.g., when older versions of a system have to be maintained to support customers who did not upgrade to the most recent versions. The problem is even more present in SECOs where multiple contributors add variable assets to a configurable system that does not have a synchronized development cycle and end-users may configure products individually using combinations of different versions of variable assets. In consequence, various vendors publish new versions of variable assets that may depend on certain versions or version ranges of other assets.

Currently, this dependency can only be expressed as part of an asset's realization in the solution space, but not on a conceptual level in the problem space. However, the choice of a version of a certain configurable asset essentially influences configuration options of the system for a particular variant at a given time.

Using the example of the TurtleBot driver (see Section 1), the problem is illustrated in Figure 4.1. The engine of the initial version 1.0 of the TurtleBot was a retrofitted vacuum cleaning robot of the make iRobot Create[1]. Over the course of time, the TurtleBot was revised to a new version 2.0.

[1] `http://irobot.com/create`

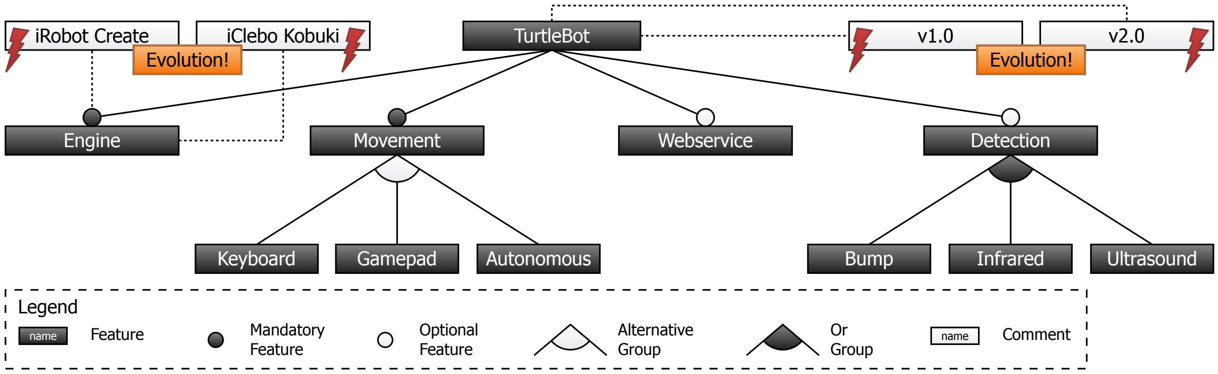


Figure 4.1.: Evolution yields new versions of features with interdependencies and incompatibilities. However, common feature models can only represent features for variability in space but not their versions for variability in time.

Among other things, in the new version, the original engine was replaced by a dedicated development platform for driving robots of the make iClebo Kobuki[2]. Even though the new engine serves the same purpose of locomotion, the driver software differs because of another protocol of operating the engine. However, parts of the original realization of the engine are used to create the new version so that the new version is an incremental change to the old version. Hence, both versions are related along a chronological development line. Further evolution of the realization of either one of the development lines may yield branches which relate versions in a tree structure, e.g., when development of the driver is continued for both the Create and Kobuki engines.

Due to similar functionality of the new engine, the feature itself maintains its identity. However, because of the different software required, the feature should be represented in a new version. Yet, the old version of the feature has to be maintained as well because instances of the old TurtleBot version are still in use and should be supported further by the driver. However, not all combinations of versions of different features are possible as, e.g., the new version of the TurtleBot is incompatible with the old version of the engine and the old version of the TurtleBot depends on the old version of the engine. Hence, the notion of versions has an impact on the configuration options of the driver software–even on a conceptual level.

When using common feature models, this is problematic for two reasons: First, common feature models are not capable of representing versions as yielded by variability in time. Second, feature models possess no mechanism to express interdependencies and incompatibilities between different versions of features, esp. when a range of versions satisfies a requirement, such as “all versions greater than 2.0”.

A number of extensions have been designed for common feature models to express various specific concerns of configuration [CBA09, BSRC10]. However, all these approaches solely focus on configuring variable assets of an SPL or a SECO for a specified point in time, but do not consider that there may be different versions of variable assets that have to be respected. For example, in the SECO surrounding Eclipse[3], users that install additional extensions (which can be perceived as features) are often confronted with dependencies on other extensions *in a certain version range*. Previous work of this thesis has analyzed the Eclipse SECO and has determined that implementation versions resulting from evolution may entail incompatibilities with other versions or introduce new requirements to other versions [SA13]. Furthermore, apart

[2] http://kobuki.yujinrobot.com
[3] http://eclipse.org

from individual versions, requirements and incompatibilities may apply to entire ranges of versions. Due to these reasons, the selection of an implementation version becomes relevant for defining a valid product of a software family making it a conceptual concern of the problem space that should be reflected in the variability model.

Based on these scenarios and the previous work on analyzing the evolution of SECOs [SA13], `R1` of Section 3.4.2 is refined to the following requirements for a variability modeling approach capturing aspects of variability in space and time on a conceptual level:

`R1.1` **Conceptual Level**: The notion of versions has to be lifted from the mere realization level of implementation artifacts to a conceptual level independent of a particular type of realization asset.

`R1.2` **Versions as Configurable Units**: It has to be possible to use versions to configure products of a software family (in contrast to using them merely for analysis or versioning the entire software family).

`R1.3` **Development Lines with Branching**: It has to be possible to represent the logical relation of versions including branching, i.e., whether a version may have multiple direct successors.

`R1.4` **Version Interdependencies**: It has to be possible to express dependencies on and incompatibilities with versions and version ranges.

`R1.5` **Intention of Variability in Time**: It is necessary to explicitly represent the intention of variability in time to have a sufficient distinction from concepts for variability in space.

To address these requirements, the following sections introduce *Hyper-Feature Models (HFMs)* as an extension to common feature models and a version-aware constraint language that allows capturing both variability in space and time on a conceptual level.

4.2. Formal Definition of Feature Models

A feature model organizes features in a tree where selection of a feature implicitly selects its parent feature [PBvdL05]. Features can be either *optional* or *mandatory* and may be grouped into *alternative* or *or* groups permitting the selection of exactly one or at least one feature of the group respectively (see Section 3.2.1).

To give a precise definition of HFMs as extension to common feature models, first a formal definition of cardinality-based feature models is presented as basis. For this formal definition, the formalization of [SLW12] is employed. Definition 13 defines the syntax of feature models and Definition 14 their semantics. In the definitions, $\mathcal{P}(X)$ denotes the power set of a set X.

Definition 13: Feature Model Syntax

A feature model is a 4-tuple $\mathrm{FM} = (\mathcal{F}, \prec, \lambda, \Phi)$ with

1. $\mathcal{F}$: a finite set of features,
2. $\prec \subseteq \mathcal{F} \times \mathcal{F}$: a relation forming a rooted tree on $\mathcal{F}$,
3. $\lambda : \mathcal{P}(\mathcal{F}) \rightharpoonup (\mathbb{N}_0 \times \mathbb{N}_0)$: a function assigning minimum and maximum cardinality to features and feature groups, and
4. Φ: a set of propositional formulas over $\mathcal{F}$ representing cross-tree constraints.

Using a formally defined model to represent variability allows for determining possible configurations, performing checks (e.g., to detect dead features) and calculating metrics (e.g., number of configurations). In [SLW12], the semantics of a feature model are defined in terms of the specified set of valid configurations similarly to Definition 14.

Definition 14: Feature Model Semantics

Let $f_R \in \mathcal{F}$ be the root feature of a feature model. A configuration $\mathcal{C} \subseteq \mathcal{F}$ of a feature model has to satisfy the following conditions to be valid:

1. The root feature is part of all configurations.
   $f_R \in \mathcal{C}$
2. Cardinality constraints of features and feature groups are respected by the configuration.
   $f_1 \in \mathcal{C} \wedge F = \{f_2 \in \mathcal{F} | f_1 \prec f_2\} \wedge \lambda(F) = (k, l) \Rightarrow k \leq |F \cap \mathcal{C}| \leq l$
3. For each selected feature, the parent feature has to be in the configuration.
   $f_1 \in \mathcal{C} \wedge f_2 \in \mathcal{F} \wedge f_2 \prec f_1 \Rightarrow f_2 \in \mathcal{C}$
4. All propositional formulas of the cross-tree constraints have to be satisfied by the configuration.
   $\mathcal{C} \models \bigwedge_{\phi_i \in \Phi} \phi_i$

Let $\mathbb{F}$ be the set of all possible valid feature models over $\mathcal{F}$. The semantic evaluation function is defined as $[\![\cdot]\!] : \mathbb{F} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{F}))$. The semantics of the feature model FM is defined as $[\![\mathrm{FM}]\!] = \mathbb{C}$ such that $\forall \mathcal{C} \in \mathbb{C} : \mathcal{C}$ is valid for FM.

Using this formalization, the example displayed in Figure 3.5 and Figure 3.9 may be represented as illustrated in Formula 1.

4.3. Definition of Hyper-Feature Models

To remedy the shortcomings of common feature models with regard to representing variability in time, this section introduces *Hyper-Feature Models (HFMs)*[4]. HFMs contain a new concept besides features and feature groups to model *feature versions* as configurable units. Hence, HFMs are intended to capture both variability in space and time on a conceptual level in a unified notation.

A feature version represents a snapshot of its containing feature's realization in the solution space at a given point in time and, thus, addresses requirement `R1.2` of Section 4.2. In common feature models, a feature represents the atomic unit of configuration. However, HFMs further divide features into multiple feature versions. These feature versions may be used to define configurations and, thus, variants in the form of software systems. Hence, HFMs allow for more fine-grained control over the configuration than common feature models.

It is explicitly not the intention to represent changes to the feature model or its structure using the version concept of HFMs, such as a feature changing from optional to mandatory or a new feature being introduced [TBK09], as this is *versioning of the feature model* and not *supporting versions within a feature model* for configuration purposes.

Figure 4.2 depicts an example HFM in a graphical notation showing features and feature versions for the driver software of the TurtleBot robot platform as introduced in Chapter 1. The different types of engines were modeled as versions of the `Engine` feature instead of as alternative

[4] The name "Hyper-Feature Model" was chosen due to the representation of multiple dimensions within one notation similarly, e.g., to *Hyperspace Programming* [OT02] as a multi-dimensional separation of concerns [SSA14a].

Formula 1: Formalization of the feature model for the TurtleBot driver software.

1. $\mathcal{F} = \{\text{TurtleBot}, \text{Engine}, \text{Movement}, \text{Keyboard}, \text{Gamepad}, \text{Autonomous}, \text{Webservice}, \text{Detection}, \text{Bump}, \text{Infrared}, \text{Ultrasound}\}$
2. $\prec = (\text{TurtleBot}, \text{Engine}),\ (\text{TurtleBot}, \text{Movement}),\ (\text{TurtleBot}, \text{Webservice}),\ (\text{TurtleBot}, \text{Detection}),\ (\text{Movement}, \text{Keyboard}),\ (\text{Movement}, \text{Gamepad}),\ (\text{Movement}, \text{Autonomous}),\ (\text{Detection}, \text{Bump}),\ (\text{Detection}, \text{Infrared}),\ (\text{Detection}, \text{Ultrasound})$
3. $\lambda(\{\text{TurtleBot}\}) = (1, 1)$
   $\lambda(\{\text{Engine}, \text{Movement}, \text{Webservice}, \text{Detection}\}) = (2, 4)$
   $\lambda(\{\text{Engine}\}) = (1, 1)$
   $\lambda(\{\text{Movement}\}) = (1, 1)$
   $\lambda(\{\text{Keyboard}, \text{Gamepad}, \text{Autonomous}\}) = (1, 1)$
   $\lambda(\{\text{Keyboard}\}) = (0, 1)$
   $\lambda(\{\text{Gamepad}\}) = (0, 1)$
   $\lambda(\{\text{Autonomous}\}) = (0, 1)$
   $\lambda(\{\text{Webservice}\}) = (0, 1)$
   $\lambda(\{\text{Detection}\}) = (0, 1)$
   $\lambda(\{\text{Bump}, \text{Infrared}, \text{Ultrasound}\}) = (1, 3)$
   $\lambda(\{\text{Bump}\}) = (0, 1)$
   $\lambda(\{\text{Infrared}\}) = (0, 1)$
   $\lambda(\{\text{Ultrasound}\}) = (0, 1)$
4. $\Phi = \{\langle\text{Autonomous} \rightarrow \text{Detection}\rangle, \langle\text{Keyboard} \vee \text{Gamepad} \rightarrow \text{Webservice}\rangle\}$

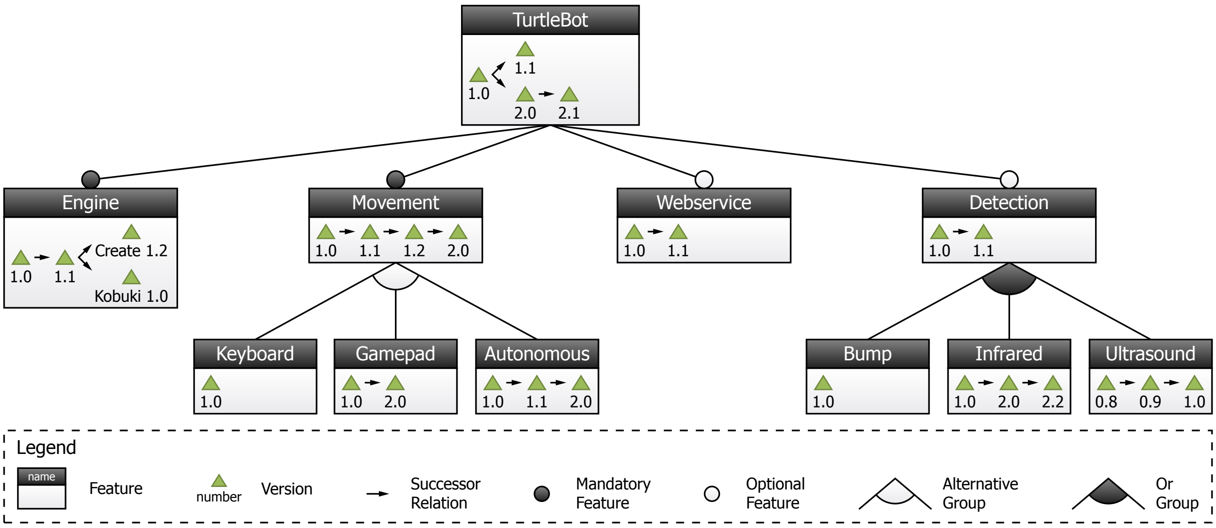


Figure 4.2.: Graphical example of an HFM with feature versions as configurable elements.

child features of the engine, as they are not provided as configuration options of the TurtleBot, but merely as part of evolution. In addition, the feature `Engine` serves the same purposes of enabling locomotion for the two types of engine and only the technical realization varies. All other variable assets are available in at least one version, such as version *1.0* of `Keyboard`.

To provide a precise definition of the syntax along with a rigid semantics, the following defines HFMs using set theory. Definition 15 builds upon Definition 13 for feature models and extends it for the purpose of representing versions.

Definition 15: Hyper-Feature Model Syntax

A Hyper-Feature Model is a 6-tuple HFM $= (\mathcal{F}, \mathcal{V}, \prec, \lambda, \Upsilon, \text{pred})$ with

1. $\mathcal{F}$: a finite set of features,
2. $\mathcal{V}$: a finite set of feature versions,
3. $\prec \subseteq \mathcal{F} \times \mathcal{F}$: a decomposition relation on $\mathcal{F}$ forming a rooted tree of features,
4. $\lambda : \mathcal{P}(\mathcal{F}) \rightharpoonup (\mathbb{N}_0 \times \mathbb{N}_0)$: a function assigning minimum and maximum cardinality to features and feature groups,
5. $\Upsilon : \mathcal{F} \rightharpoonup \mathcal{P}(\mathcal{V})$: a function relating a feature to the set of its versions, and
6. $\text{pred} : \mathcal{V} \rightharpoonup \mathcal{V} \cup \{\varepsilon\}$: a predecessor function relating each $v \in \mathcal{V}$ to its predecessor $v_p \in \mathcal{V}$ or ε (the empty version) if v does not have a predecessor.

Due to the effects of evolution, a feature may be present in multiple versions. These versions are not completely detached from one another, but have a successor/predecessor relation (except for the initial and most recent versions). This relation is mostly established by the chronological order of creation of versions so that newer versions supersede older versions, e.g., versions *1.2* and *1.1* of `Movement` in Figure 4.2. The predecessor relation is captured using the function *pred*.

However, a particular version may serve as predecessor for multiple versions due to branching, when a new version has a predecessor other than the previously most recent version. Figure 4.2 illustrates branching with versions *Create 1.2* and *Kobuki 1.0* of feature `Engine`. HFMs support branching by allowing feature versions to have multiple successors, but only at most one predecessor. This relation spans up the branching tree caused by evolution, which addresses requirement `R1.3`. Hence, in addition to the conditions given in Definition 15, a number of well-formedness rules have to be satisfied for a valid HFM:

Definition 16: Hyper-Feature Model Well-Formedness

Let ε be the empty version. A Hyper-Feature Model has to satisfy the following conditions to be well-formed:

1. Each feature has potentially multiple versions with exactly one initial version not having a predecessor.
   $(\forall f \in \mathcal{F} : \Upsilon(f) = \mathcal{V}_f \rightarrow |\mathcal{V}_f| \geq 1) \wedge (\exists! v \in \mathcal{V}_f : \text{pred}(v) = \varepsilon)$
2. Each version has to belong to exactly one feature.
   $\forall v \in \mathcal{V} : \Upsilon(f_1) = \mathcal{V}_1 \wedge v \in \mathcal{V}_1 \wedge \Upsilon(f_2) = \mathcal{V}_2 \wedge v \in \mathcal{V}_2 \rightarrow f_1 = f_2$ with $f_1, f_2 \in \mathcal{F}$
3. A version and its predecessor version have to belong to the same feature.
   $\forall v \in \mathcal{V} : \text{pred}(v) = v_p \wedge \Upsilon(f_1) = \mathcal{V}_1 \wedge v \in \mathcal{V}_1 \wedge \Upsilon(f_2) = \mathcal{V}_2 \wedge v_p \in \mathcal{V}_2 \rightarrow f_1 = f_2$

Using this formalization, the example displayed in Figure 4.2 may be represented as illustrated in Formula 2 and Formula 3.

Formula 2: Formalization of the TurtleBot HFM (1/2).

1. $\mathcal{F}$ =*(similar to Formula 1)*
2. $\mathcal{V}$ = {1.0 (TurtleBot), 1.1 (TurtleBot), 2.0 (TurtleBot), 2.1 (TurtleBot), 1.0 (Engine), 1.1 (Engine), Create 1.2 (Engine), Kobuki 1.0 (Engine), 1.0 (Movement), 1.1 (Movement), 1.2 (Movement), 2.0 (Movement), 1.0 (Keyboard), 1.0 (Gamepad), 2.0 (Gamepad), 1.0 (Autonomous), 1.1 (Autonomous), 2.0 (Autonomous), 1.0 (Webservice), 1.1 (Webservice), 1.0 (Detection), 1.1 (Detection), 1.0 (Bump), 1.0 (Infrared), 2.0 (Infrared), 2.2 (Infrared), 0.8 (Ultrasound), 0.9 (Ultrasound), 1.0 (Ultrasound)}
3. $\prec$: *(similar to Formula 1)*
4. λ : *(similar to Formula 1)*
5. Υ(TurtleBot) = {1.0 (TurtleBot), 1.1 (TurtleBot), 2.0 (TurtleBot), 2.1 (TurtleBot)}
   Υ(Engine) = {1.0 (Engine), 1.1 (Engine), Create 1.2 (Engine), Kobuki 1.0 (Engine)}
   Υ(Movement) = {1.0 (Movement), 1.1 (Movement), 1.2 (Movement), 2.0 (Movement)}
   Υ(Keyboard) = {1.0 (Keyboard)}
   Υ(Gamepad) = {1.0 (Gamepad), 2.0 (Gamepad)}
   Υ(Autonomous) = {1.0 (Autonomous), 1.1 (Autonomous), 2.0 (Autonomous)}
   Υ(Webservice) = {1.0 (Webservice), 1.1 (Webservice)}
   Υ(Detection) = {1.0 (Detection), 1.1 (Detection)}
   Υ(Bump) = {1.0 (Bump)}
   Υ(Infrared) = {1.0 (Infrared), 2.0 (Infrared), 2.2 (Infrared)}
   Υ(Ultrasound) = {0.8 (Ultrasound), 0.9 (Ultrasound), 1.0 (Ultrasound)}

To define the semantics of an HFM by the set of valid configurations $\mathcal{C} \subseteq \mathcal{F} \cup \mathcal{V}$, Definition 14 is extended in Definition 17.

Hence, a configuration of an HFM consists of features and associated feature versions. Figure 4.3 illustrates a valid HFM configuration in a graphical notation for HFMs.

The configuration of Figure 4.3 may further be represented in the formal notation as displayed in Formula 4.

4.4. Creation of Hyper-Feature Model Versions

A new version of a feature is created when its realization is changed in a meaningful way, e.g., by eliminating defects. However, these changes do not necessarily have to yield a new version when there neither are effects on possible combinations with versions of other features nor other (e.g., economical) reasons to model a new feature version.

The distinction of internal and external variability introduced in Section 2.1.2 is essential for the creation of new versions in an HFM. Whenever evolutionary changes affect external variability, a new version of a feature has to be created as the changes are, by definition, relevant for external stakeholders of configuring products of a software family. On the other hand, when changes only affect internal variability, it most likely is not necessary to propagate them as versions to the HFM, as they are probably not relevant to end-users. Exceptions may occur when the respective changes should be explicitly communicated to end-users, e.g., in the case

Formula 3: Formalization of the TurtleBot HFM (2/2).

6. pred(1.0 (TurtleBot)) = ε
pred(1.1 (TurtleBot)) = 1.0 (TurtleBot)
pred(2.0 (TurtleBot)) = 1.0 (TurtleBot)
pred(2.1 (TurtleBot)) = 2.0 (TurtleBot)
pred(1.0 (Engine)) = ε
pred(1.1 (Engine)) = 1.0 (Engine)
pred(Create 1.2 (Engine)) = 1.1 (Engine)
pred(Kobuki 1.0 (Engine)) = 1.1 (Engine)
pred(1.0 (Movement)) = ε
pred(1.1 (Movement)) = 1.0 (Movement)
pred(1.2 (Movement)) = 1.1 (Movement)
pred(2.0 (Movement)) = 1.2 (Movement)
pred(1.0 (Keyboard)) = ε
pred(1.0 (Gamepad)) = ε
pred(2.0 (Gamepad)) = 1.0 (Gamepad)
pred(1.0 (Autonomous)) = ε
pred(1.1 (Autonomous)) = 1.0 (Autonomous)
pred(2.0 (Autonomous)) = 1.1 (Autonomous)
pred(1.0 (Webservice)) = ε
pred(1.1 (Webservice)) = 1.0 (Webservice)
pred(1.0 (Detection)) = ε
pred(1.1 (Detection)) = 1.0 (Detection)
pred(1.0 (Bump)) = ε
pred(1.0 (Infrared)) = ε
pred(2.0 (Infrared)) = 1.0 (Infrared)
pred(2.2 (Infrared)) = 2.0 (Infrared)
pred(0.8 (Ultrasound)) = ε
pred(0.9 (Ultrasound)) = 0.8 (Ultrasound)
pred(1.0 (Ultrasound)) = 0.9 (Ultrasound)

Formula 4: Formalization of an example configuration for the TurtleBot HFM.

$\mathcal{C}_1 = \{\text{TurtleBot}, 2.0\ (\text{TurtleBot}),\ \text{Engine}, \text{Kobuki 1.0}\ (\text{Engine}),\ \text{Movement}, 1.1\ (\text{Movement}),$
$\text{Autonomous}, 1.1\ (\text{Autonomous}),\ \text{Detection}, 1.0\ (\text{Detection}),\ \text{Bump}, 1.0\ (\text{Bump})\}$

Definition 17: Hyper-Feature Model Semantics

Let $f_R \in \mathcal{F}$ be the root feature of an HFM. A configuration $\mathcal{C} \subseteq \mathcal{F} \cup \mathcal{V}$ of an HFM has to satisfy the following conditions to be valid:

1. The root feature is part of all configurations.
   $f_R \in \mathcal{C}$
2. Cardinality constraints of features and feature groups are respected by the configuration.
   $f_1 \in \mathcal{C} \wedge F = \{f_2 \in \mathcal{F} | f_1 \prec f_2\} \wedge \lambda(F) = (k, l) \Rightarrow k \leq |F \cap \mathcal{C}| \leq l$
3. For each selected feature, the parent feature has to be in the configuration.
   $f_1 \in \mathcal{C} \cap \mathcal{F} \wedge f_2 \in \mathcal{F} \wedge f_2 \prec f_1 \Rightarrow f_2 \in \mathcal{C}$
4. For each selected version, the containing feature has to be part of the configuration.
   $\forall v \in \mathcal{C} \cap \mathcal{V} : \Upsilon(f) = \mathcal{V}_f \wedge v \in \mathcal{V}_f \rightarrow f \in \mathcal{C}$
5. For each selected feature, there has to be exactly one version in the configuration.
   $\forall f \in \mathcal{C} \cap \mathcal{F} : \Upsilon(f) = \mathcal{V}_f \rightarrow \exists v \in \mathcal{V}_f : v \in \mathcal{C} \wedge (v_1, v_2 \in \mathcal{C} \cap \mathcal{V}_f \rightarrow v_1 = v_2)$

Let $\mathbb{H}$ be the set of all possible valid HFMs over $\mathcal{F} \cup \mathcal{V}$. The semantic evaluation function is defined as $[\![\cdot]\!] : \mathbb{H} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{F} \cup \mathcal{V}))$. The semantics of the HFM is defined as $[\![\text{HFM}]\!] = \mathbb{C}$ such that $\forall \mathcal{C} \in \mathbb{C} : \mathcal{C}$ is valid for HFM.

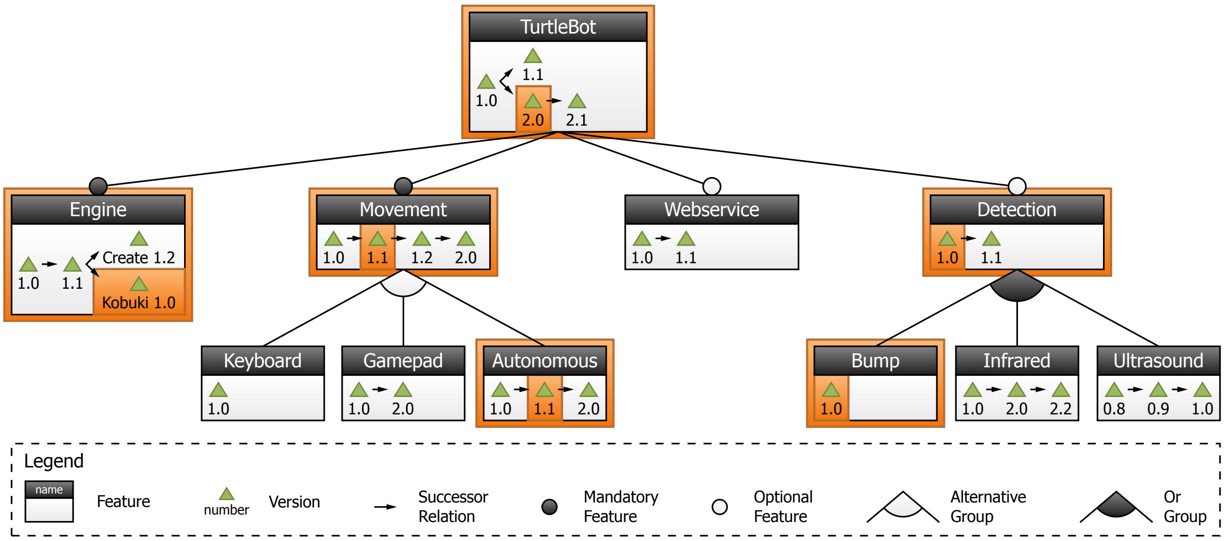


Figure 4.3.: Graphical example of an HFM configuration containing both features and adequate feature versions.

of a bug fix that inevitably has side effects such as altering intended functionality where an update to the new version requires a conscious user choice.

To reduce the effort of creating an HFM, it is further possible to semi-automatically gather information on implementation versions from a source code repository. For practical applications of HFMs, versions may further be encompassed by additional metadata, such as creators of the version or the date of creation. However, in the formal representation, versions are described solely by a textual name, as well as the relation to their containing feature and the respective predecessor version.

4.5. Version-Aware Constraints to Represent Version Dependencies and Incompatibilities

In addition to the configuration knowledge captured in the HFM, cross-tree constraints may be specified in order to express dependencies or incompatibilities of features and versions spanning across the HFM. To express these constraints, this section presents a language that can establish constraints over versions and version ranges, e.g., to express that certain versions are incompatible with versions of another feature, which addresses requirement `R1.4` of Section 4.2. To give a precise definition of the syntax of the version-aware constraint language, all possible constructs for a constraint of the version-aware constraint language are enumerated in Definition 18.

Definition 18: Version-Aware Constraint Language Syntax

Let $f \in \mathcal{F}$ be a feature, $v_a, v_b \in \mathcal{V}$ versions and op $\in \{>, \geq, =, \leq, <\}$ an operator over versions. The following are basic expressions of the version-aware constraint language:

1. f (feature presence)
2. $f\ [v_a - v_b]$ (version range restriction)
3. $f\ [\text{op}\ v_a]$ (relative version restriction)
4. $?f\ [v_a - v_b]$ (conditional version range restriction)
5. $?f\ [\text{op}\ v_a]$ (conditional relative version restriction)

Let ψ_1 and ψ_2 be version-aware constraints. The following are compound expressions of the version-aware constraint language:

6. ψ_1 (basic expression)
7. (ψ_1) (nested expression)
8. $\neg\psi_1$ (negation)
9. $\psi_1 \wedge \psi_2$ (conjunction)
10. $\psi_1 \vee \psi_2$ (disjunction)
11. $\psi_1 \rightarrow \psi_2$ (implication)
12. $\psi_1 \leftrightarrow \psi_2$ (equivalence)

The semantics of version-aware constraints is defined in terms of reducing the language's constructs to expressions of propositional logic with defined semantics in Definition 19.

Constructs 6–12 may be used to express cross-tree constraints using Boolean logic referencing features by the feature presence expression (construct 1). For example, it is possible

Definition 19: Version-Aware Constraint Language Semantics

Let $\text{pred}^*(v) : \mathcal{V} \rightharpoonup \mathcal{P}(\mathcal{V})$ be the transitive closure over $\text{pred}(v)$, $\text{succ}(v) = \{v_s \in \mathcal{V} | \text{pred}(v_s) = v\}$ an auxiliary function to determine successors of a version and $\text{succ}^*(v) : \mathcal{V} \rightharpoonup \mathcal{P}(\mathcal{V})$ the transitive closure over $\text{succ}(v)$. A version-aware constraint ψ is satisfied over a configuration $\mathcal{C}$, denoted $\mathcal{C} \models \psi$, if the following holds:

1. Feature presence is satisfied iff the respective feature is part of the configuration.
   a) $\mathcal{C} \models f$ if $f \in \mathcal{C}$
   b) $\mathcal{C} \not\models f$ if $f \notin \mathcal{C}$
2. A version range restriction is satisfied if there is a version for the constrained feature in the configuration that is within the specified range.
   $\mathcal{C} \models f\ [v_a - v_b]$ if $\mathcal{C} \models f \in \mathcal{C} \wedge v_a \in \text{pred}^*(v_b) \wedge \exists v \in \mathcal{C} \cap \mathcal{V} : \Upsilon(f) = \mathcal{V}_f \wedge v \in \mathcal{V}_f \wedge$ $(v = v_b \vee v \in \text{pred}^*(v_b) \setminus \text{pred}^*(v_a))$
3. A relative version range restriction is satisfied if there is a version for the constrained feature in the configuration that satisfies the respective operator.
   a) $\mathcal{C} \models f\ [> v_a]$ if $\mathcal{C} \models f \in \mathcal{C} \wedge \exists v \in \mathcal{C} \cap \mathcal{V} : (\Upsilon(f) = \mathcal{V}_f \wedge v \in \mathcal{V}_f \wedge v \in \mathit{succ}^*(v_a))$
   b) $\mathcal{C} \models f\ [\geq v_a]$ if $\mathcal{C} \models f\ [> v_a] \vee f\ [= v_a]$
   c) $\mathcal{C} \models f\ [= v_a]$ if $\mathcal{C} \models f \in \mathcal{C} \wedge \exists v \in \mathcal{C} \cap \mathcal{V} : (\Upsilon(f) = \mathcal{V}_f \wedge v \in \mathcal{V}_f \wedge v = v_a)$
   d) $\mathcal{C} \models f\ [\leq v_a]$ if $\mathcal{C} \models f\ [< v_a] \vee f\ [= v_a]$
   e) $\mathcal{C} \models f\ [< v_a]$ if $\mathcal{C} \models f \in \mathcal{C} \wedge \exists v \in \mathcal{C} \cap \mathcal{V} : (\Upsilon(f) = \mathcal{V}_f \wedge v \in \mathcal{V}_f \wedge v \in \mathit{pred}^*(v_a))$
4. A conditional version range restriction is satisfied if the version range restriction is satisfied, but is only evaluated if the constrained feature is present.
   $\mathcal{C} \models ?f\ [v_a - v_b]$ if $\mathcal{C} \models f \in \mathcal{C} \rightarrow f\ [v_a - v_b]$
5. A conditional relative version restriction is satisfied if the relative version restriction is satisfied, but is only evaluated if the constrained feature is present.
   $\mathcal{C} \models ?f\ [\text{op } v_a]$ if $\mathcal{C} \models f \in \mathcal{C} \rightarrow f\ [\text{op } v_a]$

6.-12. Semantics are those of propositional logic.

to specify that the selection of one feature demands the selection of another feature (e.g., Autonomous → Detection) or that a combination of features depends on the presence of another feature (e.g., Keyboard ∨ Gamepad → Webservice).

In addition, version restrictions (constructs 2 and 3) respect the notion of variability in time present in HFMs by allowing the definition of constraints over sets of feature versions.

As first type of version restriction, version range restrictions specify a range of possible versions that satisfy the restriction by providing a lower and an upper version bound. For example, TurtleBot $[1.0 - 1.1]$ is satisfied for a version $v \in \mathcal{C} \cap$ {TurtleBot 1.0, TurtleBot 1.1}. This type of restriction is useful if both the upper and lower version bound of the range are defined within the HFM. This constitutes a fixed list of versions that is usually unaffected by future evolution of the features, as new versions of a feature generally are appended as new successors to the most recent versions. An exception to this rule exists, if a new intermediate version for a feature is introduced, which may happen, e.g., if recreating the evolution history of an existing software family. Version range restrictions can be used to express the type of version dependencies specified by manifest files of OSGi[5] bundles such as the plug-ins in the SECO around the Eclipse platform.

As second type of version restriction, relative version restrictions define a set of potential versions by means of a restriction in relation to one specific referenced version using an operator op $\in \{>, \geq, =, \leq, <\}$. For example, TurtleBot [> 1.0] is satisfied for a version $v \in \mathcal{C} \cap$ {TurtleBot 1.1, TurtleBot 2.0, TurtleBot 2.1}. Unlike version range restrictions, relative version restrictions specify an open range of versions satisfying a restriction (with the exception of the operator =). Thus, at the time of defining a relative version restriction, it is not necessary to be aware of all elements within the set of versions satisfying the restriction. For example, it is possible to specify that a selected version should be newer than the referenced version because a depending feature version is incompatible with older versions of the feature in question. Newer versions of the referenced feature would then automatically be valid options satisfying the version-restricted feature reference when they are added to the feature model due to evolution. For example, if a constraint TurtleBot [$\geq$ 2.0] $\rightarrow$ Engine [$\geq$ Kobuki 1.0] was specified and a new version *Kobuki 1.1* superseding *Kobuki 1.0* of the feature `Engine` was added to the feature model, then the constraint would still evaluate to `true` if the (previously unknown) version *Kobuki 1.1* of the `Engine` was selected as part of a configuration. Even though a similar restriction on versions can be specified with the syntax of the previously mentioned OSGi bundles as well, this notation offers no dedicated language construct to express explicitly open version ranges. However, it seems beneficial to capture this intent within a separate construct.

Conditional version range restrictions (construct 4) and conditional relative version restrictions (construct 5) are convenience constructs of the version-aware constraint language to avoid accidentally creating false optional features. For example, TurtleBot [$\geq$ 2.0] $\rightarrow$ Webservice [$\geq$ 1.1] would (possibly unintentionally) make the feature `Webservice` mandatory for all configurations containing feature `TurtleBot` in at least version *2.0*. To preserve the optional variability type of the feature, conditional version restrictions may be used as they are only evaluated if the constrained feature is part of the configuration. Hence, the previous constraint may be reformulated as TurtleBot [$\geq$ 2.0] $\rightarrow ?$Webservice [$\geq$ 1.1] to only be evaluated if `Webservice` is part of the configuration.

With the version-aware constraint language, the semantics of an HFM with version-aware constraints is defined in terms of valid configurations $\mathcal{C} \subseteq \mathcal{F} \cup \mathcal{V}$ according to Definition 20.

Definition 20: Hyper-Feature Model with Version-Aware Constraints Semantics

Let Ψ be a set of version-aware constraints with $\Psi \supseteq \Phi$. A configuration $\mathcal{C} \subseteq \mathcal{F} \cup \mathcal{V}$ of a Hyper-Feature Model with version-aware constraints (HFM, Ψ) has to satisfy the following conditions to be valid:

1. The configuration $\mathcal{C}$ has to be valid for the HFM as defined in Definition 17:
   $\mathcal{C} \models \mathrm{HFM}$
2. The configuration has to satisfy all version-aware constraints as defined in Definition 19:
   $\mathcal{C} \models \bigwedge_{\psi_i \in \Psi} \psi_i$

Let $\mathbb{H}$ be the set of all possible valid HFMs over $\mathcal{F} \cup \mathcal{V}$ and $\mathbb{A}$ the set of all possible expressions of the version-aware constraint language over $\mathcal{F} \cup \mathcal{V}$. The semantic evaluation function is defined as $[\![\cdot]\!] : (\mathcal{P}(\mathbb{H}), \mathcal{P}(\mathbb{A})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{F} \cup \mathcal{V}))$. The semantics of the HFM with version-aware constraints is defined as $[\![(\mathrm{HFM}, \Psi)]\!] = \mathbb{C}$ such that $\forall \mathcal{C} \in \mathbb{C} : \mathcal{C}$ is valid for HFM and $\mathcal{C} \models \Psi$.

[5] `http://osgi.org`

By using the version-aware constraint language, it is possible to formulate the interdependencies and incompatibilities of different ranges of versions in the TurtleBot example. Using a combination of features with and without version restrictions within a single constraint allows specification of interdependencies that cross space and time. Formula 5 presents all version-aware constraints and Section 4.7 further elaborates on reasons for these constraints.

Formula 5: Version-aware constraints of the HFM in Figure 4.2.

(1) Autonomous → Detection
(2) Keyboard $\vee$ Gamepad → Webservice
(3) Infrared [$\geq$ 2.0] $\vee$ Ultrasound → Detection [$\geq$ 1.1]
(4) TurtleBot [$\geq$ 2.0] → Engine [$\geq$ Kobuki 1.0]
(5) TurtleBot [1.0 – 1.1] → Engine [$\leq$ Create 1.2]
(6) TurtleBot [$\geq$ 2.0] →?Webservice [$\geq$ 1.1]

For the state of an HFM at one particular point in time, it is possible to reduce all version-aware constraints to propositional logic by resolving all version restrictions to a logical or expression of the enumerated versions they address at that particular time. For example, the constraints of Formula 5 can be reduced to propositional logic for the state of the HFM in Figure 4.2 as illustrated in Formula 6.

Formula 6: Reduction of the version-aware constraints of the HFM in Figure 4.2 to propositional logic for the current state of the HFM.

1. Autonomous → Detection
2. Keyboard $\vee$ Gamepad → Webservice
3. Infrared [$\geq$ 2.0] $\vee$ Ultrasound → Detection [$\geq$ 1.1] $\equiv$
   (2.0 (Infrared) $\vee$ 2.2 (Infrared)) $\vee$ Ultrasound → 1.1 (Detection)
4. TurtleBot [$\geq$ 2.0] → Engine [$\geq$ Kobuki 1.0] $\equiv$
   (2.0 (TurtleBot) $\vee$ 2.1 (TurtleBot)) → Kobuki 1.0 (Engine)
5. TurtleBot [1.0 – 1.1] → Engine [$\leq$ Create 1.2] $\equiv$
   (1.0 (TurtleBot) $\vee$ 1.1 (TurtleBot)) → (1.0 (Engine) $\vee$ 1.1 (Engine) $\vee$ Create 1.2 (Engine))
6. TurtleBot [$\geq$ 2.0] →?Webservice [$\geq$ 1.1] $\equiv$
   (2.0 (TurtleBot) $\vee$ 2.1 (TurtleBot)) → (Webservice → 1.1 (Webservice))

However, this transformation is only valid for the constellation of versions *at the current state of the HFM*. If new versions were added, the formulas in propositional logic would (possibly) be invalidated as they are based on the previous state of the HFM. In consequence, the version-aware constraints would have to be re-evaluated to find appropriate versions matching the version restrictions. Hence, even though writing constraints for HFMs in mere propositional logic would be possible, it should be avoided: First, the representation in propositional logic tends to get overly verbose through explicit enumeration of all possible versions matching a version-restriction and, thus, lacks readability. Second, the explicit enumeration of versions as logical disjunction clauses does not capture the original intent of specifying an interval of matching versions. Third, the explicitly enumerated ranges of versions in the propositional constraints would have to be adapted when new versions are added to the HFM, which is a source of errors during evolution.

The presented version-aware constraint language avoids these problems and, thus, should be preferred over mere propositional logic when specifying constraints on HFMs.

4.6. Hyper-Feature Models are a True Extension to Feature Models

At first glance, it seems that common feature models [KCH$^+$90] and their extensions may be used to emulate versions of features as configuration units similarly to the representation in HFMs. Particularly, dummy features of common feature models and special attributes of attributed feature models [CHE05] may seemingly serve this purpose. Figure 4.4 illustrates a feature with versions as a) HFM, b) dummy features in common feature models and c) attributes in an attributed feature model.

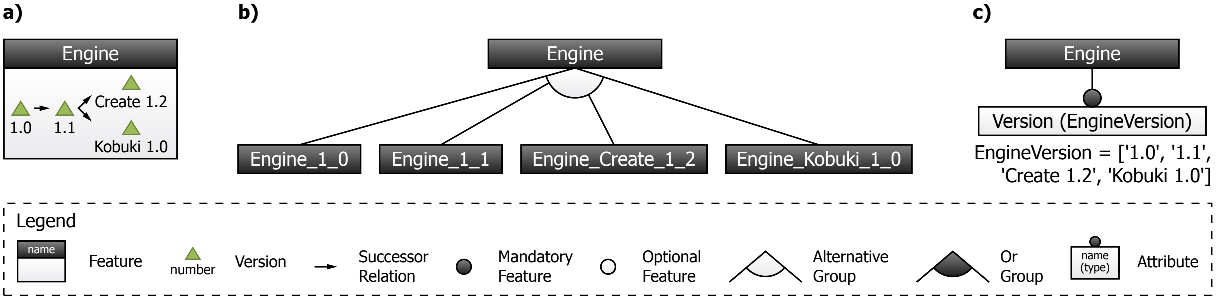


Figure 4.4.: Different notations to express features and feature versions: a) HFMs, b) dummy features of common feature models, c) attribute of attributed feature models.

In common feature models [KCH$^+$90], it is possible to introduce dummy features with version numbers as children of the feature in question, e.g., features `Engine_1_0`, `Engine_1_1` etc. as children of `Engine` as represented in Figure 4.4 b). The alternative group ensures that exactly one "version" is selected when the feature is selected. However, modeling versions as features does not adequately reflect the intent of variability in time as it mixes two different concerns in trying to express two different concepts with the same language construct. Furthermore, features in a group have no particular order so that the successor/predecessor relation of versions is lost when encoding versions in common feature models. As a consequence, neither development lines nor branching can be expressed. In addition, the lack of a relation between versions greatly complicates expressing constraints over versions. For example, expressing that a version of one feature requires at least a particular version of another feature (or any more recent version on the same development line) can only be achieved by explicitly enumerating all possible choices in a disjunctive clause. This is tedious when a great number of versions exists. Furthermore, it causes problems when new versions are added as result of evolution, as the respective constraints may have to be updated manually, when the new versions are supposed to satisfy the constraint. This procedure is both error prone and tedious. Due to the limitations in expressiveness and the severe inconveniences with regard to expressing dependencies and incompatibilities, modeling feature versions as dummy features of common feature models is no equivalent alternative to HFMs.

As a second option, it may seemingly be possible to use attributed feature models [CHE05] to reflect versions, e.g., as represented in Figure 4.4 c). For this purpose, each feature is extended by a special attribute representing its respective version. The domain of possible values for the attribute contains all existing version numbers of that feature. When the values are perceived as an ordinal type, there may even be a (linear) order between the versions of a feature. However, this approach again does not capture the successor/predecessor relation and, thus, does not support branching of versions, as it cannot express tree structures. Introducing a tree structured data type as domain of the attribute might address this problem, but is not done in practice. In comparison to using dummy features, it is easier to express constraints on versions and version ranges when using attributes, because binary relations, such as “greater than”, may be expressed without enumerating all possible satisfying values. In either case, modeling feature versions as attributes does not represent a clean separation of concerns as a single language construct is used to represent aspects of both variability in space and time.

Due to the limitations of common feature models and attributed feature models with respect to adequately representing versions of features and constraints over these versions, both approaches cannot be considered alternatives to HFMs for capturing variability in space and time. Hence, HFMs are a true extension to feature models with the intent of capturing feature versions as configurable units. In combination with the version-aware constraint language, it is further possible to express dependencies and incompatibilities of versions and version ranges.

4.7. Case Study

To demonstrate the feasibility of modeling variability in space and time with HFMs, a case study was performed modeling the driver software for the TurtleBot domestic robot. As explained in Chapter 1, the driver software consists of configurable functionality and multiple different versions of that functionality. For the evaluation, a preliminary version of the tool suite explained in Chapter 7 as extension of Eclipse[6] was used. The concepts were implemented as a metamodel based on EMF Ecore (see Section 3.1) for HFMs and a metamodel aided with a concrete syntax in EMFText (see Section 3.1) for the version-aware constraint language.

The driver software was developed over the period of approximately 1.5 years with a variety of evolutionary changes to the Java source code creating multiple versions of its configurable units. The evolution of the driver was reconstructed by evaluating the data of its source code repository and semi-automatically creating versions in a manually created initial HFM.

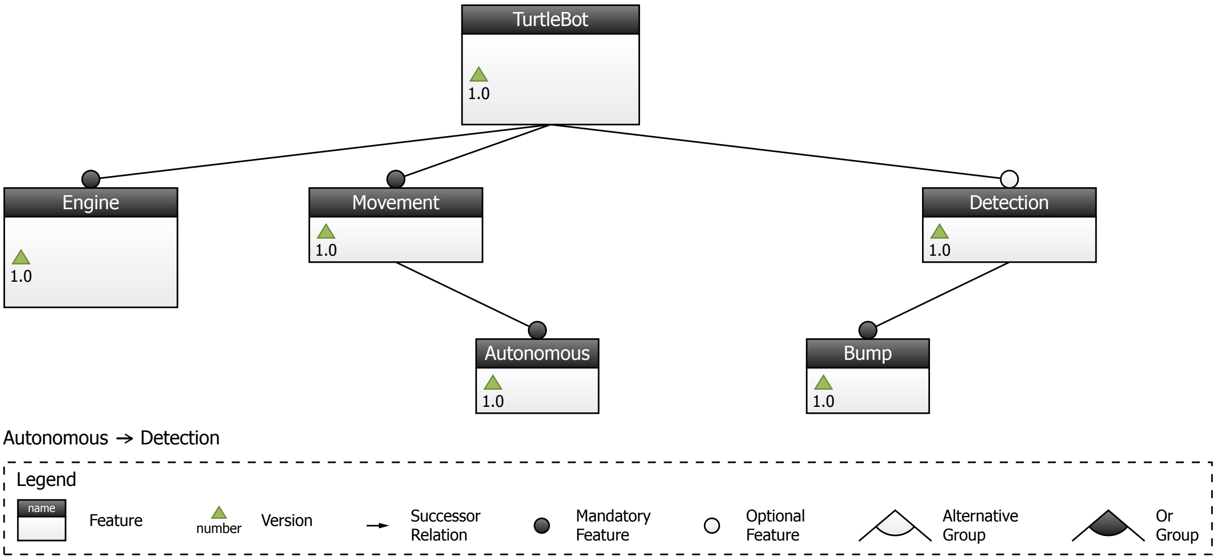


Figure 4.5.: HFM representing the original variability of the TurtleBot driver software.

[6] http://eclipse.org

In the original version of the driver, the software family consisted of six features each with the respective initial version: TurtleBot, Engine, Movement, Autonomous, Detection and Bump as represented by Figure 4.5. The evolutions performed on the driver software can roughly be grouped into four stages:

S1 Features Webservice, Keyboard and Infrared were added with their respective initial versions (see Figure 4.6).

S2 Features Ultrasound and Gamepad were added with their initial versions. Furthermore, new versions were created for features Engine, Movement, Autonomous, Webservice, Detection and Infrared representing improved implementations as well as fixes for defects (see Figure 4.7).

S3 The new version *2.0* of the feature TurtleBot was introduced entailing the new version *Kobuki 1.0* of Engine, as well as a new version *0.9* of the Ultrasound sensor for reasons of compatibility with the new TurtleBot implementation. Furthermore, new versions were created for Movement and Gamepad (see Figure 4.8).

S4 Updates to the TurtleBot were performed by creating versions *1.1* and *2.1* in two separate branches. In addition, the Engine was updated on the 1.x branch of the old hardware platform creating version *Create 1.2*. Finally, the most recent versions were created for Movement, Autonomous, Infrared and Ultrasound (see Figure 4.9).

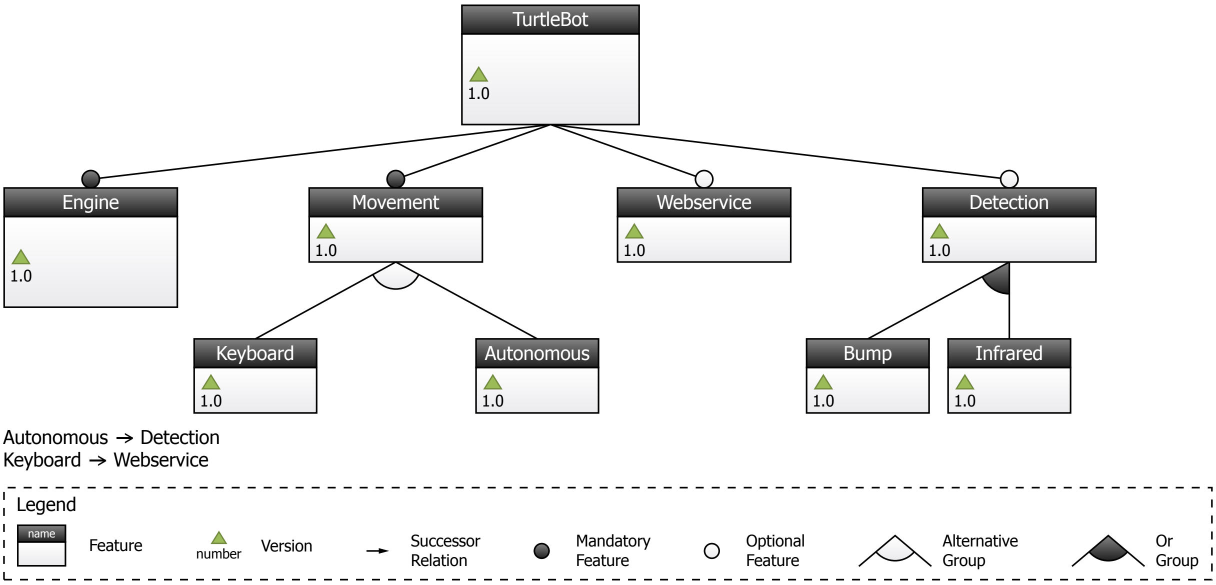


Figure 4.6.: HFM representing variability after the first stage of evolution of the TurtleBot driver software.

In the original version of the driver, there was a single constraint stating that autonomous operation of the robot requires an obstacle detection mechanism to be present (see Formula 5 (1)). To limit configuration options as well as to express dependencies and incompatibilities between different versions introduced as part of evolution, further constraints were formulated using the version-aware constraint language of Section 4.5.

S1 The constraint Keyboard → Webservice was defined to capture that keyboard control depends on remote communication with the robot via WiFi.

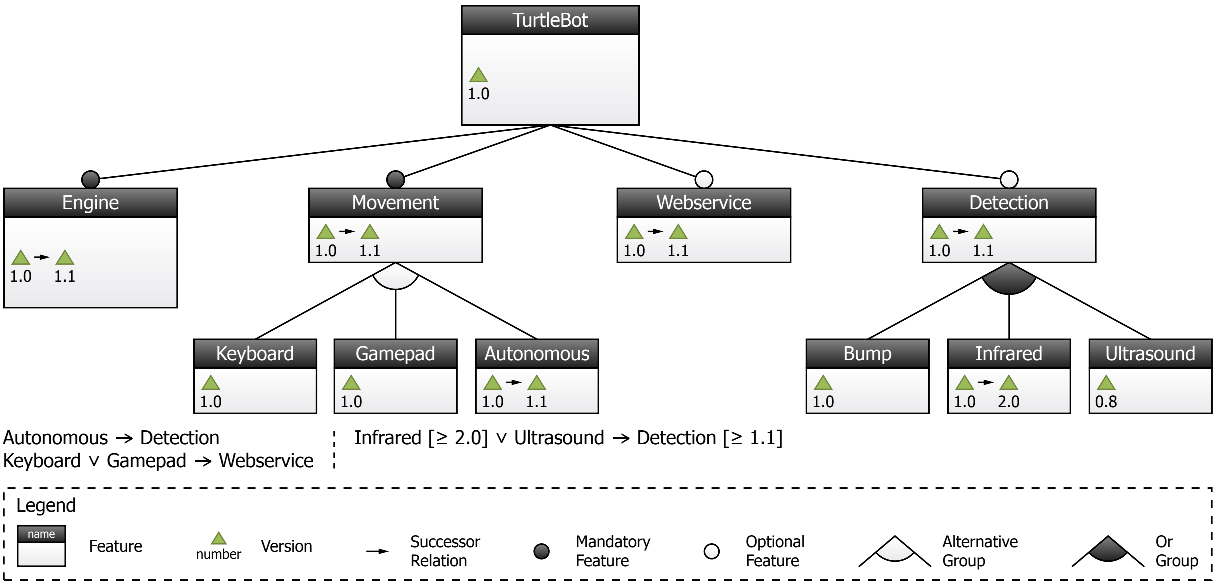


Figure 4.7.: HFM representing variability after the second stage of evolution of the TurtleBot driver software.

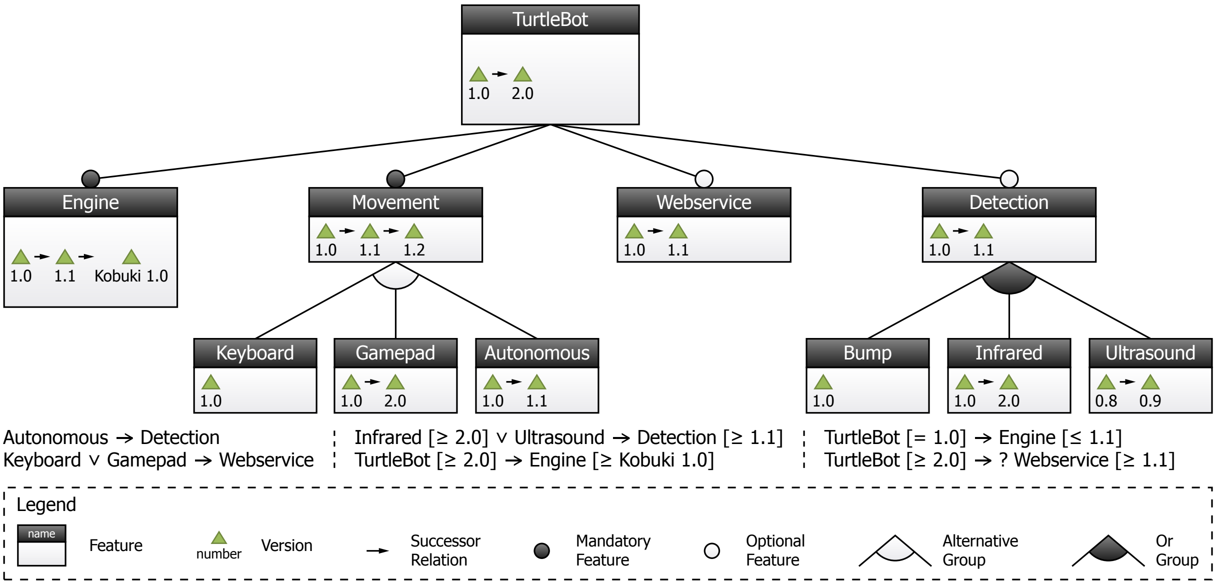


Figure 4.8.: HFM representing variability after the third stage of evolution of the TurtleBot driver software.

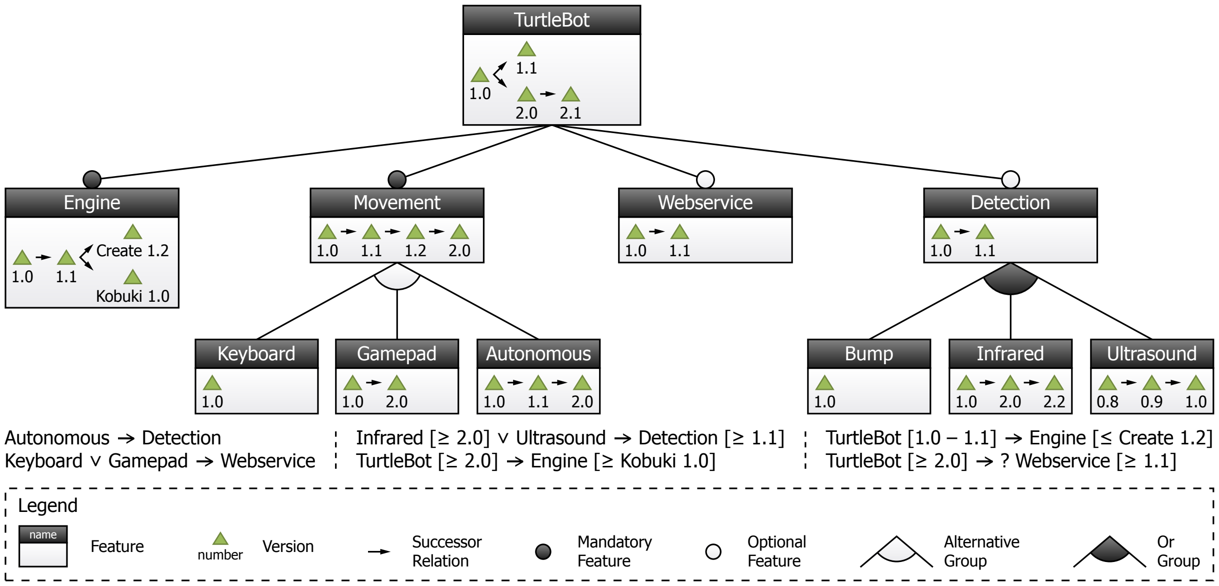


Figure 4.9.: HFM representing variability after the fourth stage of evolution of the TurtleBot driver software.

S2 The constraint Keyboard → Webservice was extended to also include the new control via gamepad, see Formula 5 (2). Furthermore, the constraint in Formula 5 (3) was added to express dependence of the infrared and ultrasound sensors on the newly created version of the obstacle detection mechanism.

S3 The constraint in Formula 5 (4) and the constraint TurtleBot [= 1.0] → Engine [$\leq$ 1.1] were added to express that the new `TurtleBot` depends on the Kobuki engine. Furthermore, the constraint in Formula 5 (6) was added to state that, as of version *2.0* of the `TurtleBot`, the `Webservice` should have version *1.1* or above if it is selected as part of a configuration.

S4 The constraint in Formula 5 (5) was created to refine the previous constraint defining dependence of the old revision of the `TurtleBot` on compatible revisions of the `Engine`.

Using HFMs presented in this chapter, it was possible to capture all different versions of features of the driver software as configurable units on a conceptual level, which satisfies requirements `R1.1` and `R1.2`. Logical development lines and branching could be captured using the successor/predecessor relation, which satisfies requirement `R1.3`. Dependencies of feature versions on ranges of other versions could be expressed employing the version-aware constraint language, which satisfies requirement `R1.4`. Evolutionary changes cutting across multiple features, such as API changes, could be represented by individual versions of the affected features and version-aware constraints. Finally, the explicit intention of variability in time is captured in HFMs by a dedicated version concept, which satisfies requirement `R1.5`.

4.8. Demarcation from Related Work

The work related to HFMs dealing with variability in time in SPLs and SECOs is discussed with regard to the requirements of Section 4.1, in order to demarcate HFMs from the respective approaches. Table 4.1 contrasts HFMs with all discussed related approaches.

Ducasse et al. [DGF05] model software evolution by treating history as first class entity. They capture a snapshot of a realization at a particular moment in time as a version. A history element represents knowledge about evolution of these versions. The resulting History Metamodel (Hismo) is used primarily for analyses determining certain evolution patterns regarding the type of evolution, such as a *Pulsar* with alternating many or few changes per version or a *Supernova* with increasingly more changes per version.

With the Hismo metamodel, the authors capture versions on a conceptual level independently of the underlying concrete realization artifact, which fulfills `R1.1`. Even though their metamodel acknowledges the importance of versions as first-class entities, there is no intent of using them within a configuration, leaving `R1.2` unsatisfied. Development lines can be expressed with full support for branching due to explicit predecessor and successor relations, which satisfies `R1.3`. However, there is no mechanism to express interdependencies of versions due to their intended focus, which leads to a negative result regarding `R1.4`. Finally, the intention of variability in time is clearly represented in the metamodel due to the distinction of `StructuralEntity` and `Version` elements, which fulfills `R1.5`.

Seidl and Aßmann [SA13] focus on capturing evolution of SECOs for analysis, a work prior to this thesis. For this purpose, the Technical Ecosystem Modeling (Tecmo) notation is introduced in the form of a metamodel. Respective models may capture `Products` consisting of arbitrary `Artifacts`, where either of these `EcosystemElements` may further be split up into an arbitrary number of versions. First-class entities for dependencies and logical development

	R1.1 Conceptual Level	R1.2 Versions as Configurable Units	R1.3 Development Lines with Branching	R1.4 Version Interdependencies	R1.5 Intention of Variabiltiy in Time
Variability Model for Variability in Time	+	+	+	+	+
History Metamodel (Hismo) [DGF05]	+	-	+	-	+
Technical Ecosystem Modeling (Tecmo) [SA13]	+	-	+	o (Only Direct)	+
Feature-Driven Versioning [ME08]	+	-	o (No Branching)	-	+
Change-Oriented Programming (ChOP) [EVC+07]	-	o (Implicit as Change Sets)	o (Change Set Interdependencies)	o (Change Set Interdependencies)	+
Version-Management Tools [vGP06]	-	+	+	-	+
Common Feature Models [KCH+90]	+	o (As Dummy Features)	-	o (Using Regular Constraints & Enumeration)	-
Attributed Feature Models [CHE05]	+	+	o (Implicit, No Branching)	o (Using Regular Constraints)	-
Common Variability Language (CVL) [HMPO+08]	+	+	o (Implicit, No Branching)	o (Using Regular Constraints)	-

Table 4.1.: Comparison of the variability model for variability in time of Chapter 4 with related approaches using the requirements of Section 4.1.

lines are used to represent relations of versions. Based on the information within the respective models gathered from a SECO, such as the one surrounding Eclipse, different types of analyses may be performed. In particular, the status of a SECO, differences of two versions of the same product within a SECO, as well as trends within the progression of changes to variability within a SECO may be performed. Each of the analyses may be performed for the entire evolution history or previously selected temporal perspectives representing only a part of it.

The Tecmo metamodel features dedicated elements for both product and artifact versions to lift versions to a conceptual level, which satisfies `R1.1`. However, the information in the respective models is used for the sole purpose of analysis, but not for configuration, so that `R1.2` is not fulfilled. The relation of individual versions is captured with a dedicated `Supersedes` relation that may also represent branching, which fulfills `R1.3`. Dependencies of versions can be captured within a dedicated `Dependency` class. However, the structure of the metamodel only permits representing direct dependencies of single versions so that more complex dependencies cannot be expressed and `R1.4` is partially unsatisfied. Finally, the intention of variability in time is clearly visible within Tecmo models due to the focus on SECO evolution of the respective metamodel, which fulfills `R1.5`.

Mitschke and Eichberg [ME08] introduce a versioning scheme for feature models of SPLs they call *feature-driven versioning*. Each feature has a *feature logical version* capturing changes to the structure of the feature model underneath the feature and a *feature container version* capturing changes to the realization of a feature in the solution space. Even though feature container versions seem similar to feature versions in HFMs, there is only exactly one feature container version for each feature at any given point in time. Hence, the approach puts the feature model under version control, but does not support multiple versions of a feature within the feature model for configuration of products.

With regard to the initially posed requirements, this leads to the following conclusions: Through the explicit notion in the feature model, versions of both the feature model and the associated realization are lifted to the conceptual level, which satisfies `R1.1`. However, there is only exactly one of each of these versions available at any given time, so that they cannot be used for configuration, which leaves `R1.2` unsatisfied. Development lines represented in the approach are implicit and, hence, only allow for linear relations, but no branching, so that `R1.3` is fulfilled only partially. Furthermore, there are no means to specify interdependencies of the existing versions so that `R1.4` is unsatisfied. However, the explicit focus on change due to evolution in the notation, the explicit intent of variability in time is represented adequately, which satisfies `R1.5`.

Ebraert et al. [EVC^{+}07, EMD08, ESJ11] define Change-Oriented Programming (ChOP), which uses *change objects* to represent evolutionary modifications on realization assets in order to capture variability in time. Multiple change objects are grouped to form change sets. Information on the interrelation of the change sets is used to generate a feature model to also address variability in space [ECHD09]. Furthermore, **Hendrickson and van der Hoek** [HvdH07] use change sets on software architectures and map them to features using explicit relationships within the change sets.

Feature models generated from change sets are very much aligned with fine-grained realization concerns, but not with conceptually coherent units of functionality and versions are only specified as transformation instructions, so that `R1.1` is not satisfied. Change sets do not necessarily correspond to individual versions, but may be used to structure modifications to configurable units by convention, which partially satisfies `R1.2`. Development lines may only be captured by specifying correspondent dependencies within change sets, but not by a dedicated language construct, which only partially satisfies `R1.3`. The same mechanism may be used to specify interdependencies between change modules representing individual versions, but not between version ranges, so that R1.4 is only partially satisfied. As evolution is in focus of ChOP, the intention of variability in time is adequately captured and R1.5 is satisfied.

Van Gurp and Prehofer [vGP06] argue that tools for variability management and product derivation should be combined in order to cope with evolving variable assets. In particular, they suggest adding properties representing information from the variability model as configuration knowledge to artifacts under version control. Thus, their approach uses a procedure reverse to the one presented in this chapter by transferring configuration knowledge from the problem space to the solution space instead of introducing a versioning notation in the problem space.

Due to the integration of version information with realization assets, version information for features is not made explicit on a conceptual level, so that R1.1 is unsatisfied. By changing the active version in the tools, it is possible to perceive versions as configurable units, which satisfies R1.2. The support for branching is inherent in version control systems, so that R1.3 is satisfied. Version interdependencies and especially incompatibilities may not adequately be represented on a level more conceptual than in realization assets so that R1.4 is unsatisfied. With versions being a first-class entity in version control systems, R1.5 is satisfied.

As discussed in Section 4.6, neither common feature models nor attributed feature models may be used as adequate substitute for HFMs when representing variability in time. Nevertheless, both these approaches overlap with HFMs in many areas so that they are discussed again as related work explicitly elaborating on how they satisfy the requirements posed in Section 4.1.

Kang et al. [KCH$^+$90] introduce (common) *feature models* to capture configuration options on a conceptual level which satisfies R1.1. However, versions can only be represented as dummy features, which leaves R1.2 partially unsatisfied. A relation between the dummy features to capture development lines cannot be expressed, as there is no defined order on features and branching is not possible, so that R1.3 is not fulfilled. Dependencies between the dummy features can be expressed using regular constraints, but requirements on ranges of "versions" can only be specified by explicitly enumerating all possible matching dummy features, which only partially fulfills R1.4. Finally, the intention of variability in time is not captured with dummy features, so that R1.5 is unsatisfied.

Czarnecki et al. [CHE05] introduce *attributed feature models*, which may represent versions as special attributes of the feature model on a conceptual level, so that R1.1 is satisfied. By assigning a concrete value for an attribute during configuration, versions are available as configurable units, which satisfied R1.2. The relation of versions on development lines may be captured implicitly (assuming a linear order on versions in an enum type), which cannot capture branching and, thus, leaves R1.3 partially unsatisfied. Dependencies on versions may be specified using regular constraints on attributes without taking the relation of versions into account, so that R1.4 is satisfied only partially. The intention of variability in time is not adequately captured within attributed feature models, as they provide no dedicated language construct, which leaves R1.5 unsatisfied.

Haugen et al. [HMPO$^+$08] introduce the Common Variability Language (CVL) with the intent of adding standardized variability to arbitrary base models by overlaying variability information over realization assets of a software family. Configuration knowledge is captured on a conceptual level in Variability Specifications (VSpecs) (see Section 2.2.1). Conceptual configurations may be defined from a VSpec and used with variant derivation facilities to realize changes associated with variability in space. CVL contains language constructs for both features and properties (resembling attributes), so that it has similar qualities with regard to the posed requirements as attributed feature models.

Feature versions may be modeled on a conceptual level by defining special attributes to features, which satisfies R1.1. Hence, versions may be used as configurable units, so that R1.2 is fulfilled.

Development lines may only be represented by imposing an implicit order on the enumerated values of an attribute's value domain, so that R1.3 is satisfied only partially. Interdependencies of versions may only be expressed using regular constraints, but not with dedicated language constructs, leaving R1.4 partially unsatisfied. Using these constructs for variability in space to represent variability in time does not adequately address R1.5.

In addition to the approaches discussed above, management of versions may be addressed by tools such as Concurrent Versions System (CVS)[7], SVN[8] or Git[9]. Additionally, configurations of artifacts in different versions with interdependencies can be managed with repositories such as the OSGi Bundle Repository (OBR)[10], repositories available for Maven[11] or the Oscar Bundle Repository[12]. However, these solutions are tightly integrated with the respective realization assets and are not available on a conceptual level.

4.9. Chapter Summary

In this chapter, the need to capture information regarding variability in time (evolution) besides variability in space (configuration) within a variability model of the problem space was identified. For this purpose, Hyper-Feature Models (HFMs) were presented with explicit elements for feature versions and a predecessor relation among versions to express branching. In addition, a version-aware constraint language was introduced to express cross-tree constraints on versions and version ranges, e.g., to express incompatibility of one version of a feature with versions of another feature. The feasibility of the conceptual modeling was demonstrated by applying it in a case study modeling aspects of variability in space and time of the configurable driver software for the TurtleBot robot from Chapter 1. Figure 4.10 illustrates the contributions of Chapter 4 in context of the thesis.

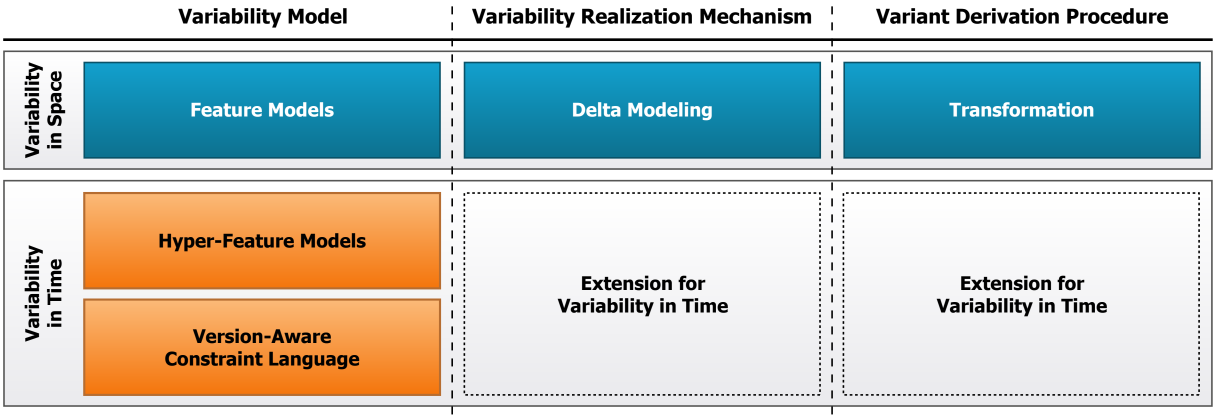

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Hyper-Feature Models
Version-Aware Constraint Language
Extension for Variability in Time
Extension for Variability in Time

Figure 4.10.: Contributions of Chapter 4 in context of the thesis.

[7] http://cvs.nongnu.org
[8] http://subversion.apache.org
[9] http://git-scm.com
[10] http://osgi.org/Repository
[11] http://maven.apache.org
[12] http://sourceforge.net/projects/oscar-osgi

5. Creating Delta Languages Suitable for Variability in Space and Time

The contents of this chapter are largely based on the work published in [SSA14b, SSA14c, SSA13].

Summary *Due to the use of transformations, delta modeling is principally suitable to perform changes associated with variability in space and time, but there currently are no applications in this area. To cope with a wide variety of languages of realization assets in a software family, custom delta languages are required for all source languages, which are tedious to create and lack interoperability due to different implementation technologies. In this chapter, remedy to these problems is provided by first extending the notion of delta modules to provide delta operations suitable for variability in time. Furthermore, approaches are presented to automatically derive delta languages for textual or graphical languages in the form of EMOF-based metamodels. Using these contributions, it is possible to automatically generate the syntax and large parts of the semantics of custom delta languages to handle variability in space and time by inspecting source languages' metamodels.*

5.1. Current Delta Languages are not Suitable for Variability in Time

Due to the beneficial qualities regarding evolution and an open variant space described in Section 3.2.2, delta modeling is used to apply changes to the realization assets of a software family required by a certain configuration. For this purpose, custom delta languages are employed to specify delta modules realizing changes associated with features and feature versions. However, in the current state, delta modeling is not fully suitable for the intended application due to multiple reasons, as visualized in Figure 5.1:

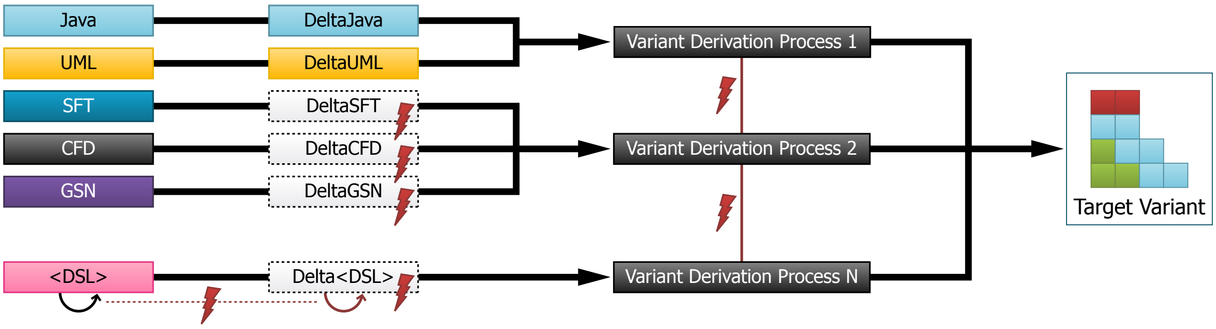


Figure 5.1.: Current creation and application of delta languages is tedious due to the multitude of different source languages with graphical/textual representations that require a delta language. Furthermore, technical incompatibility of existing delta languages and their inability to handle evolution hinder application.

With multiple different languages specifying a software family (e.g., design models, source code etc.), a variability realization mechanism needs to be applicable to all languages whose artifacts are affected by different configurations. Furthermore, if these artifacts are subject to variability in time, the changes associated with evolution need to be handled as well. Due to the interdependency of variability in space and time regarding configuration options, both dimensions may not be separated completely in some cases. Changes associated with variability in space commonly consist of adding, modifying and removing certain elements of an artifact in the source language. However, the changes associated with variability in time may be more complex and extensive in nature, as they perform (possibly) unpredicted evolution affecting arbitrary parts of the system. In consequence, delta languages used for the purposes of variability in time need to be more expressive than common delta languages. Nevertheless, variability in space should only be performed using the respective operations in order to reduce the probability for harming system integrity. However, integrating variability in time into delta languages to support evolution has not been investigated.

Furthermore, delta modeling is only applicable, if all languages of the software family support it through a dedicated delta language. However, due to the diverse types and representations of different artifacts in a software family, a wide variety of delta languages is required. This is challenging: First, many languages, in particular domain-specific languages, do not have a pre-defined delta language. Second, source languages may be textual and graphical in nature. Third, new languages may be introduced and existing ones may be altered as part of evolution, which requires adaptation of the respective delta language as well. Creating a delta language manually for a specific source language is tedious, as not only the language's syntax and semantics have to be devised but also the tools to create delta modules needs to be created. Hence, manual creation and maintenance of delta languages requires extensive efforts.

Even though implementations of delta languages exist for source languages such as Java [SBB+10, KHS+14], Class Diagrams [Sch10], Matlab/Simulink [HKM+13] or Component Fault Diagrams (CFDs) [SSA13], they are currently incompatible with one another due to different implementation technologies. Due to this lack of interoperability, multiple tools are required to handle variability of different source languages in a software family. In addition, with different creators of these languages, the syntax of delta languages may differ, which makes it hard for users to specify delta modules for different source languages.

Based on these problems, R2 of Section 3.4.2 is refined to the following requirements for a variant derivation mechanism based on delta modeling that is capable of coping with variability in space and time:

R2.1 **Handle Variability in Space**: Represent changes of solution space assets due to configuration associated with individual features.

R2.2 **Handle Variability in Time**: Represent changes of solution space assets due to evolution associated with individual feature versions.

R2.3 **Textual/Graphical Source Languages**: Handle source languages that utilize various different representations.

R2.4 **Define Syntax/Semantics of Operations**: Define standard syntax and semantics for common modification operations.

R2.5 **Automatically Create Standard Operations**: Derive common modification operations automatically from a source language.

Generating delta languages on basis of a common framework may address these requirements. However, existing approaches [HHK+13] are limited to deriving the syntax of delta languages for textual languages from grammars and can neither generate an interpreter for their semantics nor tools for product derivation. The following sections introduce concepts and a framework meeting above requirements to create custom delta languages for source languages given as EMOF-based metamodels (see Section 3.1).

5.2. Software Fault Trees as Example of a Source Language

To illustrate the concepts in this chapter, the graphical language of Software Fault Trees (SFTs) [Lev95] is used as example. When safety-critical software systems are created from a software family, the respective safety-documentation artifacts describing the system for analysis and certification need to be configured, too [SSA13, DL04]. Thus, when using delta modeling as variability realization mechanism, languages such as SFTs need a delta language to express variability. Furthermore, similar to all other realization assets, SFTs are subject to variability in time in the course of evolution. Hence, SFTs were chosen as a running example, as they represent a graphical source language. Furthermore, they demonstrate many of the principal challenges in creating delta languages suitable for variability in time and, yet, are sufficiently comprehensible.

Furthermore, SFTs are one of the languages used for realization assets of the TurtleBot driver of the running example (see Section 1.3) and can be represented as a metamodel of model-driven software development (see Section 3.1). To allow for easier reading, the previous example for an SFT from Figure 1.4 is repeated in Figure 5.2 and the metamodel for SFTs previously presented in Figure 3.3 is repeated in Figure 5.3. Figure 5.2 shows an example SFT, where a root fault of a mobile robot causing a collision is successively decomposed into its causal constituents.

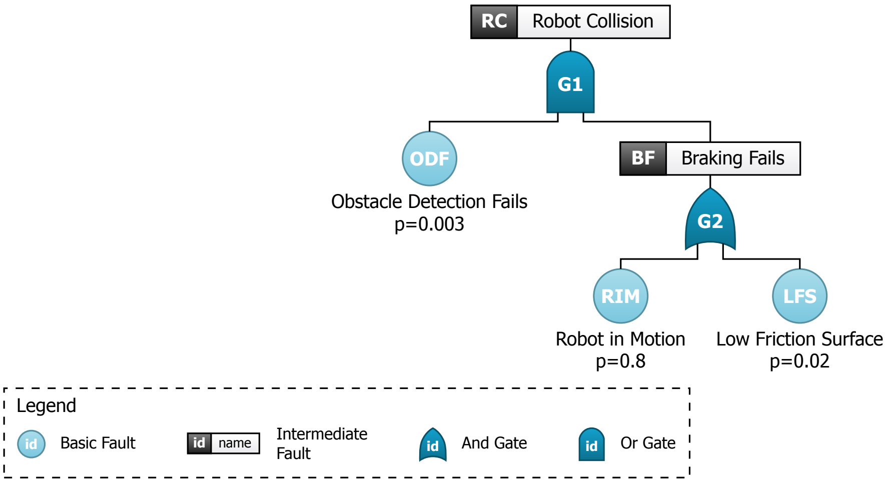


Figure 5.2.: Example of an SFT describing potential combinations of causes for a robot collision (as presented in Figure 1.4).

Figure 5.3 shows the metamodel for SFTs used within the chapter. The metaclass `SFTSoftwareFaultTree` represents the root element of the SFT. In addition, `SFTFault` is the abstract base class for its specializations `SFTBasicFault` and `SFTIntermediateFault` representing the respective faults. Finally, `SFTGate` represents logical gates with the respective logical operator of the enumeration `SFTGateType`. In the metamodel, structural features of metaclasses are distinguished into references and attributes: References relate elements to instances of metaclasses. Attributes have values with basic types, custom data types or enumerations. Furthermore, references are distinguished into single-valued and many-valued references: Single-valued references have an upper bound of one. Many-valued references have an upper bound greater than one resulting in a (possibly ordered) set of values.

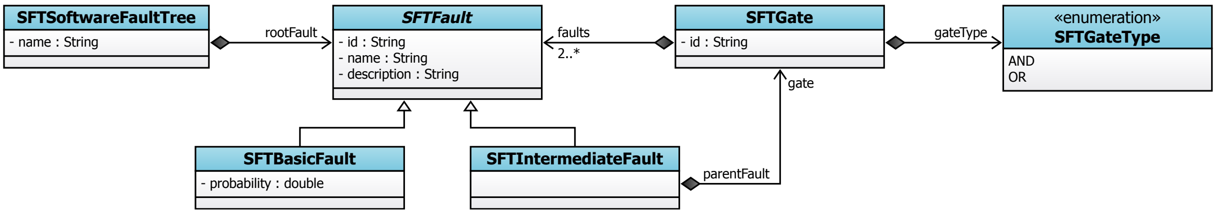


Figure 5.3.: Metamodel for SFTs used to instantiate SFT models (as presented in Figure 3.3).

To cope with variability in space, a delta language should provide operations to create new instances of all concrete metaclasses and to reference existing elements. *DeltaSFT* as delta language for SFTs conforming to the presented metamodel should further allow for adding and removing faults to/from the many-valued `faults` reference to `SFTGate` as well as to set and unset the value of the single-valued `gate` reference to `SFTIntermediateFault`. Furthermore, DeltaSFT has to support modification of the attribute `name` for both fault trees and faults as well as `probability` of basic faults and `gateType` of gates by assigning a new value. The `id` of both faults and gates is closely related to the identity of the respective elements and, thus, should not be subject to changes caused by variability in space. However, in the course of variability in time, this field may change. In addition, multiple further operations that perform modifications of SFT elements may be required to cope with variability in time, e.g., to perform semantics-preserving refactorings or controlled modification of the error propagation logic in an SFT. Hence, delta operations for variability in space have intentionally limited expressiveness in order to avoid inadvertently damaging system integrity during configuration. On the other hand, delta operations concerned with variability in time have to be expressive enough to cope with evolution of the system. Thus, to also represent modifications associated with variability in time, delta modeling has to be extended.

5.3. Evolution Delta Modules as Manifestation of Variability in Time

Principally, both variability in space and time can be captured as transformations on realization artifacts using delta modeling [SBB$^+$10, DS11, KLL$^+$14]. However, there currently is no distinction between delta modules used for configuration (variability in space) and evolution (variability in time). Even though both operations are very similar, as they alter the target artifact by adding, modifying or removing elements, they have distinct differences:

1. **Intent**: A configuration delta module performs changes as part of creating a system variant with the configured functionality, whereas an evolution delta module performs changes to meet new or altered requirements.
2. **Predictability**: A configuration delta module creates an a-priori known variant of the system, whereas an evolution delta module creates an a-priori unknown version of the system.
3. **Expressiveness**: A configuration delta module has an (intentionally) limited expressiveness to perform changes suitable for configuration, whereas an evolution delta module has to be expressive enough to alter all parts of a system potentially affected by evolution.

These differences stem from the fact that a configuration delta module is used to enable or disable functionality associated with a certain feature of a software family, whereas an evolution delta module is used to update a feature to a different revision in order to meet new or changed requirements. The latter may encompass fixing of defects as well as minor changes to functionality that do not alter the identity of a feature with regard to its intended functionality.

Formally, the the set of all delta modules of one particular software family is defined according to Definition 21.

Definition 21: Delta Modules

Let $\Delta = \{\delta_1, \ldots, \delta_n\}$ be the set of all delta modules of one particular software family, Δ_C the set of all configuration delta modules and Δ_E the set of all evolution delta modules. Then the following holds:

1. Each delta module δ is either a configuration or an evolution delta module but never both. $\Delta_C \cap \Delta_E = \varnothing$
2. The set of all delta modules consists only of configuration and evolution delta modules. $\Delta_C \cup \Delta_E = \Delta$

Furthermore, delta modules may specify requires relations stating that all required delta modules necessarily have to be applied when the original delta module is selected and that they have to be applied before the original delta module. Formally, this is captured by the function $\mathrm{req}(\delta) : \Delta \rightharpoonup \mathcal{P}(\Delta)$.
In addition, application-order constraints may specify that a delta module has to be applied after a set of other delta modules (without the explicit need to apply them upon selection of the original delta module). Formally, this is captured by the function $\mathrm{aoc}(\delta) : \Delta \rightharpoonup \mathcal{P}(\Delta)$.

Due to the different characteristics of configuration and evolution delta modules, it is further necessary to distinguish delta operations regarding expressiveness and intended use as *configuration delta operations* and *evolution delta operations*. Even though it is the explicit intent of a configuration delta module to change the functionality of an asset, the respective delta operations are meant to maintain identity of the altered artifacts [SBB+10]. As a consequence, modifying identifiers of an artifact is considered an operation employed solely for evolution. This includes changing direct identifiers, such as the name of SFT elements, as well as parts of identifiers, such as a class name in Java, as it is part of the qualified name identifying the class. Furthermore, refactorings are considered evolution operations as they have the explicit intent of *not altering* functionality and, thus, are not used for configuration such as refactorings to refine basic faults in SFTs or to extract methods and super classes in Java source code. In the following, evolution delta operations may exclusively be employed within evolution delta modules whereas configuration delta operations may be used in both types of delta modules. Hence, within this work, the distinction into variability in space and time is made on level of both delta modules and their delta operations. However, the presented concepts are applicable even if entirely different delta languages for configuration and evolution are employed.

5.4. Automating Delta Language Generation

To employ delta modeling for a particular type of realization artifact, a delta language for the respective source language is required. A delta language defines a number of delta operations suitable to alter artifacts of the source language. Existing delta languages [SBB+10, KHS+14, Sch10, HKM+13, SSA13] define delta operations for the purposes of variability in space. However, with the extensions of this thesis, it is further necessary to provide delta operations for modifications associated exclusively with variability in time. Depending on the nature of these delta operations, it is possible to automatically derive significant parts of their syntax and semantics from the structure specified in the metamodel of the source language. The following sections elaborate on operations that are commonly found in delta languages and, thus, can be derived automatically, as well as on those that have to be specified manually.

5.4.1. Standard Delta Operations Realize Usual Functionality

With the example of SFTs, the need for 6 types of delta operations was illustrated: setting and unsetting the value of single-valued references, adding and removing values of many-valued references, modifying the value of attributes and detaching an element from its container. Furthermore, the need for another operation was identified that can insert a value into a many-valued reference at a specified position provided that the set of values is ordered. Using these 7 types of operations, the following sections define semantics for standard delta operations for variability modeling with EMOF-based models, and they illustrate how to derive them from a source notation's metamodel.

```java
public void generateDeltaOperations(EObject metamodel) {
  List<EReference> allEReferences = getAllEReferences(metamodel);
  List<EClass> allEClasses = getAllEClasses(metamodel);

  generateSetUnsetDeltaOperations(allEReferences);
  generateAddInsertRemoveDeltaOperations(allEReferences);
  generateModifyDeltaOperations(allEClasses);
  generateDetachDeltaOperations(allEReferences);
}
```

Listing 5.1: Principle algorithm for generating delta operations in Java code. The individual methods are refined in Listing 5.2, Listing 5.3, Listing 5.4 and Listing 5.5.

The definition of these operations is included in a *delta dialect* (see Section 5.5.2). The following sections focus on conceptual details of creating delta operations and omit technical concerns that are explained in Section 5.5.2. However, for a more illustrative explanation, the realization of a delta dialect in Section 5.5.2 is referenced as example. Furthermore, Listing 5.1 illustrates the overarching algorithm[1] for generating delta operations from a metamodel, which is used to combine the generation operations presented in Listing 5.2, Listing 5.3, Listing 5.4 and Listing 5.5.

5.4.1.1. Set and Unset Delta Operations

Set and unset delta operations are used to alter the value of a single-valued reference. A set delta operation assigns a new value to a specified single-valued reference. In contrast, an unset delta operation replaces the current value with the default value for that reference as defined by the metamodel.

Set and unset delta operations are derived from a source language's metamodel by collecting all references that are changeable and single-valued. For each reference in the set, a set and unset delta operation are defined. Listing 5.2 demonstrates this procedure using an excerpt of a Java program.

The delta dialect for the SFTs in Listing 5.6 in Section 5.5.2 contains definitions for two set delta operations (Lines 8–9, 26–27) and two unset delta operations (Lines 10, 28). Set delta operations take two parameters: The first one (`value`) is the new value for the single

[1] The structure of the generation algorithm was simplified for reasons of easier explanation without any effects on its semantics. The actual structure of the algorithm is demonstrated in Appendix A.

```java
public void generateSetUnsetDeltaOperations(List<EReference> allEReferences) {
  for (EReference eReference : allEReferences) {
    if (eReference.isChangeable()) {
      if (!eReference.isMany()) {
        createSetOperation(eReference);
        createUnsetOperation(eReference);
      }
    }
  }
}
```

Listing 5.2: Algorithm for generating set and unset delta operations in Java code.

valued reference, the second one (`element`) specifies the element containing the reference to be set. Unset delta operations take merely a single parameter (`element`) denoting the element containing the reference to be unset as the default value used as substitute is specified in the metamodel. The name of the reference affected by set and unset delta operations is specified as part of a delta operation's signature, e.g., in Listing 5.6 Line 21, the reference `gate` of the metaclass `SFTIntermediateFault` is specified. The type of the parameters can be derived from the metamodel: The type of parameter `element` of both set and unset delta operations is obtained by finding the class containing the single-valued reference. Furthermore, the type of `value` of set delta operations is the one the inspected reference refers to. With respect to the metamodel for SFTs depicted in Figure 5.3, for the `gate` reference between the metaclasses `SFTIntermediateFault` and `SFTGate`, the types for the `element` and `value` parameters would be `SFTIntermediateFault` and `SFTGate`, respectively.

As part of the generation process, default names of the set and unset delta operations are created. For this procedure, the following patterns are employed:

- Set: `set<ReferenceName>Of<ContainingClassName>`
- Unset: `unset<ReferenceName>Of<ContainingClassName>`

Both `<ReferenceName>` and `<ContainingClassName>` denote variables to be filled with the name of the reference to be modified (e.g., `gate`) as well as the name of the metaclass containing the reference, respectively. For the latter, any prefixing acronyms (i.e., sequences of multiple uppercase characters before the first letter) are removed, e.g., transforming `SFTIntermediateFault` to "IntermediateFault". Furthermore, values of variables are capitalized to create a camel-cased name for the delta operation. As a result, unique yet descriptive names are created, such as "setGateOfIntermediateFault" or "unsetGateOfIntermediateFault". However, these names merely constitute suggestions and may be changed deliberately by creators of the delta language.

Set and unset delta operations are generally considered configuration delta operations. From the mere structure specified in the metamodel of the source language, it is not possible to determine whether changes on the reference in question should exclusively be performed in the course of evolution. Hence, disallowing set and unset delta operations in configuration delta modules might limit the expressiveness of the delta language to the point that it cannot express concerns of variability in space. If the respective delta operations should exclusively be used for expressing modifications related to variability in time, users may change them to evolution delta operations.

5.4.1.2. Add, Insert and Remove Delta Operations

Add, insert and remove delta operations are provided to manipulate the set of values of many-valued references. An add operation appends a given element to the set of values and a remove operation detaches it from the set. Thus, the semantics of a remove operation is different from that in other approaches to delta modeling [SBB+10, DS11] where it completely erases an element from the model: In case of the approach presented here, the element is only detached from the specified list of references. An insert operation places the element at a certain position within the set of values, which is only sensible, if the set is ordered.

Add, insert and remove delta operations are derived from a source language's metamodel in a similar way as set and unset delta operations: First, a set of all references that are changeable and, in this case, many-valued is collected. As insert delta operations are only sensible for ordered sets of values, for this type of operation, references further have to be marked as being ordered to be included. For each reference in the set, the respective delta operations are created. Listing 5.3 illustrates this general procedure in the form of a partial Java program.

```java
public void generateAddInsertRemoveDeltaOperations(List<EReference> allEReferences) {
  for (EReference eReference : allEReferences) {
    if (eReference.isChangeable()) {
      if (eReference.isMany()) {
        createAddOperation(eReference);
        createRemoveOperation(eReference);

        if (eReference.isOrdered()) {
          createInsertOperation(eReference);
        }
      }
    }
  }
}
```

Listing 5.3: Algorithm for generating add, insert and remove delta operations in Java code.

The delta dialect for SFTs in Listing 5.6 contains one add delta operation (Line 38) and one remove delta operation (Line 39). As none of the many-valued references of the metamodel is marked as being ordered, no insert delta operations are required. Add and remove operations have two parameters: The first parameter (`value`) specifies the value that is to be added/removed. The second parameter (`element`) is used to provide the concrete element for add and remove operations. The name of the affected reference is specified as part of the operation's signature, e.g., in Line 38 of Listing 5.6, the reference `faults` of the metaclass `SFTGate` is specified. Insert operations also use the `value` and `element` parameters, but take a third parameter (`index`) specifying at which position an element should be inserted into the ordered set of values. The type of the `element` and `value` parameters can be derived from the source language's metamodel: The type of `element` is that of the metaclass containing the reference to be modified, the type of `value` is the one the specified reference is pointing to and the type of the `index` parameter of insert operations is fixed as integer.

As part of the generation process, default names of the add, insert and remove delta operations are created. For this procedure, the following patterns are employed:

- Add: `add<ElementClassName>To<ReferenceName>Of<ContainingClassName>`
- Insert: `insert<ElementClassName>Into<ReferenceName>Of<ContainingClassName>`
- Remove: `remove<ElementClassName>From<ReferenceName>Of<ContainingClassName>`

Similar as with set and unset operations, `<ReferenceName>` and `<ContainingClassName>` denote variables containing the names of the reference (e.g., “faults”) and its containing metaclass without prefix (e.g., “Gate” for `SFTGate`). Furthermore, the variable

`<ElementClassName>` contains the name of the metaclass of the `element` parameter without prefixes (e.g., “Fault” for `SFTFault`). These patterns create names such as “addFaultToFaultsOfGate”, “removeFaultFromFaultsOfGate” or “insertFaultIntoFaultsOfGate” for the created add, remove and insert delta operations.

From the structure specified in a source language’s metamodel, it is not possible to determine whether the respective created add, insert and remove delta operations may be used to realize variability in space or exclusively variability in time. Hence, it seems ill-advised to restrain editing capabilities of a delta language as part of the generation process so that all created delta languages are assumed to be configuration delta operations by default. However, a configuration delta operation may be changed to an evolution delta operation within the definition of the delta language by prepending the operation signature with the keyword `evolution`, e.g., as done for the custom delta operation (see Section 5.4.2) in Lines 21–22 of Listing 5.6.

5.4.1.3. Modify Delta Operations

Modify delta operations are used to alter the values of an attribute. In contrast to manipulating referenced values, modification of attribute values is free of side effects (e.g., automatically updated inverse references). Hence, creators of a delta language should be made aware of this difference so that set and modify delta operations are distinguished explicitly. In consequence, modify delta operations have a different meaning from that in other approaches to delta modeling [SBB^{+}10, DS11], where they are used solely to signal that the contents of a hierarchically decomposed element are being altered. In the approach presented here, such a marker is not required, as it is possible to reference elements directly even if they are nested within a containment hierarchy.

Modify delta operations are derived from a source language’s metamodel by inspecting all of its concrete (i.e., non-abstract) metaclasses. For each of these metaclasses, the attributes are iterated and those are collected that are changeable. For each attribute in this set, a modify delta operation is generated. However, a distinction is made depending on whether the respective attribute is marked as being an ID within the Ecore metamodel. As altering IDs inevitably changes the identity of an element, this operation is considered as being solely used as part of evolution. In consequence, either a configuration or an evolution delta operation is created. Listing 5.4 illustrates this procedure by an excerpt of a Java program.

```java
public void generateModifyDeltaOperations(List<EClass> allEClasses) {
  for (EClass eClass : allEClasses) {
    if (!eClass.isAbstract()) {
      for (EAttribute eAttribute : eClass.getEAllAttributes()) {
        if (eAttribute.isChangeable()) {
          boolean isEvolutionOperation = false;

          if (eAttribute.isID()) {
            isEvolutionOperation = true;
          }

          createModifyOperation(eAttribute, isEvolutionOperation);
        }
      }
    }
  }
}
```

Listing 5.4: Algorithm for generating modify delta operations in Java code.

The delta dialect for SFTs in Listing 5.6 defines 10 modify delta operations out of which 7 are configuration delta operations (Lines 11–12, 26–20, 31–34, 41) and 3 are evolution delta operations modifying identifiers (Lines 15, 29–30, 40). Each of these modify operations takes two parameters: The first parameter (`value`) represents the new value for the attribute. The second parameter (`element`) specifies the metaclass containing the attribute. The name of the attribute to be altered is provided as part of the operation's signature, e.g., in Lines 17–18 of Listing 5.6, the attribute `description` of the metaclass `SFTBasicFault` is specified. The types of the `element` and `value` parameters of modify operations can directly be gathered from the source language's metamodel: The type of the `element` parameter is the metaclass containing the specified attribute and the type of the `value` parameter is the same as that of the attribute to be altered. Hence, the type of the `value` parameter is necessarily a primitive data type (in the sense of EMOF) or an enum.

When generating modify delta operations, default names for the derived delta operations are synthesized from the source language's metamodel. For this purpose, the pattern `modify<AttributeName>Of<ContainingClassName>` is used. The variable `<AttributeName>` contains the name of the attribute to be modified and `<ContainingClassName>` the name of the class that contains the attribute without prefixes. Using this pattern, names such as "modifyNameOfBasicFault" are created.

For modify delta operations, it is possible to identify a group of operations that is merely used for changes associated with variability in time: As altering an *identifier* is tightly connected to the *identity* of an element, changing its value inevitably alters the identity of the element, which should only be performed as part of evolution (see Section 3.2.2). Hence, the respective delta operations are created as evolution delta operations exclusively available in evolution delta modules (e.g., in Lines 15, 29–30, 40 of Listing 5.6). For the remaining modify operations, creators of the delta language have to restrict availability of delta operations where applicable.

5.4.1.4. Detach Delta Operations

Detach delta operations are used to remove an element from its container when knowing only the element, but not necessarily its container or the respective containing reference. In this respect, a detach operation can be perceived as the counterpart to the respective remove operation on a many-valued reference or unset operation on a single-valued reference. Detach operations are generated for all concrete metaclasses that possibly are the target of a containment reference[2]. For this purpose, all containment references of the source language's metamodel are iterated. For their respective types, the concrete (i.e., non-abstract) metaclasses are collected in a set. For all metaclasses in this set, a respective detach delta operation is created. Listing 5.5 illustrates this procedure in the form of a partial Java program.

The delta dialect for SFTs in Listing 5.6 defines 3 detach delta operations (Lines 23, 35, 42). A detach delta operation takes a single parameter `element` denoting the element that is to be removed from its container. The concrete reference containing the element during run time is determined dynamically. In the delta dialect for SFTs, no detach delta operations were created for the metaclasses `SFTSoftwareFaultTree` (as it is not contained by any reference) and `SFTFault` (as it is abstract).

A default name for detach operations is synthesized from the class name of the parameter using the pattern `detach<ElementClassName>`. The variable `<ElementClassName>` con-

[2] This may not be the case for all metaclasses of a metamodel. For example, if a base class is abstract, it does not need a dedicated detach delta operation. Furthermore, if there is a specific root metaclass for a metamodel, it may never appear within a containment reference.

```java
public void generateDetachDeltaOperations(List<EReference> allEReferences) {
  Set<EClass> eClassesForDetachOperations = new HashSet<EClass>();

  for (EReference eReference : allEReferences) {
    if (eReference.isContainment()) {
      EClassifier eClassifier = eReference.getEType();
      Collection<EClass> concreteEClasses = findConcreteEClasses(eClassifier);
      eClassesForDetachOperations.addAll(concreteEClasses);
    }
  }

  for (EClass eClass : eClassesForDetachOperations) {
    createDetachOperation(eClass);
  }
}
```

Listing 5.5: Algorithm for generating detach delta operations in Java code.

tains the name of the concrete metaclass without any prefixes. Hence, names such as "detachBasicFault" or "detachGate" are created.

Detach operations may be used as both configuration and evolution delta operations. However, the mere structure of the source language's metamodel does not allow for distinction between these intended uses. Hence, all generated detach operations are created to be configuration delta operations to provide flexibility in altering artifacts of the source language. However, they may be manually restrained from being employed as configuration operations by marking them as evolution delta operations, if the use for the respective source language demands it.

5.4.2. Custom Delta Operations Realize Specialized Functionality

Custom delta operations are used to declare delta operations with user-defined domain-specific semantics that could not be expressed using the generated standard delta operations. This enables creators of a delta language to utilize knowledge of the semantics of the source language to provide specifically tailored operations, e.g., to avoid dangling references according to the constraints of the source language.

Furthermore, arbitrarily complex delta operations may be specified, e.g., as evolution delta operations to realize changes associated with variability in time. Using this mechanism, it is, e.g., possible to provide refactorings tailored to the source language. For example, in Java, it would be possible to provide operations to move declarations of fields and methods along the inheritance hierarchy or to extract new classes containing a subset of the fields and methods of an original class. The delta dialect for SFTs in Listing 5.6 defines 1 custom delta operation for the purposes of evolution (Lines 21–22).

Due to the flexible nature of custom delta operations, there are no conventions regarding names of the operations or restrictions regarding number and type of parameter. Furthermore, the semantics of these operations depends entirely on the behavior intended by the creator of the delta language, so that the implementation to interpret the respective custom delta operations needs to be provided manually. However, using the delta language creation infrastructure presented in Section 5.5, custom delta operations are automatically integrated in the overall infrastructure, so that only the actual changes to be performed by the delta operation have to be specified as transformations on the respective model. To ease the definition of complex delta operations, it is further possible to reuse the definition of other (standard or custom) delta operations to define compound delta operations.

For the derivation of delta operations, it was decided to explicitly *not* include two specific operations in the set of standard delta operations that may, in consequence, have to be realized as custom delta operations:

For one, a replace delta operation was not defined as it inherently depends on the semantics of the source language whether elements of the exact same type, those compatible in the sense of subtype polymorphism or semantically equivalent elements may be used as substitutes. Hence, neither the syntax nor the semantics of a replace delta operation could be fixed in general.

Furthermore, no standard delta operation was defined that completely erases an element from the model along with all its references. Such an operation would have too many (potentially unintended) side effects to be sensible for variability modeling in general. Hence, an element either has to be deleted step by step using standard delta operations, or a custom delta operation specific to the source language has to be defined, which may be done for abstract metaclasses to cover multiple concrete metaclasses at once. The nature of such an operation as either configuration or evolution delta operation has to be determined by creators of the delta language.

5.5. Delta Language Creation Infrastructure

To reuse parts common to all delta languages and to put custom delta languages onto a common technological base, a language creation infrastructure is provided for delta language generation employing the procedures described in Section 5.4.

For this purpose, two languages are used: 1) The *common base delta language*, which provides functionality common to all delta languages such as creating and referencing elements and 2) a *delta dialect*, which provides delta operations specific to the source language. A custom delta language is created by combining the common base delta language with a delta dialect specific to the respective source language. This general architecture is illustrated in Figure 5.4.

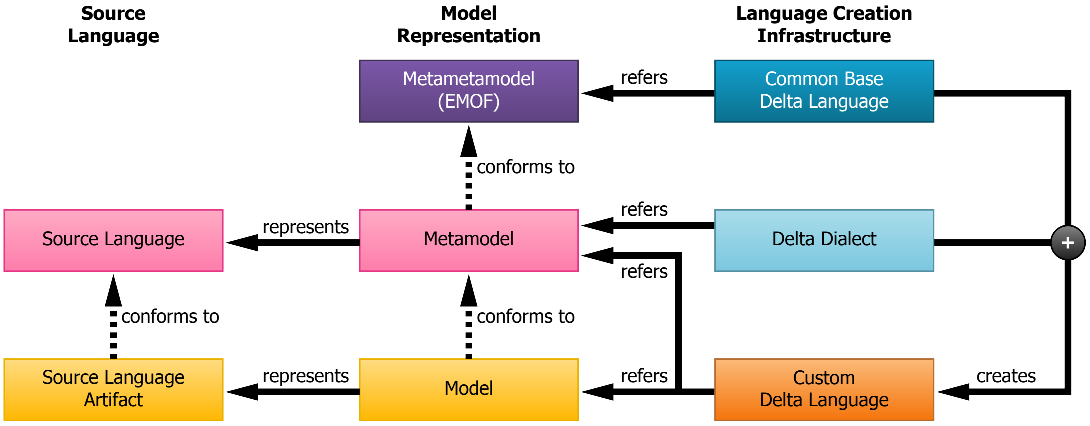


Figure 5.4.: Architecture of the delta language creation infrastructure.

The result of the generation process are syntax, semantics and tools for the custom delta languages including editor support, parsers and interpreters. The delta languages are agnostic of the representation of the source language so that both textual and graphical source languages are supported. However, the delta language to modify them always has a textual representation as is common in other delta modeling approaches as well [SBB+10, Sch10, LSKL12, SSA13, HKR+11, HKM+13]. The generated delta languages seamlessly integrate due to their common base in EMOF so that an arbitrary number of different delta languages can be used with a single tool suite.

The main goal of the language creation infrastructure is to allow for the swift creation of custom delta languages by reusing common parts and generating the majority of the delta dialect for a particular source language as far as feasible (including syntax and semantics of delta operations). For the example of SFTs as source language, a delta dialect is derived from the language's metamodel, which is combined with the common base delta language to form *DeltaSFT* as custom delta language. The following sections explain the common base delta language as well as delta dialects and describe the creation of DeltaSFT as an example of a custom delta language.

5.5.1. The Common Base Delta Language Provides Shared Functionality for all Delta Languages

The common base delta language operates on the level of the metametamodel (M3 level or *EMOF*, see Figure 3.1) using e.g., `EReference`s of Ecore as elements, but not on their instances in the metamodel of the source language, such as the reference `faults` of the metaclass `SFTGate` in the metamodel for SFTs defined in Figure 5.3. Hence, the common base delta language requires no knowledge of the source language's metamodel. In consequence, it is provided entirely by the language creation infrastructure. The common base delta language represents the basis of the custom delta language that is to be created.

A great number of language structures commonly used in delta languages is provided by the common base delta language.

- **Model References** import models specified as being subject to variability.
- **Delta Module Dependencies** list delta modules that have to be applied as prerequisite for a certain delta module.
- **Element Identifiers** are resolved to the respective elements in the modified model in order to use them for modification (language dependent identifiers are possible).
- **Element Constructors** are dynamically created with named parameters to instantiate elements of the source language using their respective metaclasses.
- **Variable Definition and Scoping** to hold referenced or created model elements for use within a delta module.
- **Delta Operation Invocation** allows for calling delta operations with the adequate number and types of arguments.

These constructs are available in all generated delta languages, but can be defined independently from the concrete source language. To avoid having to define them for each delta language individually, these constructs are provided as part of the language creation infrastructure and are shared between different delta languages as core metamodel package. As the common base delta language is defined in a metamodel, it is possible to perform operations such as type checks to ensure that the types of referenced objects, variables and parameters are compatible. Furthermore, a concrete textual syntax is provided with the metamodel of the common base delta language, which is used as basis for the textual custom delta language when combining the common base delta language with a delta dialect.

5.5.2. Delta Dialects Define Delta Operations for Custom Delta Languages

Delta dialects define delta operations suitable for expressing variability both in space and in time for a particular source language, e.g., to add faults as children of a gate for SFTs. Thus, a delta dialect is the part of a custom delta language that ties to the metamodel of a specific source

```
1  deltaDialect
2  {
3    configuration:
4      metaModel: <http://vicci.eu/sft/1.0>;
5
6    deltaOperations:
7      //Software Fault Tree
8      setOperation setRootFaultOfSoftwareFaultTree(SFTFault value,
9        SFTSoftwareFaultTree[rootFault] element);
10     unsetOperation unsetRootFaultOfSoftwareFaultTree(SFTSoftwareFaultTree[rootFault] element);
11     modifyOperation modifyNameOfSoftwareFaultTree(String value,
12       SFTSoftwareFaultTree[name] element);
13
14     //Basic Fault
15     evolution modifyOperation modifyIdOfBasicFault(String value, SFTBasicFault[id] element);
16     modifyOperation modifyNameOfBasicFault(String value, SFTBasicFault[name] element);
17     modifyOperation modifyDescriptionOfBasicFault(String value,
18       SFTBasicFault[description] element);
19     modifyOperation modifyProbabilityOfBasicFault(Double value,
20       SFTBasicFault[probability] element);
21     evolution customOperation refineBasicFault(SFTBasicFault basicFault, String gateId,
22       SFTGateType gateType, SFTFault subFault1, SFTFault subFault2);
23     detachOperation detachBasicFault(SFTBasicFault element);
24
25     //Intermediate Fault
26     setOperation setGateOfIntermediateFault(SFTGate value,
27       SFTIntermediateFault[gate] element);
28     unsetOperation unsetGateOfIntermediateFault(SFTIntermediateFault[gate] element);
29     evolution modifyOperation modifyIdOfIntermediateFault(String value,
30       SFTIntermediateFault[id] element);
31     modifyOperation modifyNameOfIntermediateFault(String value,
32       SFTIntermediateFault[name] element);
33     modifyOperation modifyDescriptionOfIntermediateFault(String value,
34       SFTIntermediateFault[description] element);
35     detachOperation detachIntermediateFault(SFTIntermediateFault element);
36
37     //Gate
38     addOperation addFaultToFaultsOfGate(SFTFault value, SFTGate[faults] element);
39     removeOperation removeFaultFromFaultsOfGate(SFTFault value, SFTGate[faults] element);
40     evolution modifyOperation modifyIdOfGate(String value, SFTGate[id] element);
41     modifyOperation modifyGateTypeOfGate(SFTGateType value, SFTGate[gateType] element);
42     detachOperation detachGate(SFTGate element);
43 }
```

Listing 5.6: Textual representation of a delta dialect for SFTs.

language. The delta language itself is created by combining the common base delta language with the respective delta dialect for the source language. The structure for delta dialects is defined in a metamodel and specified in a concrete textual syntax (see Listing 5.6).

Listing 5.6 shows the textual representation of a delta dialect for SFTs conforming to the metamodel introduced in Figure 5.3 as generated by the language creation infrastructure. When combining this delta dialect with the common base delta language, DeltaSFT is created, which can be used to specify variability for SFTs in both configuration and evolution delta modules. In the `configuration` section of the delta dialect (Lines 3–4), the metamodel of the source language is identified by specifying its Uniform Resource Identifier (URI) as parameter to the `metaModel` key (Line 4). In the `deltaOperations` section (Lines 6–42), individual delta operations available in the delta language are defined with their signature.

Delta dialects allow for distinction between the two types of delta operations by using one of the keywords `configuration` or `evolution` as modifier to the definition of a delta operation.

This restrains evolution operations to only be available within evolution delta modules. If the keyword is omitted, the delta operation is implicitly assumed to be a configuration operation, which may be used in both configuration and evolution delta modules. Furthermore, dedicated keywords for any of the 7 types of standard delta operations may be used (e.g., `setOperation` or `addOperation`). The structure of each standard delta operation's signature obeys the conventions introduced in Section 5.4. Furthermore, custom delta operations may be defined using the keyword `customOperation` with an arbitrary number of parameters of arbitrary types. For the 7 types of standard delta operations, the implementation of an interpreter is provided automatically as their semantics are defined. For custom delta operations, the respective implementation has to be provided manually as Java code. However, an adequate integration into the generated interpreter is provided by creating Java method stubs for the interpretation of the defined custom delta operations.

5.5.3. Custom Delta Languages Enable Variability in Source Languages

A custom delta language is created by automatically introducing references between the metamodels of the common base delta language and the respective delta dialect. Along with a metamodel, a concrete textual syntax is provided for the resulting custom delta language that is synthesized from the textual syntax of the common base delta language and the metaclasses in the source language. Listing 5.7 provides an example of a delta module specified in DeltaSFT, which was created using the delta dialect in Listing 5.6. It modifies an SFT capturing the causes for the collision of a mobile robot similar to the TurtleBot. The basic variant of the SFT is loaded in Line 3. In Lines 5–16, it is further modified by applying delta operations specific to the source language of SFTs to include fault propagation paths for an add-on distance sensor.

```
1  configuration delta "Add Bump and Distance Sensors"
2    dialect <http://vicci.eu/ecosystem/sft/1.0>
3    requires <../core/RobotCollision.sft>
4  {
5    removeFaultFromFaultsOfGate(<ODF>, <G1>);
6    SFTIntermediateFault odf = new SFTIntermediateFault(id: "ODF",
7      name: "Obstacle Detection Fails");
8    addFaultToFaultsOfGate(odf, <G1>);
9
10   SFTGate g3 = new SFTGate(id: "G3", gateType: SFTGateType.AND);
11   setGateOfIntermediateFault(g3, odf);
12
13   addFaultToFaultsOfGate(new SFTBasicFault(id: "BSF",
14     name: "Bump Sensor Fails", probability: 0.003), g3);
15   addFaultToFaultsOfGate(new SFTBasicFault(id: "DSF",
16     name: "Distance Sensor Fails", probability: 0.0007), g3);
17 }
```

Listing 5.7: Example usage of DeltaSFT to alter SFTs in the course of variability in space.

Furthermore, Listing 5.8 depicts an example of applying DeltaSFT to modify an SFT in the course of variability in time to perform evolution. As part of the further development of the robot, sensors were added to detect wheelspin and deviation from the intended course through inaccuracy in the robot's positioning. Data from these sensors is used to synthesize an intermediate fault that indicates that the robot is slipping. The evolution delta module in Listing 5.7 reflects these changes by refining the previously generic basic fault indicating a low friction surface into a disjunction of the basic faults reported by the respective sensors (Lines 10–14) and changing the fault's name and description to be more appropriate (Lines 7–8).

```
1  evolution delta "Add Wheelspin and Steering Inaccuracy Detection"
2    dialect <http://vicci.eu/ecosystem/sft/1.0>
3    requires <../core/RobotCollision.sft>
4  {
5    SFTBasicFault lfs = <LFS>;
6
7    modifyIdOfBasicFault("RSL", lfs);
8    modifyDescriptionOfBasicFault("Robot Slipping", lfs);
9
10   SFTBasicFault wsd = new SFTBasicFault(name : "WSD",
11     description : "Wheelspin Detected", probability: 0.016);
12   SFTBasicFault sia = new SFTBasicFault(name : "SIA",
13     description : "Steering Inaccurate", probability: 0.004);
14   refineBasicFault(lfs, "G4", SFTGateType.OR, wsd, sia);
15 }
```

Listing 5.8: Example usage of DeltaSFT to alter SFTs in the course of variability in time.

To reflect the intended use as either configuration or evolution delta modules in DeltaEcore, distinct keywords `configuration` and `evolution` are used as modifiers to the keyword `delta` as, e.g., in Line 1 of Listing 5.7 or Listing 5.8, respectively. For reasons of backward compatibility, the prior keyword may be neglected to implicitly create a configuration delta module. The use of evolution delta operations is restrained automatically by the generated delta language so that they can only be employed within evolution delta modules. Even though the generated custom delta languages have a textual representation, their foundation in a metamodel principally allows for a graphical representation as well.

5.6. Case Study

To demonstrate feasibility of the delta language generation concepts presented in this chapter, a case study was performed. In the case study, the suitability of the concepts presented in this chapter is evaluated for 4 different source languages: Software Fault Trees (SFTs) [Lev95, Her00], Component Fault Diagrams (CFDs) [SSA13, KLM03], Checklists (CLs) [O'C04, Lev95, Rus92] and the Goal Structuring Notation (GSN) [KW04] (see Section 1.3). An example of SFTs was already presented in Figure 5.2. Examples of CFDs, CLs and the GSN were presented in Section 1.3 and are repeated for easier reading in Figure 5.5, Listing 5.9 and Figure 5.6, respectively.

```
checklist "Test Braking System"

group "Surface Type"
  F1 "Wooden Floor"
x F2 "Carpet"
x F3 "Concrete"
  F4 "Wet Floor"

group "Speed Level"
x S1 "Low Speed"
  S2 "Regular Speed"
  S3 "High Speed"
```

Listing 5.9: Example of a CL representing different combinations of conditions to inspect during test of a robot's braking system.

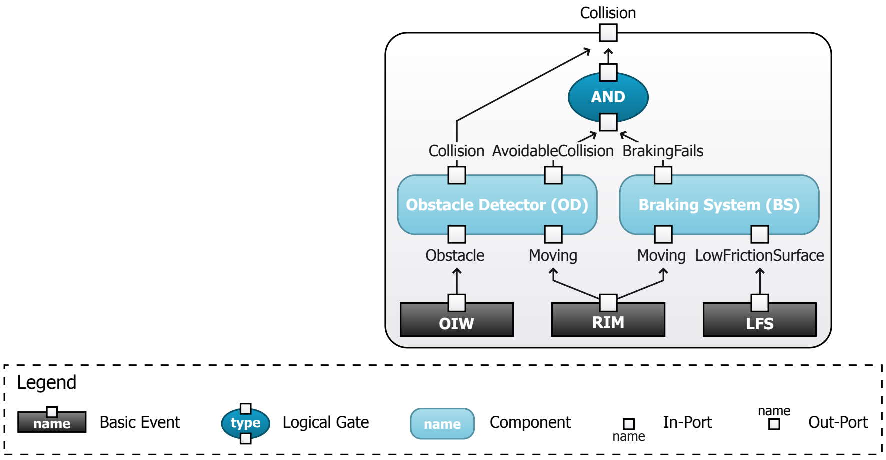


Figure 5.5.: Example of a CFD describing potential combinations of causes for a robot collision as a reusable component.

All these languages stem from the area of certifying safety-critical systems and are used as part of the realization assets for the running example of the configurable TurtleBot driver (see Chapter 1). Despite their similarity in origin, the languages contain many different characteristics representing a wide range of languages. The abstract syntax of SFTs is represented by a tree, that of CFDs and CLs by a reducible graph and that of the GSN by a general graph. SFTs, CFDs and the GSN have a graphical syntax, whereas CLs have a textual syntax. Finally, the GSN may reference model elements from SFTs, CFDs and CLs interconnecting artifacts of the different languages.

Within the case study, delta languages are generated from the respective metamodels of the source languages in order to investigate adequacy of the introduced concepts with regard to the requirements listed in Section 5.1. For CFDs, a dedicated delta language was already presented as part of previous work [SSA13]. For SFTs, CLs and the GSN, the respective delta languages were created manually to have a reference for comparison with delta languages generated with the presented concepts. To determine in how far the posed requirements are satisfied, the

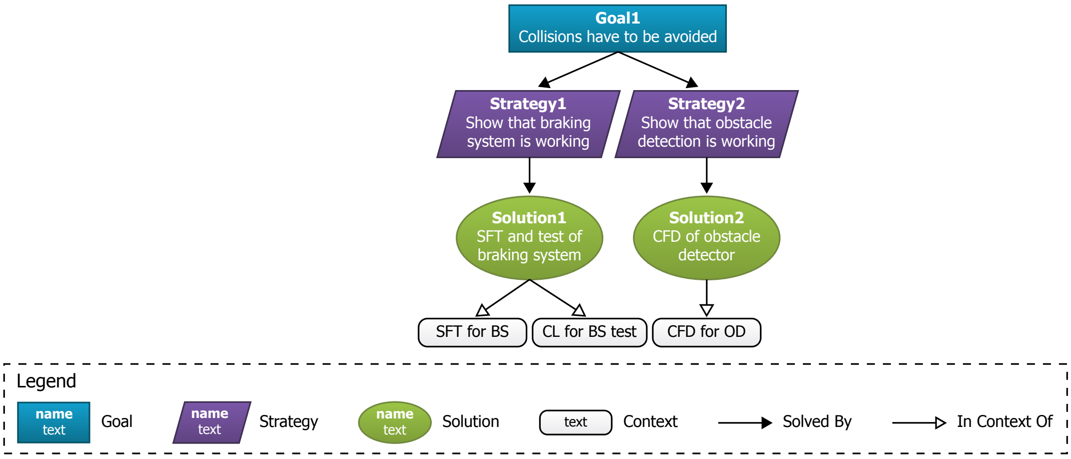
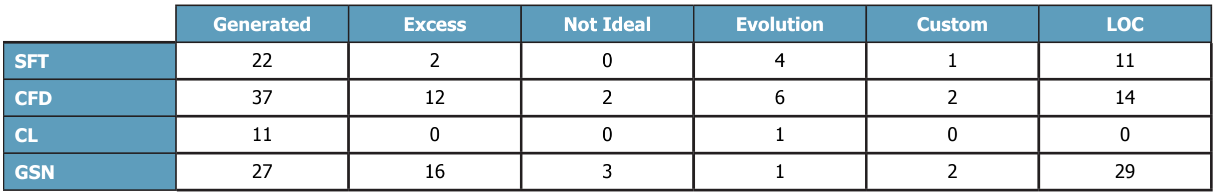


Figure 5.6.: Example of the GSN representing a semi-formal line of argumentation on the safety of a robot's collision avoidance mechanism.

adequacy of derived delta operations is analyzed and the complexity of implementing custom delta operations is inspected by measuring Lines of Code (LOC).

To generate the delta languages, the implementation of the concepts of this thesis as presented in Chapter 7 was employed. Within the delta languages, configuration and evolution delta operations are distinguished where the latter may exclusively be used in evolution delta modules but not in configuration delta modules. In Table 5.1, metrics for the generated languages are provided. The column *Generated* contains the number of all generated delta operations, *Excess* counts those delta operations that are redundant (e.g., providing access to an opposite reference), *Not Ideal* lists the number of operations that were perceived as not being elegant for the intended purpose (e.g., setting the bounding box for the graphical representation of an element instead of moving and resizing it) and *Evolution* states the number of (generated or manually created) delta operations that had to be marked as evolution operations in order to disallow access to model elements that should not be affected by variability in space. Finally, *Custom* lists the number of custom delta operations used in the delta language (both for configuration and evolution) and *LOC* states the number of lines of code required to implement their intended semantics.



	Generated	Excess	Not Ideal	Evolution	Custom	LOC
SFT	22	2	0	4	1	11
CFD	37	12	2	6	2	14
CL	11	0	0	1	0	0
GSN	27	16	3	1	2	29

Table 5.1.: Results of deriving delta dialects for the source languages of the case study.

The generated standard delta operations of all delta dialects were sufficient for handling variability in space in the respective source languages with regard to the original expectations regarding the manually created delta dialects. However, CFDs and GSNs have a relatively large number of excess methods. This is mostly due to the presence of multiple opposite references where delta operations were generated for both the original and opposite reference, which creates redundancy. However, the majority of these delta operations is considered useful and merely was not included in the original delta language due to the implementation effort at the time. Furthermore, the delta languages for SFTs, CFDs, CLs and the GSN each contain evolution delta operations to modify identifiers of artifacts in the respective source languages. To realize additional delta operations, SFTs required 1, CFDs 2 and the GSN 2 custom delta operations with 11, 14 and 29 LOC, respectively.

With regard to the initially posed requirements, the following conclusions are drawn: Using the language generation capabilities presented in this chapter, it was possible to completely generate delta languages for the respective source languages that are expressive enough to handle variability in space so that `R2.1` is satisfied. Even though it was possible to alter all elements with the derived delta operations, in some cases, providing more elegant delta operations was desirable. For example, the generated delta operations to alter the visual appearance of CFD elements suggested setting the bounding box of the element, whereas the source language used delta operations to move and resize the element, which seemed more intuitive to use. Delta operations missing from the generated delta dialect could be realized by custom delta operations and manual implementation of the semantics in the dialect interpreter.

Within the generated delta languages, delta operations designated for addressing variability in time could be specified with the desired semantics. The operations could further be prevented from being used for variability in space by declaring them as evolution delta operations so that

R2.2 is satisfied. However, generation is limited to modification operations for identifiers whereas further evolution delta operations have to be provided manually. Access to elements considered immutable in the course of variability in space could be restricted by manually marking the respective operations as evolution delta operations. With the model-based nature of the language generation, which perceives source languages as metamodels, both textual and graphical source languages can be handled as demonstrated by the various examples of the case study so that R2.3 is satisfied. With the standard delta operations, the conventions for their names and parameters as well as their defined semantics, it was possible to derive a fully functional delta language from a source language given as metamodel which satisfies both R2.4 and R2.5.

5.7. Demarcation from Related Work

Multiple publications exist that present individual delta languages for particular source languages, such as for Java [SBB$^+$10], Class Diagrams [Sch10], State Charts [LSKL12], Component Fault Diagrams [SSA13], the architectural language MontiArc [HKR$^+$11] or Matlab/Simulink [HKM$^+$13]. However, these delta languages are tightly integrated with their source languages and, thus, serve as archetypes of syntax and semantics of delta languages, but not as basis for generating custom delta languages for arbitrary metamodels.

Further approaches strive for goals similar to that of this chapter by generating or providing languages to manipulate artifacts of various source languages. The following paragraphs discuss these approaches in detail with regard to the requirements of Section 5.1. Table 5.2 contrasts the variability realization mechanism for variability in time and the presented language generation capabilities with all discussed related approaches.

Haber et al. [HHK$^+$13] present the work related closest to that of this chapter, as it has a similar goal of generating a delta language for a given source language. In the approach, a concrete syntax for a custom delta language is derived by means of grammar extension from a provided textual source language's grammar. For this purpose, the grammar of a common delta language is used as basis and extended with syntactical definitions of common delta operations. The result is the grammar of a textual delta language for the provided source language.

With the generated delta operations, it is principally possible to realize modifications associated with variability in space, if an interpreter provides semantics, so that R2.1 is satisfied. However, handling changes associated with variability in time is not in focus of the approach, so that R2.2 is unsatisfied. The approach defines a delta language exclusively as concrete textual syntax but is incapable of handling graphical source languages, so that R2.3 is unsatisfied. With the mere creation of a textual grammar, only the syntax of a delta language is provided, which leaves R2.4 unsatisfied. Even though a selection of standard operations is created automatically, their semantics is not defined, which leaves R2.5 partially unsatisfied.

Sánchez et al. [SLFG09] present another closely related approach where a framework is used to define domain-specific languages for variability management for artifacts of a particular metamodel. **Zschaler et al.** [ZSS$^+$10] extend this work by bootstrapping SPL technologies to create a family of variability modeling languages. Within these approaches, it is possible to define custom operations to modify artifacts conforming to the metamodel of the source language. A direct addressing mechanism is used where elements of the modified artifact may be referenced by giving their identifier. The concrete resolution of an identifier to the respective model element has to be provided by the creator of the variability language. Semantics of the defined modification operations have to be provided manually by specifying transformation rules in a general purpose transformation language.

	R2.1 Handle Variability in Space	**R2.2 Handle Variability in Time**	**R2.3 Textual/ Graphical Source Languages**	**R2.4 Define Syntax/Semantics of Operations**	**R2.5 Automatically Create Standard Operations**
Variability Realization Mechanism for Variability in Time	+	+	+ (Textual, Graphical)	+ (Syntax, Semantics)	+
Delta Language Grammar Extension [HHK+13]	+	-	o (Textual)	- (Syntax)	o (No Semantics)
Variability Modeling Language Family (VML*) [SLFG09, ZSS+10]	+	o	+ (Textual, Graphical)	+ (Syntax, Semantics)	-
FeatureHouse [AKL13]	+	-	o (Textual)	+ (Syntax, Semantics)	o (No Derivation)
Common Variability Language (CVL) [HMPO+08]	+	o	+ (Textual, Graphical)	o (Manual Syntax, Semantics)	o (No Derivation)
Change-Oriented Programming (ChOP) [EVC+07]	o	+	o (Textual)	o (Semantics)	+
Domain-Specific Model Transformation [RW11]	-	+	+ (Textual, Graphical)	o (General Syntax, Semantics)	-
General Model Transformation (QVT, ATL, ETL etc.)	-	+	+ (Textual, Graphical)	-	-
Invasive Software Composition (ISC) [Aßm03]	o	o	+ (Textual, Graphical)	o (Semantics)	-

Table 5.2.: Comparison of the variability realization mechanism for variability in time of Chapter 5 with related approaches using the requirements of Section 5.1.

The variability management languages created using this approach are explicitly designed to cope with variability in space so that R2.1 is satisfied. However, even though generally possible due to the general nature of transformation operations, handling modifications for the purposes of variability in time is not in focus of the approach, so that R2.2 is only partially satisfied. Through the definition of the variability management language for a metamodel, the approach may target both textual and graphical languages, so that R2.3 is satisfied. The approach allows for the specification of both syntax and semantics of modification operations, so that R2.4 is satisfied. However, no standard operations are defined or generated leaving R2.5 unsatisfied.

Apel et al. [AKL13] present FeatureHouse as an approach for generalizing software composition by superimposition for artifacts written in different languages. FeatureHouse can be seen as a language workbench for feature-oriented variability modeling languages in the sense of FOP, which is similar to the approach for delta-oriented variability modeling presented in this chapter. However, the resulting languages use a compositional variability realization mechanism, which is not suitable for the purposes of the thesis (see Section 3.2.2). Furthermore, FeatureHouse does not operate on metamodels of the source languages but relies on the parse tree for the source language and the concept of Feature Structure Trees (FSTs), which resemble abstract syntax trees. The FSTs can be composed using a set of predefined operations with associated semantics similar to the standard delta operations provided in this chapter. So far, FeatureHouse has only been used for textual languages.

The languages generated by FeatureHouse are designated for handling variability in space but are not suitable for handling variability in time, which satisfies `R2.1` and, at the same time, leaves `R2.2` unsatisfied. Furthermore, the dependence of FeatureHouse on an FST is tightly integrated with textual source languages. Even though the potentially graph-based structure of graphical source languages may, in principle, be reduced to an FST, FeatureHouse currently does not support this procedure, so that `R2.3` remains unsatisfied. With the set of predefined operations with both syntax and semantics in FeatureHouse, `R2.4` is satisfied. However, the automatic generation of these operations is not supported, so that `R2.5` is unsatisfied.

Haugen et al. [HMPO$^+$08] introduce the Common Variability Language (CVL) as a standardization effort for variability languages. The approach is closely related to that presented in this chapter, as it has the goal of extending arbitrary MOF-based models with a variability realization mechanism. Conceptual configuration knowledge is captured in *VSpecs* (resembling feature models) and overlayed on realization assets. Variant derivation facilities may be employed to manifest the effects of changes associated with variability in realization assets.

The primary concern of CVL is the handling of variability in space, which is supported both on conceptual and realization level so that `R2.1` is satisfied. However, variability in time may only be manifested on realization level through transformations by using workarounds on the conceptual level, so that `R2.2` is satisfied only partially. Due to the model-based nature of the approach and its independence of the concrete syntax of the source language, CVL supports both textual and graphical source languages, which fulfills `R2.3`. Furthermore, CVL defines semantics of certain standard operations that may be performed as part of variability modeling but does not provide a standard syntax, so that `R2.4` remains partially unsatisfied. Finally, the provided standard operations are generic, so that there is no derivation of operations for individual source languages, which satisfies `R2.5` partially.

Ebraert et al. [EVC$^+$07, EMD08, ESJ11] define Change-Oriented Programming (ChOP) to capture evolutionary changes to source code within *change objects*. A set of interrelated change objects is grouped to a *change set*. Changes on the respective assets are captured strictly by recording modifications and are represented in a model containing change objects and change sets. In consequence, the approach does not offer a particular language to alter artifacts. Furthermore, modifications to artifacts have to be transactional in the sense that, at the end of a change set, the altered artifact is valid. With this approach, primarily issues of variability in time are addressed. Basic support for variability in space exists by using the change information to extract feature models [ECHD09].

Even though deriving feature models for variability in space is possible, the created models are tightly aligned with the technical realization and do not appropriately capture conceptual concerns of configuration which satisfies `R2.1` only partially. Through the focus on evolution, concerns of variability in time can be handled appropriately, which satisfies `R2.2`. Even though it might be possible to apply the general concepts of the approach to graphical source languages, currently, there only are implementations for textual source languages, so that `R2.3` is satisfied only partially. Through the strict recording of changes as opposed to specifying them explicitly, there is no syntax for the change operations but only a semantics, which satisfies `R2.4` partially. Even though the approach does not employ a generation procedure, standard operations are available for the supported languages, so that `R2.5` is satisfied.

Besides approaches providing or generating languages to specifically handle variability, there are also more general approaches to model transformation that can be utilized for similar purposes.

Rumpe and Weisemöller [RW11] generate a domain specific model transformation language from the concrete syntax of a source language. With the approach, it is possible to alter artifacts of arbitrary languages provided that they are available as models conforming to a metamodel of the respective language. The transformation languages may principally be used for configuration or evolution. However, the focus of the approach is not on variability so that no standard variational operations with defined semantics or a variant derivation mechanism are provided.

Hence, variability in space is not handled explicitly leaving `R2.1` unsatisfied. However, the general applicability of model transformation and its wide range of modification operations make it suitable for handling variability in time, so that `R2.2` is satisfied. Even though the generated transformation languages are strictly textual, they may address both textual and graphical source languages due to their model-based nature, which satisfies `R2.3`. In the approach, standard operations are outlined with a general syntax and semantics, so that `R2.4` is satisfied. However, the generation of these operations is marked as future work and is not realized within the approach leaving `R2.5` unsatisfied.

In addition, there are multiple **general purpose model transformation approaches** of which graph-based approaches are most suitable for variability modeling [CH06] with specifications, such as QVT[3], and languages targeting Ecore, such as ATL[4] or ETL[5]. With these languages, it is possible to alter artifacts of arbitrary languages represented as models in all regards irrespective of the intent for the modification. However, using general purpose model transformation languages to express variability is problematic, as they are not tailored to the field of variability management. As a result, it is problematic to restrain available modification operations for a particular purpose, which creates the risk of inadvertently damaging system integrity by using too powerful modification operations.

Hence, general purpose model transformation languages have no dedicated means for handling variability in space, which leaves `R2.1` unsatisfied. However, due to their powerful modification operations, they may be used to handle variability in time so that `R2.2` is satisfied. Due to modifying language artifacts as models, general purpose model transformation approaches may be applied to both textual and graphical source languages, which satisfies `2.3`. However, the lack of defined standard operations and their generation leaves both `R2.4` and `R2.5` unsatisfied.

Aßmann [Aßm03] defines Invasive Software Composition (ISC) as a fragment-based gray-box composition technique. The approach employs transformation to adapt and extend *components* (represented as *fragment containers*) at specific (implicit or explicit) *hooks*. Using a *composer*, hooks are bound by performing a set of transformation operations, e.g., substituting a series of statements for a hook within a source code fragment. Due to its capacity to also delete elements, ISC may be perceived as a transformational variability realization mechanism for both variability in space and time.

However, despite its capacities to alter source language artifacts, ISC offers no dedicated support for either variability in space or variability in time, such as definition of configurations or derivation of variants, so that both `R2.1` and `R2.2` are satisfied only partially. The hook mechanism of ISC is independent of the source notation's concrete syntax so that both textual and graphical source languages are supported, which satisfies `R2.3`. Furthermore, ISC may be applied to various source languages, possibly employing a different composition language, so that the approach itself defines only semantics of modification operations but no concrete syntax, which leaves `R2.4` partially unsatisfied. Finally, there is no intention of automatically creating a realization of standard operations for individual source languages so that `R2.5` is unsatisfied. However, due to its nature

[3] `http://omg.org/spec/QVT/1.0`
[4] `http://eclipse.org/atl`
[5] `http://eclipse.org/epsilon/doc/etl`

as a general approach to composition that also supports removal of elements, ISC could be used as an implementation of delta modeling if explicit support for software families was added.

5.8. Chapter Summary

This chapter presented concepts and a language creation infrastructure to generate delta languages for source languages given as EMOF-based metamodels in order to express variability in space and time. An explicit distinction of configuration and evolution delta modules was introduced to perform changes related to enabling or disabling functionality on the one hand and meeting changed requirements or fixing defects on the other hand. Furthermore, procedures were introduced to derive syntax and semantics for custom delta languages from a source language's metamodel. For this purpose, the syntax and semantics for 7 types of standard delta operations were defined. Furthermore, a procedure was devised to analyze an EMOF-based metamodel of a source language to find suitable instances of these operations.

This information was used to define a delta dialect to extend a common base delta language in order to create a custom delta language. The generated delta languages integrate seamlessly into a common technological infrastructure, which makes them interoperable. Furthermore, they can be applied to a wide variety of different source languages with both textual and graphical representation as demonstrated in a case study. With the created delta languages, modifications associated with variability in space and time can be manifested as transformations on realization assets, which serves as basis for the derivation of individual variants. Figure 5.7 illustrates the contribution of Chapter 5 in context of the thesis.

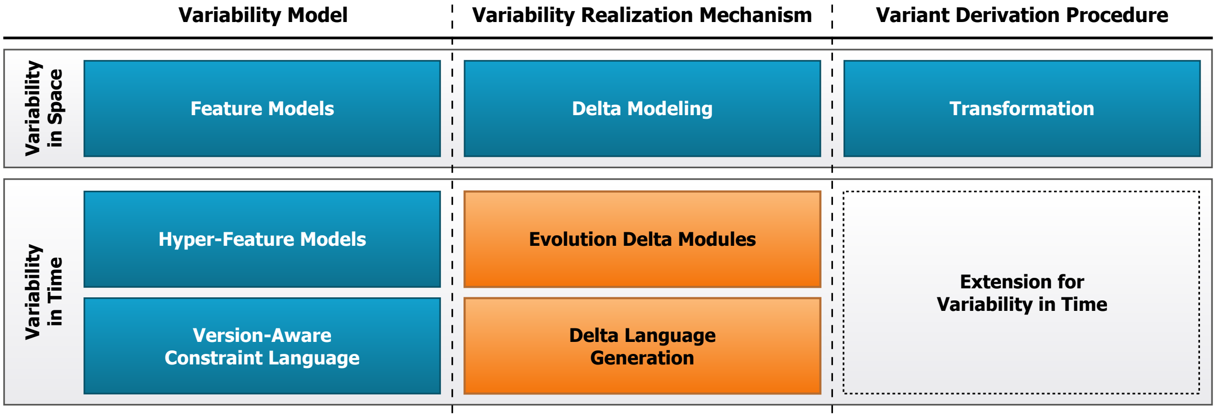

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Hyper-Feature Models
Evolution Delta Modules
Extension for Variability in Time
Version-Aware Constraint Language
Delta Language Generation

Figure 5.7.: Contributions of Chapter 5 in context of the thesis.

6. Deriving Variants with Variability in Space and Time

The contents of this chapter are largely based on the work published in [SSA14c, SSA14a].

Summary *Over the course of time, variable assets of SPLs and especially SECOs are subject to change in order to meet new requirements as part of software evolution (variability in time). The effects of variability in time may influence variability in space, e.g., when not all customers upgrade their respective products immediately or completely. Hence, variability in space and time have to be handled simultaneously. However, there currently is no approach that can create variants with a selection of variable assets in various versions. This chapter introduces an integrated approach to derive variants with information regarding variability in space and time. The approach combines HFMs (see Chapter 4) with an extension of delta modeling (see Chapter 5) to allow for the derivation of concrete software systems from an SPL or a SECO configuring both features and versions. In addition, an algorithm is provided to automatically select valid combinations of versions for a pre-configuration of features.*

6.1. Variant Derivation Cannot Handle Variability in Time

As explained in Chapter 4, different versions may have to be maintained for compatibility reasons and may introduce dependencies on and incompatibilities with other versions. This makes the notion of variability in time relevant for configuration. However, versions cannot be adequately captured in common feature models nor incorporated into variants of a software family by a variability realization mechanism. To remedy this problem, Chapter 4 introduced Hyper-Feature Models (HFMs) and a version-aware constraint language. In HFMs, each feature specifies multiple *feature versions* each reflecting one particular state of evolution of the realization assets associated with the feature. Feature versions are arranged along development lines as increments to the predecessor version, which also supports branching. Whether a version of a realization asset is made explicitly available on the conceptual level of the HFM mostly depends on whether that revision is made available to customers (see Section 4.4). Furthermore, Chapter 5 extended delta modeling as variability realization mechanism to cope with variability in time by supporting dedicated evolution delta modules and generating suitable delta languages for newly created and changing source languages. This makes delta modeling available for all artifacts affected by variability.

To derive variants from an SPL or a SECO that contain both variability in space and time, a suitable variant derivation mechanism is required that can assemble realization assets associated with features and their respective versions. This is relevant for SPLs and SECOs, as not necessarily all customers of products update to the most recent version immediately or want to upgrade all variable assets at once (e.g., due to budget constraints or internal dependencies on older versions of individual assets). This problem is especially present in SECOs: For one, there is usually no central instance controlling evolution, which results in multiple versions of variable assets with interdependencies. Furthermore, products may be configured directly by users, which may yield constellations of feature versions that have not been anticipated explicitly. Hence, a mechanism to express variability in space and time and to incorporate this knowledge into the variant derivation process is required. However, there currently exists no approach to derive variants of software families with both variability in space and time.

This chapter presents an integrated approach to derive variants with information regarding variability in space and time. HFMs are used to conceptually represent variability in space and time as features with versions. It is the explicit goal to handle the evolution of individual variable assets of the software family, but not to handle the case of evolving the variant space spanned by variability in space, especially not when reducing configuration options. Hence, the SPL evolution categories of *refactoring* and *generalization* from [TBK09] are supported but not *specialization* or *arbitrary edits*. However, in contrast to [TBK09] where modifications are defined for feature models, the changes are performed on realization assets associated with features and feature versions.

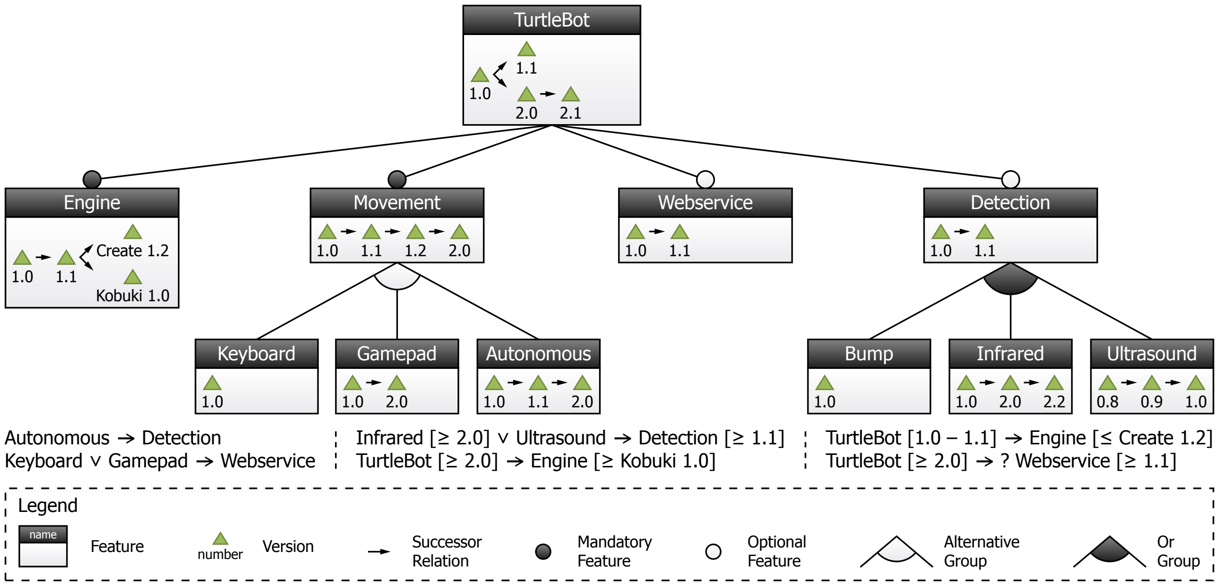


Figure 6.1.: Graphical representation of the HFM for the TurtleBot driver software with feature versions as configurable elements and version-aware constraints.

These problems are illustrated using the running example of the TurtleBot driver introduced in Chapter 1. The respective HFM is depicted in Figure 6.1. Furthermore, maintaining versions of the variable functionality and allowing for different combinations thereof in variants is necessary as not all instances of the driver deployed to multiple TurtleBots are updated simultaneously. In addition, some of the robots cannot be updated completely, e.g., due to the constraints imposed by the iRobot Create engine. Furthermore, extensions to individual TurtleBots in hardware and software depend on old versions of some features. This creates a need for variant derivation including variability in space and time. Based on the problems illustrated in these scenarios, R3 of Section 3.4.2 is refined to the following requirements for a variant derivation procedure capable of creating variants with various variable assets in different versions:

R3.1 **Employ Variability Model**: Define configurations utilizing the conceptual level of a variability model.

R3.2 **Automated Application Order**: Automate the determination of application order of changes as far as possible.

R3.3 **Simplified Configuration**: Support for users in selecting suitable configurations on a conceptual level.

R3.4 **Product Creation**: Derive concrete products of the software family.

R3.5 **Unanticipated Products**: Support for creation of products encompassing variability in time that are valid but have not explicitly been thought of before.

6.2. Associating Features and Feature Versions with Delta Modules

When using delta modeling in conjunction with feature models, application conditions (see Section 3.2.2) are used to determine under which constellation of features in a configuration a certain delta module has to be applied. Upon variant derivation, all delta modules satisfying a logical expression of features in the respective configuration are collected. When using HFMs instead of common feature models, this procedure differs significantly, as delta modules are assigned to features *and versions* for the purpose of configuration *and evolution*, respectively. HFMs allow for modeling of variability in space and time in the form of features and feature versions on a conceptual level. Both types of variability can principally be captured as transformations on realization artifacts using delta modeling [SBB$^+$10, DS11, KLL$^+$14] with explicit evolution delta modules. Hence, to establish a relation between an HFM and delta modules, an explicit mapping between the respective elements has to be introduced.

The general nature of these mappings is visualized in Figure 6.2, where the feature `TurtleBot` and its versions are mapped to a configuration delta module and various evolution delta modules. Version *2.0* is mapped to multiple delta modules that modify assets in different source languages. Furthermore, a more complex mapping is denoted textually as example.

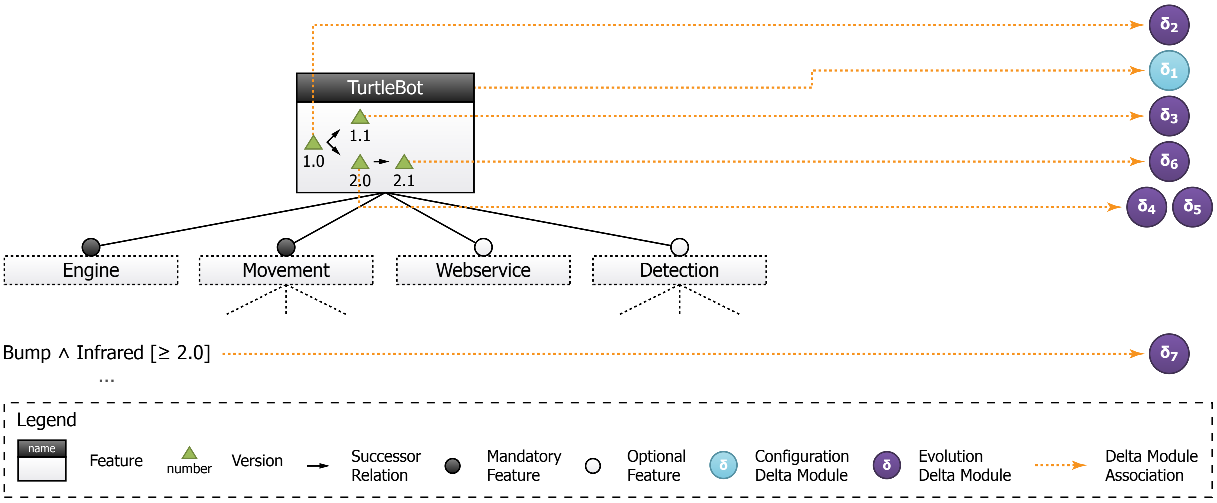


Figure 6.2.: Example of associating delta modules with features and feature versions.

In the most general case, each feature is mapped to a configuration delta module and each feature version is mapped to an evolution delta module. However, there are situations when more complex mappings are required. For one, it may be necessary to assign more than a single delta module to either a feature or a feature version, e.g., when altering realization assets in different source languages, which requires different delta languages. For this purpose, mapping not only to single delta modules but also to sets of delta modules is supported.

Furthermore, more complex conditions may have to be met, which requires logical expressions over features and feature versions. For this purpose, the specification of logical expressions is supported using the same constructs as within the version-aware constraint language (see Definition 18). For example, the expression Bump $\wedge$ Infrared $[\geq 2.0]$ of Figure 6.2 may be mapped to an evolution delta module performing an update to ensure compatibility of the `Bump` sensor and the more recent versions of the `Infrared` sensor. Using logical expressions over features and feature versions subsumes the general case, where only a single feature or version is mapped and, thus, can be used as uniform mechanism for mapping.

In contrast to using application conditions specified within each delta module individually, a separate mapping model is employed that is located external to the delta modules. This has the benefit that the connection between an HFM and the associated delta modules is more easily accessible for both the variant derivation process, as well as for changes and additions to the mapping in the course of evolution. To have a suitable structure for the mapping model, logical expressions over features and feature versions are related to sets of delta modules. This is the opposite direction of application conditions in standard delta modeling, but has the same expressiveness. Formally, a mapping is defined according to Definition 22.

Definition 22: Delta Module Mapping

Let $\mathbb{A}$ be the set of all possible expressions of the version-aware constraint language over $\mathcal{F} \cup \mathcal{V}$ according to Definition 18. A mapping function μ relating version-aware expressions to sets of delta modules can be defined as $\mu : \mathbb{A} \rightharpoonup \mathcal{P}(\Delta)$.

The example of Figure 6.2 can further be expressed in the formal notation of Definition 22 as presented in Formula 7.

Formula 7: Formalization of the mapping function of the example from Figure 6.2.

1. $\Delta = \{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6, \delta_7\}$
2. $\Delta_C = \{\delta_1\}$
3. $\Delta_E = \{\delta_2, \delta_3, \delta_4, \delta_5, \delta_6, \delta_7\}$
4. $\mu(\text{TurtleBot}) = \{\delta_1\}$
   $\mu(\text{TurtleBot } [= 1.0]) = \{\delta_2\}$
   $\mu(\text{TurtleBot } [= 1.1]) = \{\delta_3\}$
   $\mu(\text{TurtleBot } [= 2.0]) = \{\delta_4, \delta_5\}$
   $\mu(\text{TurtleBot } [= 2.1]) = \{\delta_6\}$
   $\mu(\text{Bump} \wedge \text{Infrared } [\geq 2.0]) = \{\delta_7\}$

With the formalization of the mapping function, it is further possible to determine distinct types of mapping entries according to Definition 23.

Pure variability-in-space mapping entries are similar to the application conditions known from delta modeling and are used solely for means of enabling or disabling features. Pure variability-in-time mapping entries are used to update parts of the system in the course of evolution. Hence, both these types of mapping entries address exclusively the dimension of variability in space or time, respectively. In contrast, mixed-mapping entries cross the border between both these dimensions and allow for the specification of changes in associated delta modules that are triggered when a certain combination of aspects regarding variability in space

Definition 23: Delta Module Mapping Entry Types

Let $\mu(\psi_i) = \Delta_S \subseteq \Delta$ be the set of delta modules associated with ψ_i as determined by the mapping function μ. The following types of mapping entries can be distinguished when the respective conditions hold:

1. *Pure variability-in-space mapping entry*
   a) ψ_i and all component expressions are of types 1 or 6–12 of Definition 18
   b) All delta modules of the mapping entry are configuration delta modules.
   $\forall \delta_S \in \Delta_S : \delta_S \in \Delta_C$
2. *Pure variability-in-time mapping entry*
   a) ψ_i and all component expressions are of types 2–12 of Definition 18
   b) All delta modules of the mapping entry are evolution delta modules.
   $\forall \delta_S \in \Delta_S : \delta_S \in \Delta_E$
3. *Mixed-mapping entry* when the conditions for neither 1. nor 2. hold.

and time is present in a configuration. Hence, mixed-mapping entries indicate an interconnection of changes associated with variability in space and time.

In the example formalization presented in Formula 7, the following types of mapping entries can be found: The mapping entry to δ_1 is a pure variability-in-space mapping entry, those to δ_2, δ_3, δ_4/δ_5, δ_6 are pure variability-in-time mapping entries and the one to δ_7 is a mixed-mapping entry.

6.3. Automatically Select Versions to Ease Configuration

Within the presented approach, selecting a conceptual configuration from an HFM is the first step in deriving a variant of a software family. This configuration encompasses elements of variability in space (i.e., features) as well as variability in time (i.e., feature versions). From a conceptual view, selection of functionality in the form of features presumably is more important to stakeholders defining concrete products than settling on a particular version of the feature's realization. However, when defining a configuration of an HFM that is valid with regard to Definition 17, selection of both features and feature versions may be tedious as incompatibilities of feature versions have to be resolved.

To counter this, a procedure is defined to automatically select suitable combinations of feature versions for a valid pre-selection of features. Thus, the effort is reduced from configuring both dimensions of variability in space and time to only configuring variability in space through selecting valid combinations of features similar as with common feature models. Furthermore, it is possible to also pre-select individual feature versions, e.g., to express user preference towards a particular version of a feature. Valid configurations for the remaining required versions are determined automatically.

To achieve this goal, a CSP [Tsa93] is defined on the basis of the formal definition of HFMs and the version-aware constraint language presented in Chapter 4. A CSP solver is used to determine all possible valid configurations for the selected features. During solution of the CSP, an objective function is used to determine the most suitable configuration (see below). The formal steps of this procedure are described in Formula 8.

Formula 8: Procedure for automatic version configuration.

Let $\mathcal{C}_{pre} \subseteq \mathcal{F} \cup \mathcal{V}$ be an incomplete pre-configuration of features and (potentially) versions that satisfies points 1–3 of Definition 17. Further let $\mathbb{C}_{potential} = \{\mathcal{C}_i \mid (\mathcal{C}_i \models \text{HFM}) \wedge (\mathcal{C}_i \models \Psi) \wedge ((\mathcal{C}_i \cap \mathcal{F}) \equiv (\mathcal{C}_{pre} \cap \mathcal{F})) \wedge ((\mathcal{C}_i \cap \mathcal{V}) \supseteq (\mathcal{C}_{pre} \cap \mathcal{V}))\}$ be the set of all configurations potentially used as solution for the automatic configuration procedure (i.e., those valid configurations with the same features as the pre-configuration and the pre-selected versions with further versions for the features previously without versions).

- **Input**
  - HFM: a Hyper-Feature Model
  - Ψ: a set of version-aware constraints
  - $\mathcal{C}_{pre}$: the pre-configuration of features and versions
  - $o : \mathcal{P}(\mathcal{F} \cup \mathcal{V}) \rightharpoonup [0, 1]$: an objective function assigning a quality value to configurations based on the contained versions (with greater values representing higher quality)
- **Process**
  1. Encode HFM, Ψ, $\mathcal{C}_{pre}$ and o as CSP.
  2. Apply CSP solver to solve CSP in order to determine the configuration $\mathcal{C}_{best}$.
- **Output**
  - $\mathcal{C}_{best} \in \mathbb{C}_{potential}$: configuration satisfying all points of Definition 17 and $\forall \mathcal{C}_i \in \mathbb{C}_{potential} : o(\mathcal{C}_{best}) \geq o(\mathcal{C}_i)$ (i.e., having the greatest quality value according to the employed objective function)

To find the most suitable configuration, an objective function is used on the versions selected in the configuration (as selected features are equivalent for all candidate configurations). Various factors may influence the quality value of the objective function used for HFMs:

1. **Novelty**: More recent versions towards the end of a branch are assumed to be preferable over less recent ones as they are more current. Hence, the *novelty* of a selected version is weighted as $n(v) = 1/(\text{SP}(v) + 1)$ with $\text{SP}(v)$ being the length of the shortest path to a version without successor on the same development line as version v.
2. **Importance**: Features closer to the root of the tree spanned by the HFM are assumed to represent more coarse-grain functionality and, thus, to have a larger effect on the overall system than features further down in the tree. Hence, the *importance* of a version's containing feature is weighted as $i(v) = 1/(\text{TL}(v) + 1)$ with $\text{TL}(v)$ being the depth of the feature containing version v in the tree of the HFM.
3. **Inverse Importance**: Features further away from the root of the tree spanned by the HFM are assumed to represent the actual implementation of functionality and, thus, to have a larger effect on the overall system than features further up in the tree that may just serve the purpose of conceptual containers. Hence, the *inverse importance* of a version's containing feature is weighted as $ii(v) = 1/(\text{TL}_{\text{Max}} - \text{TL}(v) + 1)$ with $\text{TL}(v)$ being the depth of the feature containing version v in the tree of the HFM and TL_{Max} being the overall depth of the tree.

These factors may be contradictory in the sense that improving the value for one automatically leads to deterioration of the other. Obviously, this is the case for importance and inverse importance, which are antithetic in nature. However, each of these values may be valid depending on the concrete nature of the targeted software family. Multiple of these factors $a_i(v)$ may be combined to an objective function for a single version as $o_v(v) = a_1(v) * \ldots * a_n(v)$.

The final value for a configuration $\mathcal{C}_i$ is $o(\mathcal{C}_i) = \sum_{v \in \mathcal{C}_i \cap \mathcal{V}} o_v(v)$. The configuration $\mathcal{C}_{best}$ with the greatest value is preferred over the others. In case there are multiple configurations with the highest value, they are perceived as having similar quality and one is selected arbitrarily as solution.

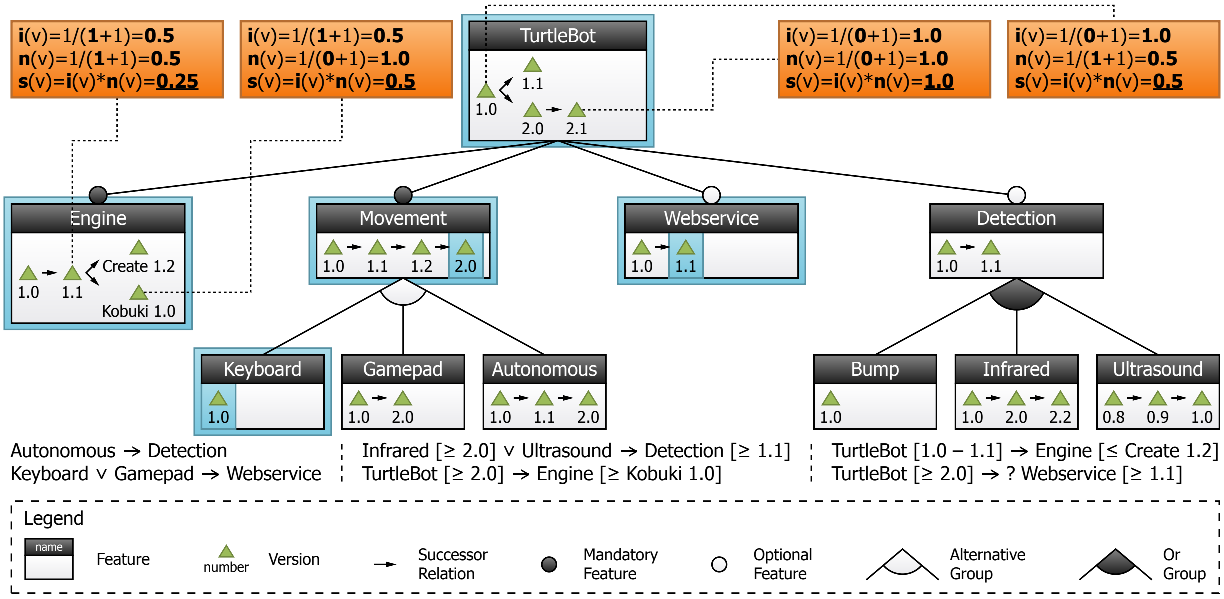


Figure 6.3.: Example of automatic version selection for the pre-configuration {TurtleBot, Engine, Movement, 2.0 (Movement), Keyboard, 1.0 (Keyboard), Webservice, 1.1 (Webservice)} using *importance* and *novelty* as factors for the objective function.

For example, if features closer to the root of the feature model affect the overall architecture of a variant, importance may be used besides novelty as factor of the objective function. In Figure 6.3, the pre-configuration {TurtleBot, Engine, Movement, 2.0 (Movement), Keyboard, 1.0 (Keyboard), Webservice, 1.1 (Webservice)} is extended to a complete configuration automatically. In the example, a selection of versions {2.1 (TurtleBot), Kobuki 1.0 (Engine)} has a value of $1.0 + 0.5 = 1.5$ and, thus, is preferred over, e.g., a selection of versions {1.0 (TurtleBot), 1.1 (Engine)} with a value of $0.5 + 0.25 = 0.75$ as well as all other possible version constellations. Furthermore, the specified version-aware constraints limit the options of possible combinations of versions so that, e.g., the version selection {1.0 (TurtleBot), Kobuki 1.0 (Engine)} is not possible. The automatic version selection procedure respects these constraints.

Similarly, if the actual realization of features is mostly in features further away from the root of the feature model, inverse importance may be used besides novelty as factor of the objective function. In Figure 6.4, the same pre-configuration as in Figure 6.3 is extended to a complete configuration automatically. In the example, a selection of versions {2.1 (TurtleBot), Kobuki 1.0 (Engine)} has a value of $0.33 + 0.5 = 0.83$ and, thus, is preferred over, e.g., a selection of versions {1.0 (TurtleBot), 1.1 (Engine)} with a value of $0.165 + 0.25 = 0.415$ as well as all other possible version constellations. While this result is similar to the one of the example of Figure 6.3, which uses importance and novelty as factors for the objective function, it is different

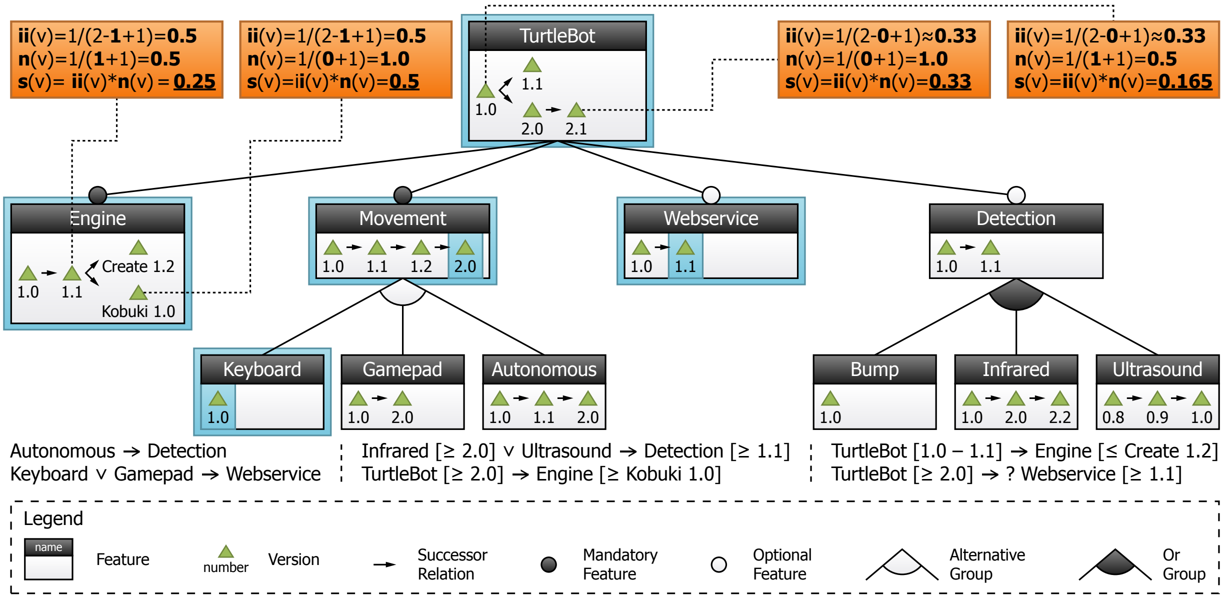


Figure 6.4.: Example of automatic version selection for the pre-configuration {TurtleBot, Engine, Movement, 2.0 (Movement), Keyboard, 1.0 (Keyboard), Webservice, 1.1 (Webservice)} using *inverse importance* and *novelty* as factors for the objective function.

in that it puts more emphasis on the versions of the feature `Engine`, which is located further away from the root of the feature model, than on the feature `TurtleBot`.

To address further needs in the automatic version selection procedure that are more specific to a particular software family, the objective function may be replaced entirely with other metrics of estimating the quality of determined configurations while still reusing the majority of the algorithm.

6.4. Application Order and Implicitly Required Delta Modules

To create a software variant for a specific configuration from an HFM, it is necessary to determine all relevant delta modules and a suitable sequence in which they can be applied. Formula 9 presents the general algorithm to perform both these procedures and the following paragraphs elaborate on the steps in detail.

6.4.1. Determining Relevant Delta Modules

To create a software variant in delta modeling, it is first necessary to collect all delta modules relevant for the given configuration. For the features of the HFM, this means that the associated configuration delta modules form the initial set of relevant delta modules. This procedure is similar as with common delta modeling, when using a feature model as described in Section 3.2.2. However, when evaluating the selected versions from the specified configuration of the HFM, this procedure has to be extended: As versions are perceived as incremental, every selected version requires its predecessor. In consequence, the evolution delta modules associated with the originally selected version and all its (transitive) predecessors are added to the set of relevant delta modules. Furthermore, mappings may exist that utilize arbitrarily complex application conditions for sets of configuration and evolution delta modules, which are added to the set of relevant delta modules if the configuration satisfies the application condition. Finally, delta modules may explicitly

Formula 9: Algorithm to determine relevant delta modules and a suitable application sequence from a configuration.

- **Input**
  - A valid configuration of features and versions from the HFM.
- **Process**
  1. Determine all relevant delta modules for the configuration.
  2. Form a dependency graph with nodes representing each delta module. Arcs have an "apply after" semantics but, initially, no arcs are present.
     a) Add the explicit requires relations within delta modules as arcs.
     b) Add additional arcs determined from the HFM structure.
  3. Perform a topological sorting to determine one possible application sequence for the delta modules.
- **Output**
  - An ordered set of the delta modules required to create the variant for the input configuration arranged along a valid application sequence.

specify that they require another delta module (e.g., for implementation reasons). The respective (transitively) required delta modules are also added to the set of relevant delta modules.

The set of relevant delta modules for a particular configuration can be defined formally based on the definition of HFMs (see Definition 15) and the mapping function μ (see Definition 22) as described in Formula 10.

6.4.2. Forming a Dependency Graph of Delta Modules

To create software variants with variability in space and time, it is further necessary to establish a concrete sequence of application for the relevant delta modules. For this purpose, a topological sorting is performed on the delta modules that respects the individual constraints of each delta module regarding potential orderings. All order constraints of delta modules may be reduced to an "apply after" relation that states that, for two delta modules δ_A and δ_B, delta module δ_A may only be applied *after* δ_B has already been applied. To apply a topological sorting on basis of a dependency graph, relevant delta modules are perceived as nodes and the "apply after" relations are perceived as arcs between those nodes.

When using HFMs, information from four sources has to be processed to create the "apply after" arcs in the graph of delta modules, denoted as $\delta_A \rightarrow_{aa} \delta_B$, when a delta module δ_A should be applied only after another delta modules δ_B:

1. **Explicit application-order constraints** specify partial orderings over delta modules by defining that a delta module should be applied after a group of delta modules. For example, an application-order constraint may specify that a delta module δ_A should be applied after the delta modules $\delta_{B1} \ldots \delta_{Bn}$. According to Definition 21, this application-order constraint may be captured formally by the function $\mathrm{aoc}(\delta_A) = \Delta_a$ with $\Delta_a = \{\delta_{B1}, \ldots, \delta_{Bn}\}$. For each delta and all its elements in the resulting set of $\mathrm{aoc}(\delta_A)$, a respective "apply after" arc is added to the graph of delta modules, e.g., $\delta_A \rightarrow_{aa} \delta_{B1}$, $\delta_A \rightarrow_{aa} \delta_{B2} \ldots \delta_A \rightarrow_{aa} \delta_{Bn}$.

Formula 10: Formalization of the set of relevant delta modules for a configuration.

Let $\mathcal{C} \in \mathbb{C}$ be the current configuration of an HFM. Furthermore, let $\text{succ}^*(v)$ be the transitive closure over the auxiliary function $\text{succ}(v)$ to determine successors of a version as introduced in Definition 19. Further let $\text{req}^*(\delta)$ be the transitive closure over the function capturing the requires relation as introduced in Definition 21. Also let $\Delta_C \subseteq \Delta$ be the set of all configuration delta modules and $\Delta_E \subseteq \Delta$ the set of all evolution delta modules as introduced in Definition 21. The following are constituents of the set $\Delta_R \subseteq \Delta$ of relevant delta modules for $\mathcal{C}$:

1. Relevant Configuration Delta Modules of Features
   $\Delta_{RC} = \{\delta \in \Delta_r | \mu(f) = \Delta_r \wedge \Delta_r \subseteq \Delta_C \wedge f \in \mathcal{F} \wedge f \in \mathcal{C}\}$
2. Relevant Evolution Delta Modules of Feature Versions
   $\Delta_{RE} = \{\delta \in \Delta_r | \mu(v_1) = \Delta_r \wedge \Delta_r \subseteq \Delta_E \wedge v_1 \in \mathcal{V} \wedge (v_1 \in \mathcal{C} \vee \exists v_2 \in \text{succ}^*(v_1) : v_2 \in \mathcal{C})\}$
3. Relevant Delta Modules with Complex Conditions
   $\Delta_{RA} = \{\delta \in \Delta_r | \mu(e) = \Delta_r \wedge \Delta_r \subseteq \Delta \wedge \mathcal{C} \models e\}$
4. Relevant Delta Modules that are Required Directly or Transitively
   $\Delta_{RR} = \{\delta \in \Delta_r | \delta_2 \in (\Delta_{RC} \cap \Delta_{RE} \cap \Delta_{RA}) \wedge \text{req}^*(\delta_2) = \Delta_r\}$

The set $\Delta_R \subseteq \Delta$ of relevant delta modules is then defined as $\Delta_R = \Delta_{RC} \cap \Delta_{RE} \cap \Delta_{RA} \cap \Delta_{RR}$.

2. **Explicit requires relations** between two delta modules δ_A and δ_B cause that, whenever delta module δ_A is part of a variant, delta module δ_B is also added to the set of required delta modules. Furthermore, the requires relation implicitly demands that delta module δ_A has to be applied after delta module δ_B as it depends on the changes performed in delta module δ_B. According to Definition 21, this requires relation may be captured formally by the function $\text{req}(\delta_A) = \Delta_a$ with $\Delta_a = \{\delta_B\}$. In consequence, for each element in the result of the function $\text{req}(\delta)$ representing the requires relation from delta module δ_A to a delta module δ_B, an “apply after” arc $\delta_A \rightarrow_{aa} \delta_B$ is added to the graph of delta modules.
3. **Implicit dependence of the initial version on its containing feature** is caused by the fact that a feature may only be updated to a particular version if it has been enabled in a variant. In consequence, for an initial version v and its containing feature f, an “apply after” arc is added between all evolution delta modules associated with v and the configuration delta modules associated with f.
4. **Implicit dependence of versions on their predecessor version** is due to the fact that versions of an HFM are perceived as incremental. Hence, the application of evolution delta modules associated with a more recent version dictates that the evolution delta modules of previous versions on the same development line have to be applied before. In consequence, for each version v_1 with an immediate predecessor v_2, “apply after” arcs are added between all delta modules associated with v_1 and those with v_2.

The first two sources of “apply after” arcs are similarly present in standard delta modeling (see Section 3.2.2). However, the last two sources of “apply after” arcs are unique to the approach of this thesis, as they represent constraints of possible orderings of delta modules that are implicitly imposed by the structure of the HFM. Based on Definition 15, Definition 22 and Formula 10, the four sources for “apply after” arcs can be described formally in Formula 11.

It is an explicit design decision not to impose an implicit order on different features. Neither feature models nor HFMs capture in their tree structure an order of the features other than their hierarchical relation. Furthermore, potential constraints on the order of delta modules

Formula 11: Sources for "apply after" arcs in the graph of relevant delta modules.

Let $\mathcal{C} \in \mathbb{C}$ be the current configuration of an HFM. Furthermore, let Δ_R be the set of relevant delta modules for the configuration as determined by Formula 10. When perceiving the delta modules of Δ_R as nodes of a graph, the following are sources for "apply after" arcs, which are denoted as $\delta_A \rightarrow_{aa} \delta_B$:

1. Explicit application-order constraints.
   $\forall \delta_1, \delta_2 \in \Delta : \text{aoc}(\delta_1) = \Delta_a \wedge \delta_2 \in \Delta_a \rightarrow \delta_1 \rightarrow_{aa} \delta_2$
2. Explicit requires relations within delta modules.
   $\forall \delta_1, \delta_2 \in \Delta : \text{req}(\delta_1) = \Delta_r \wedge \delta_2 \in \Delta_r \rightarrow \delta_1 \rightarrow_{aa} \delta_2$
3. Implicit dependence of the initial version on its containing feature.
   $v, f \in \mathcal{C} \wedge \Upsilon(f) = V \wedge v \in V \wedge \text{pred}(v) = \epsilon \wedge \mu(f) = \Delta f \wedge \mu(v) = \Delta_v \rightarrow (\forall \delta_v \in \Delta_v, \forall \delta_f \in \Delta_f : \delta_v \rightarrow_{aa} \delta_f)$
4. Implicit dependence of versions on their predecessor version.
   $v_1, v_2 \in V \wedge \text{pred}(v_2) = v_1 \wedge (\exists v_3 : v_3 \in \mathcal{C} \wedge \text{pred}^*(v_3) = V_p \wedge v_1, v_2 \in V_p) \wedge \mu(v_1) = \Delta_{v1} \wedge \mu(v_2) = \Delta_{v2} \rightarrow (\forall \delta_{v1} \in \Delta_{v1}, \forall \delta_{v2} \in \Delta_{v2} : \delta_{v2} \rightarrow_{aa} \delta_{v1}))$

of different features stem from the realization assets but not from the conceptual level of a variability model. Thus, no (arbitrary) implicit constraints are imposed on the order of delta modules of different features. Instead, explicit requires relations have to be formulated that are translated within the delta modules if required.

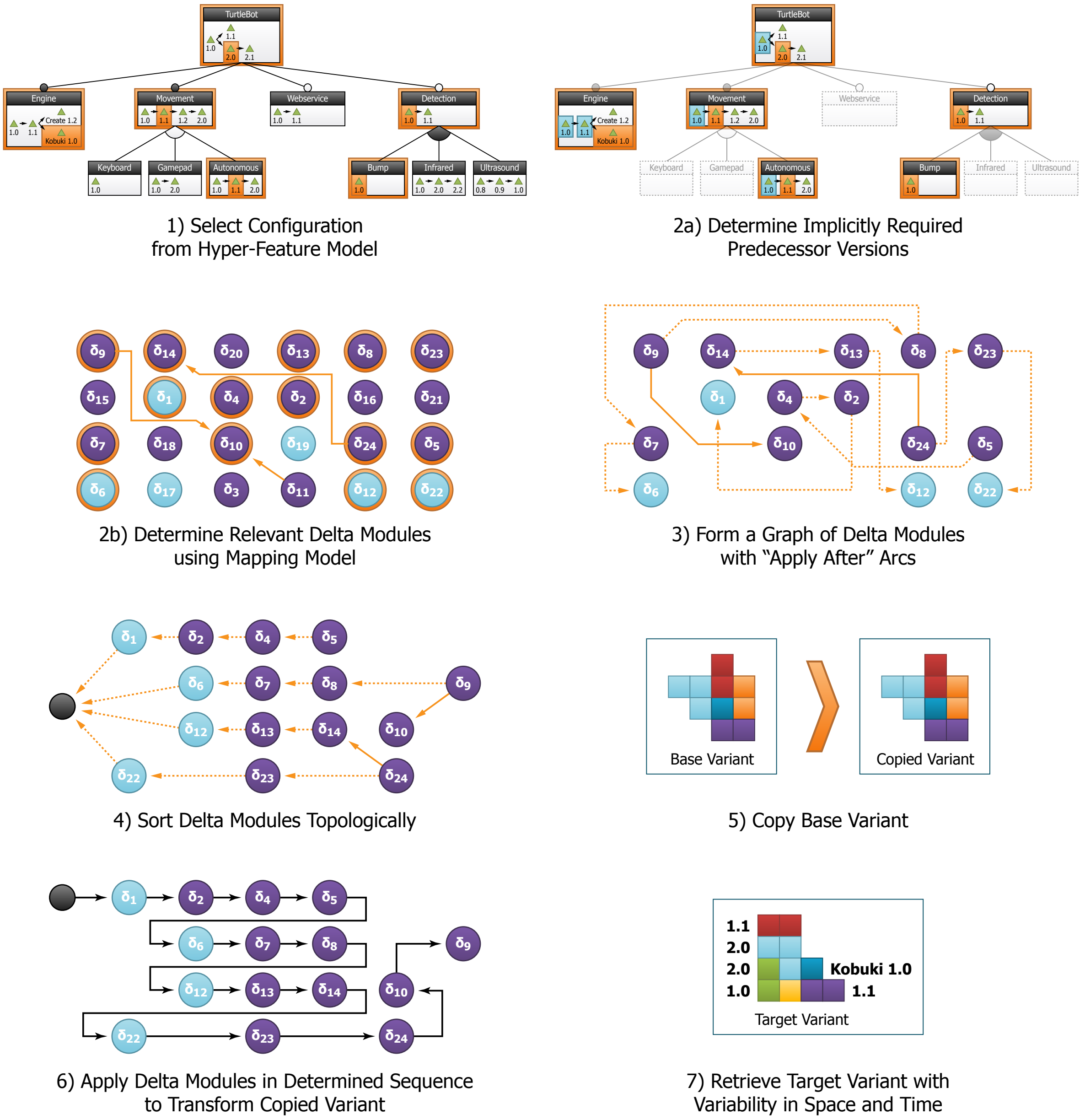
The resulting dependency graph of delta modules and "apply after" arcs is acyclic, as circular dependencies are prohibited, so that it can be subjected to a topological sorting in order to establish a concrete application sequence for the delta modules.

6.4.3. Performing a Topological Sorting of Delta Modules

By processing the aforementioned four sources for constraints on possible orderings of delta modules and by adding the respective "apply after" arcs to the graph of delta modules, sufficient input for establishing an application sequence is created. For this purpose, a topological sorting is performed on the delta modules acting as nodes of the graph. The result is one valid sequence of the delta modules that respects all individual constraints on potential orderings including those imposed by the structure of the HFM. The determined sequence of delta modules may then be used to apply the delta modules in order to retrieve a variant for the initially specified configuration (see Section 6.5).

6.5. Generating Variants with Versions of Variable Assets

With a given conceptual configuration consisting of both features and feature versions of an HFM, it is possible to employ the variant derivation mechanism to generate an actual product of the software family. For this purpose, the variant derivation procedure locates all delta modules associated with the configuration, sorts them topologically and employs the established sequence of delta modules and to apply all relevant delta modules and their operations. The result of this process is a variant of the software family that contains the configured functionality (features) in the selected revisions (feature versions). Figure 6.5 depicts this process.



1) Select Configuration from Hyper-Feature Model

2a) Determine Implicitly Required Predecessor Versions

2b) Determine Relevant Delta Modules using Mapping Model

3) Form a Graph of Delta Modules with "Apply After" Arcs

4) Sort Delta Modules Topologically

5) Copy Base Variant

6) Apply Delta Modules in Determined Sequence to Transform Copied Variant

7) Retrieve Target Variant with Variability in Space and Time

Figure 6.5.: Steps to derive a variant with variability in space and time. Only step 1) has to be performed manually.

Creating a variant with variability in space and time encompasses 7 steps where only the first one has to be performed manually:

1) A configuration of an HFM is defined by a user (may partially be automated by the algorithm in Section 6.3).
2) The relevant delta modules are determined by collecting delta modules of features, feature versions and implicitly required predecessor versions. Logical expressions of the mapping model are evaluated with regard to the provided configuration to find additional delta modules.
3) The delta modules are perceived as nodes of a graph and "apply after" arcs between the delta modules are derived from the structure of the HFM.
4) The delta modules are sorted topologically using the dependency graph of delta modules and the "apply after" arcs in order to determine a suitable application sequence.
5) The base variant of the software family is copied including all realization assets to serve as starting point for the subsequent transformations.
6) Delta modules are applied using the determined application sequence and the copied variant is transformed to the target variant by sequentially applying all delta operations.
7) The resulting target variant of the software family contains the realization artifacts in a state that represents the selected features (variability in space) and feature versions (variability in time).

This mechanism can also be used to perform updates of entire products by selecting a configuration containing similar features but more recent versions and deriving the respective variant. An equivalent procedure may be used to revert a product to a previous revision.

6.6. Case Study

The applicability of the approach introduced in this chapter is demonstrated in a case study on the configurable driver software for the TurtleBot domestic robot as introduced in Section 1. The driver has been developed over the course of approximately 1.5 years and has undergone multiple evolution cycles that can roughly be grouped into 4 stages as explained in Chapter 1. The driver software has its most recent HFM depicted in Figure 6.6 with annotations for the evolution iteration that introduced a certain feature, feature version or version-aware constraint.

The TurtleBot driver and its evolution, including the modifications on realization assets, were modeled using the approach presented in this chapter. The case study focuses the combination of the conceptual part of variability in space and time with the respective effects on realization assets. For the latter, variability in the Java source code of the driver software is inspected. As the source code is structured into multiple projects, which may be required depending on the presence of certain features and feature versions, variability of the affected Eclipse projects is also inspected. Within the case study, the conceptual knowledge about a configuration containing features and feature versions is used to derive concrete variants for the realization assets. The goal of the case study is to determine whether the presented approach meets the requirements for variant derivation with variability in space and time posed in Section 6.1.

The concepts of this chapter were implemented using model-based software development (see Chapter 7). For this purpose, a metamodel was implemented to associate logical expressions of features and feature versions with delta modules according to the definition in Section 6.2. Furthermore, the variant derivation procedure was implemented to build a graph of delta modules and include the "apply after" arcs stemming from the HFM when collecting and sorting delta modules.

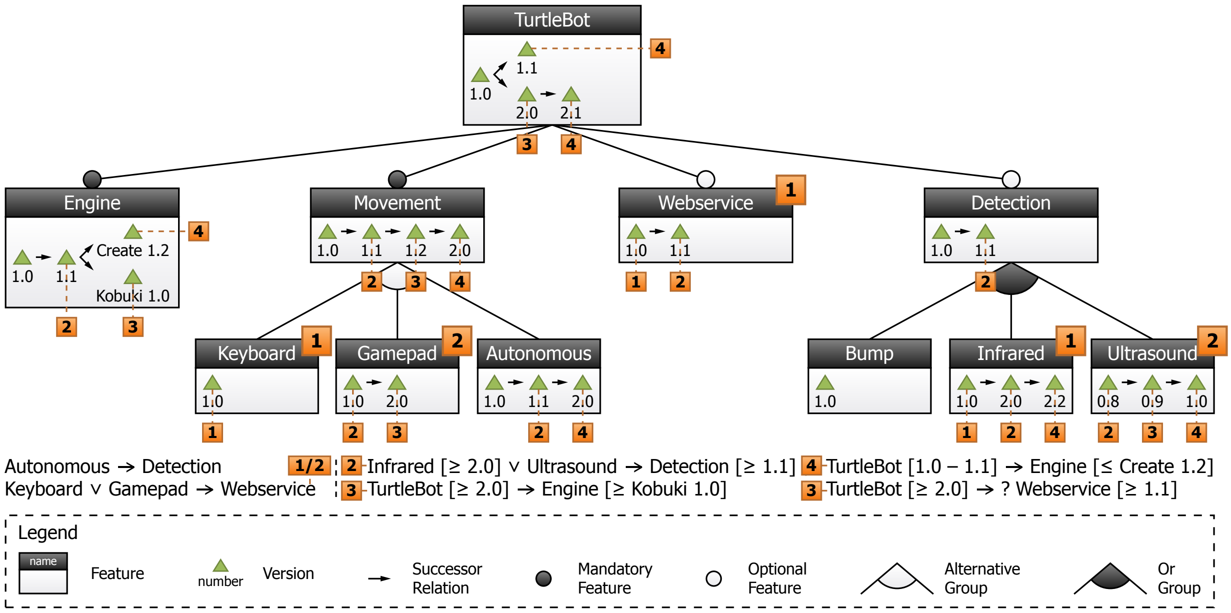


Figure 6.6.: HFM representing the most recent state of evolution of the TurtleBot driver software. Features, feature versions and version-aware constraints are annotated with the evolution stage they were introduced in.

On a conceptual level, aspects of variability in space and time are those represented in the HFM of Figure 6.6. The realization assets of the driver addressed in the case study consist mostly of Java source code. As they are subject to variability, they are altered by respective delta modules. For example, Listing 6.1 shows an example of a delta module to enable the feature `Gamepad` in a delta language for Java created with the concepts of Chapter 5. A requires relation specifies dependency on the class `Movement` used in the modification process (Line 3). A call to a delta operation creates the class `Gamepad` (Lines 6–8) and another one sets `Movement` as super class of `Gamepad` (Line 14). Identifiers are employed to reference classes used in this process (Lines 5, 10–11). The further creation of implementations for various methods is omitted in this example (Line 16).

However, the respective files are organized in Eclipse projects, which specify setup information in XML files[1]. This includes dependencies on other projects and required libraries, which are subject to variability in space, depending on the selected features, as well as variability in time when evolving dependencies of assets. For example, the aforementioned implementation of the class `Gamepad` further requires a class library for the gamepad driver to be available on the classpath of the containing Eclipse project. For this purpose, a second delta module modifies the project setup in a dedicated delta language, again created with the concepts of Chapter 5 (see Listing 6.2). A requires relation is established for the modified asset (Line 3) and calls to delta operations to add the required files for the class library to the project setup are specified (Lines 6–8).

To handle variability in all affected realization assets, delta languages for Java and the XML-based .project and .classpath files of Eclipse were created. With these languages, a total of 46 delta modules were realized, out of which 15 were configuration delta modules and 31 were evolution delta modules. Examples of configuration delta modules have already been presented in Listing 6.1 and Listing 6.2. An example of an evolution delta module is depicted in

[1] Furthermore, various artifacts documenting safety in languages such as SFT, CFD, CL or the GSN are supplied with the TurtleBot driver. These artifacts are excluded from this case study but are explained in Section 8.1.

```
configuration delta "Gamepad_Java"
  dialect <http://www.emftext.org/java>
  requires <../src/eu/vicci/turtlebot/Movement.java>
{
  Package p = <package::eu.vicci.turtlebot>;
  createClass("public class Gamepad {
                  //...
              }", p);

  Class gamepad = <class::eu.vicci.turtlebot.Gamepad>;
  Class movement = <class::eu.vicci.turtlebot.Movement>;

  //Set Movement as super class of Gamepad
  setSuperClassOfClass(movement, gamepad);

  //...
}
```

Listing 6.1: Example of a configuration delta module that enables the feature `Gamepad` in Java source code.

```
configuration delta "Gamepad_Project"
  dialect <http://vicci.eu/eclipseproject/1.0>
  requires <../.classpath>
{
  //Add dependencies on gamepad driver class library
  addRequiredLibrary("lib/lwjgl2.8.5/jar/lwjgl.jar");
  addRequiredLibrary("lib/lwjgl2.8.5/jar/lwjgl_util.jar");
  addRequiredLibrary("lib/lwjgl2.8.5/jar/jinput.jar");
}
```

Listing 6.2: Example of a configuration delta module that enables the feature `Gamepad` in the Eclipse project setup.

Listing 6.3, which performs the changes associated with updating feature `Engine` from version *1.1* to version *Create 1.2*. First, the class `Engine` is renamed to `CreateEngine` (Line 10). Then, a new super class (again called `Engine`) is extracted, which includes the definition of the `drive(..)` method (Line 13). Finally, the newly created class is made `abstract` (Line 17).

The delta operations used for configuration in Java encompassed, among others, adding and removing classes, methods and fields; modifying statement lists within method bodies; as well as toggling the `abstract` modifier for classes. Those for project configuration consisted of adding and removing dependencies to libraries and other Eclipse projects. Each feature had at least one configuration delta module assigned to it. Most features required only to modify Java source code. However, in order to enable the feature `Gamepad`, it was necessary to modify Java source code as well as the setup of the underlying Eclipse project to reference the files of a class library containing the gamepad driver. Due to this reason and the different delta languages required for the source languages, the feature `Gamepad` was realized in two individual configuration delta modules. The respective delta modules have already been presented in Listing 6.1 and Listing 6.2.

The delta operations used for evolution in Java encompassed, among others, extracting methods, interfaces and super classes; renaming classes, methods and fields; as well as adding and removing parameters and arguments of methods and method calls, respectively. No dedicated evolution delta operations were required for altering the project setup in addition to the delta operations already defined for configuration. At least one individual evolution delta module was assigned to

```
evolution delta "Engine_Create 1.2_Java"
  dialect <http://www.emftext.org/java>
  requires <../src/eu/vicci/turtlebot/Engine.java>
{
  Class engine = <class::eu.vicci.turtlebot.Engine>;
  Method driveMethod = <method::eu.vicci.turtlebot.Engine#drive()>;

  //Rename Engine to CreateEngine
  renameNamedElement("CreateEngine", engine);

  //Extract super class Engine from CreateEngine
  extractSuperClass("Engine", engine, [driveMethod]);

  //Make new Engine abstract
  Class newEngine = <class::eu.vicci.turtlebot.Engine>;
  setAbstractModifier(true, newEngine);
}
```

Listing 6.3: Example of an evolution delta module performing an update of feature `Engine` to version *Create 1.2*.

each feature version with only one exception: In versions *1.1* and *2.1* of `TurtleBot`, a defect in the base code for both branches was fixed. To make it available for both branches, two separate versions had to be created in the HFM. However, the changes to be performed were identical in both cases, so that the same evolution delta module could be used for both versions.

For the case of the branching versions *Create 1.2* and *Kobuki 1.0* of feature `Engine`, preparatory changes common to both versions, as well as changes individual to each version, had to be performed. The common changes were specified in a separate evolution delta module that was required by the evolution delta modules of either one of the versions to avoid duplication of calls to delta operations. Among others, version *2.0* of `Gamepad` was associated with multiple evolution delta modules as the Java source code was altered and the dependency on the gamepad driver was evolved to use a more recent version of the respective class library.

In addition to the pure variability-in-space and pure variability-in-time mapping entries, there were cases in which a more complex association was necessary. For example, one evolution delta module was created to ensure compatibility, when both the `Bump` sensor and the `Infrared` sensor at least at version *2.0* are used in conjunction. For this case, the logical expression $\text{Bump} \land \text{Infrared}\ [\geq\ 2.0]$ was associated with an evolution delta module updating Java code of multiple classes, which constitutes a mixed-mapping entry, as it combines aspects of variability in space and time.

It was possible to express all changes required for configuration and evolution of individual features of the TurtleBot driver using HFMs and configuration/evolution delta modules. With the specified HFM, it is possible to define concrete products of the driver software in terms of valid configurations. The ordering of delta modules imposed by the structure of the HFM sufficiently captured the dependencies that otherwise would have had to be specified explicitly. Furthermore, as no dependencies of delta modules associated with features were imposed implicitly, the ordering did not interfere with dependencies of delta modules specified explicitly due to implementation concerns, which is an orthogonal challenge discussed in detail in [KAuR$^+$09].

Evolutionary changes cutting across multiple features, such as API changes, could be represented by individual versions of the affected features and version-aware constraints. Furthermore, their manifestation could be captured within evolution delta modules. However, the expressiveness of delta languages used to perform modifications is principally defined by the creator of the delta language. The limitations imposed by the intentionally reduced expressiveness of configuration delta operations posed no problem in practice when enabling/disabling features. However, anticipating all evolution delta operations needed to perform changes associated with variability in time proved difficult in practice, especially for Java. In consequence, the set of evolution operations was extended iteratively, which may prove complicated in the future for users of delta languages. However, the language creation infrastructure of Chapter 5 allows for the extension of existing delta languages.

The automatic version selection procedure of Section 6.3 was able to complete pre-configurations consisting of selected features with suitable versions for the features. For example, for the pre-configuration {TurtleBot, Engine, Movement, 2.0 (Movement), Keyboard, 1.0 (Keyboard), Webservice, 1.1 (Webservice)}, the following 6 selections of versions obeying the semantics of the HFM, as well as the specified version-aware constraints, are possible for the `TurtleBot` and the `Engine`, respectively (in descending order of quality when using novelty and importance as objective function): {2.1 (TurtleBot), Kobuki 1.0 (Engine)}, {2.0 (TurtleBot), Kobuki 1.0 (Engine)}, {1.1 (TurtleBot), 1.1 (Engine)}, {1.0 (TurtleBot), 1.1 (Engine)}, {1.1 (TurtleBot), 1.0 (Engine)}, {1.0 (TurtleBot), 1.0 (Engine)}. The automatic version selection mechanism correctly determined {2.1 (TurtleBot), Kobuki 1.0 (Engine)} as the most suitable configuration due to the quality value assigned by the objective function.

The variants created with the procedure described in this chapter consisted of explicitly anticipated version constellations (i.e., those of compatible versions at the end of evolution iterations) as well as unanticipated, yet valid, constellations with regard to the HFM and the version-aware constraints (e.g., using `Autonomous` *1.1* with the more recent `Movement` *1.2*). The integrity and validity of the created variants was checked through functional tests and manual code inspection which revealed no defects caused by the variant derivation procedure.

These findings of the case study allow for determining in how far the approach of this chapter satisfies the requirements posed in Section 6.1. As HFMs are utilized to express configuration knowledge on a conceptual level and to define configurations, `R3.1` is satisfied. The automatic configuration procedure of Section 6.3 was able to determine the most suitable configuration for variability in space and time for given pre-configurations, so that `R3.2` is satisfied. For a topological sorting of delta modules, explicitly specified requires relations and application-order constraints were obeyed. Furthermore, a valid application sequence for evolution delta modules could be determined semi-automatically from the structure of the HFM by creating and evaluating a graph of delta modules containing “apply after” arcs, so that `R3.3` is satisfied. Furthermore, it was possible to use the approach to derive variants with features in different combinations of versions to include aspects of variability in space and in time, which satisfies `R3.4`. Finally, it was possible to create products with combinations of feature versions that never existed during the original development, but constitute valid configurations and, hence, might be present in a software family such as a SECO, so that `R3.5` is satisfied.

6.7. Demarcation from Related Work

Variability in time (evolution) in software families is considered an important topic [PCA+13, LSB+10, NTS+11, SHA12] and various taxonomies of changes exist [SE08, TBK09, EBLSP10]. Yet, there is no integrated approach for SPLs or SECOs to conceptually capture aspects of variability in space and in time, to model the changes associated with them and to derive respective variants similar to the procedure presented in this thesis. However, there are approaches that can be employed for some parts of this process, which, thus, are related work.

Some of these approaches have been discussed in relation to the individual contributions of Chapter 4 and Chapter 5, respectively. However, in this section, related work is inspected in relation to the variant derivation procedure for variability in time presented in this chapter and discussed with regard to the requirements of Section 6.1 in order to demarcate the concepts of this chapter from the respective approaches. Table 6.1 contrasts the presented variant derivation procedure with all related approaches.

As either one of the approaches addresses a different combination of aspects from a variability model, a variability realization mechanism and a variant derivation procedure, it is not possible to devise discrete categories. In consequence, the following paragraphs

	R3.1 Employ Variability Model	**R3.2 Simplified Configuration**	**R3.3 Automated Application Order**	**R3.4 Product Creation**	**R3.5 Unanticipated Products**
Variant Derivation Procedure for Variability in Time	+	+	+	+	+
Feature Models [KCH+90, CHE05]	+	o	-	-	-
Feature-Driven Versioning [ME08]	+	o	-	-	-
Formalized Variability Evolution [BKS12]	o	o	-	o	o
FeatureHouse [AKL13]	o	-	+	+	o
Feature Packs [KBS14]	o	-	(+)	o	+
Common Variability Language (CVL) [HMPO+08]	+	-	-	+	o
Variability Modeling Language Family (VML*) [SLFG09, ZSS+10]	o	-	-	+	o
Delta Modeling [SBB+10, DS11]	o	-	+	+	o
Aspect-Oriented Programming [KLM+97]	o	-	-	o	o
Domain-Specific Model Transformation [RW11]	-	-	-	o	o
General Model Transformation (QVT, ATL, ETL etc.)	-	-	-	o	o
Change-Oriented Programming (ChOP) [EVC+07, ECHD09]	o	-	+	o	+
Configuring Versioned Software Products [CW96, CW98]	-	-	-	+	+
Invasive Software Composition (ISC) [Aßm03]	-	-	-	o	o

Table 6.1.: Comparison of the variant derivation procedure for variability in time of Chapter 6 with related approaches using the requirements of Section 6.1.

discuss the approaches roughly in order from a conceptual modeling of variability to a manifestation of variability in realization assets.

Kang et al. [KCH$^+$90] introduce feature models and **Czarnecki et al.** [CHE05] extend them to attributed feature models. Theoretically, both these approaches may be used to model feature versions on a conceptual level: For one, common feature models may create dummy features in an alternative group beneath a particular feature. Furthermore, attributed feature models may introduce a feature attribute for versions with all possible versions in the domain of that attribute. Section 4.6 elaborates on both these procedures in detail and argues that they are inadequate for capturing versions on a conceptual level because they neither offer dedicated language constructs to separate the different concerns of variability in space and time nor allow for modeling branches of versions. Furthermore, neither of the approaches specifies variability realization mechanisms or variant derivation procedures.

Due to their foundation as variability models, both approaches fulfill `R3.1`. Benavides et al. and Sotani et al. introduce procedures to determine configurations from user preferences [BTRC05, SAH$^+$11] that may be used in combination with feature models to ease configuration so that `R3.2` is partially satisfied. Neither approach is concerned with manifesting changes in realization assets so that `R3.3` is unsatisfied. As a direct result, no products of the software family can be created so that `R3.4` is unsatisfied. Consequently, unanticipated products cannot be created leaving `R3.5` unsatisfied.

Mitschke and Eichberg [ME08] introduce *feature-driven versioning*. They extend each feature of a feature model with two version numbers: A *feature logical version* signals the revision of the sub-branch beneath a feature and is increased when the structure of the feature model underneath that feature changes. A *feature container version* represents the revision of the realization assets associated with a feature and is increased when one of these assets is modified in a significant way. Within the approach, versions are represented as mere numbers reflecting *that* an artifact was modified but they do not capture *how* the respective artifact changed. Furthermore, there is exactly one of these versions for each feature so that feature versions cannot be used as configurable units.

As the approach presents a variability model, `R3.1` is satisfied. Furthermore, as the approach is based on feature models, principally, similar procedures to ease configuration as those described in [BTRC05, SAH$^+$11] may be used. However, these procedures do not consider the specified versions, so that `R3.2` is satisfied only partially. The manifestation of changes in realization assets is out of scope of the approach so that `R3.3` is unsatisfied. Consequently, product creation is not possible, so that `R3.4` is unsatisfied. Unanticipated products are not supported, so that `3.5` is unsatisfied.

Brummermann et al. [BKS12] introduce *formalized variability evolution* using *configuration modeling* (an extension to decision modeling [MA02]) to support variability in time in SECOs through partial configurations, which carry default values that may be completed and altered by (chains of) customers. Furthermore, they provide default constraints for a software family but permit users to override and even remove them if suitable for their use case. They call this concept *metavariability* as they make (parts of) the variability model itself subject to variability. With their formalism, they can compare configurations changed due to evolution and detect inconsistencies.

Even though they use partial distributed variability information, a dedicated variability model is not part of the approach, so that `R3.1` is unsatisfied. Even though they do not provide procedures to support the configuration process, they describe a procedure that allows for eased transition from one configuration to another in the course of evolution, so that `R3.2` is satisfied partially. There is no information on how the changes associated with the respective configuration values manifest in realization assets and, in consequence, there is no automatic order to apply these changes so that R3.3 is unsatisfied. The described procedure is used to explicitly manage variability. However, only the conceptual configuration procedure is described, but not the creation of products, so that R3.4 is satisfied only partially. Finally, the approach may principally be used to realize unanticipated products, but the expressiveness to realize different variation points is only hinted so that R3.5 is satisfied only partially.

Apel et al. [AKL13] define FeatureHouse to create languages for capturing changes related to variability. For this purpose, a more coarse-grain variant of an Abstract Syntax Tree (AST) is defined as a Feature Structure Tree (FST) whose elements then may be modified by various operations. The provided operations are based on FOP and utilize superimposition. With the operations, it is possible to handle changes associated with variability in space. However, due to their compositional nature, they cannot model arbitrary evolutionary changes as part of variability in time.

Even though FOP may principally be combined with a variability model, FeatureHouse does not directly integrate it so that R3.1 is only partially satisfied. No procedure is provided to simplify configuration so that R3.2 is unsatisfied. Feature modules specified with the languages of FeatureHouse may specify order constraints that are utilized to determine an adequate application order, which satisfies R3.3. Applying the feature modules in that order allows creation of products of a software family, which satisfies R3.4. However, due to the compositional nature of FOP, it is principally possible to create unanticipated products, but removal of elements proofs complicated, so that R3.5 is satisfied only partially.

Keunecke et al. [KBS14] define *feature packs* as a technical solution to handling variability of SECOs through components that include fractions of a variability model. Variability information is encoded into manifest files declaring metadata of the respective components and combined upon discovery to form a partial representation of the configuration knowledge and to determine whether components may be combined. On a technical level, this procedure may be used to address concerns of variability in time. However, it imposes a component-based architecture on realization assets. Furthermore, features are assumed to be contained entirely within a single component. Hence, the approach is not suitable in general.

Feature packs only carry partial variability information, but do not utilize a dedicated variability model, so that R3.1 is satisfied only partially. Being a technical realization of variability in space and time, easing the configuration process is out of scope for the approach, so that R3.2 is unsatisfied. As features are assumed to be encapsulated completely within a single component, there is no need to apply changes associated with features, so that there is no automatic procedure to determine their order. In consequence, R3.3 is perceived as being fulfilled. Despite its foundation as a variability management approach, feature packs do not offer facilities for product creation, so that R3.4 is satisfied only partially. Finally, creation of unanticipated products is possible in the sense that new components are added, possibly replacing others, so that R3.5 is fulfilled.

Haugen et al. [HMPO^{+}08] introduce the Common Variability Language (CVL) in order to standardize variability modeling. They define *Variability Specifications (VSpecs)* to conceptually capture configuration knowledge and provide a set of standard operations to manipulate realization assets of a software family in order to manifest changes associated with variability. Variant derivation facilities may be employed to derive variants associated conceptual configurations encompassing variability in space and, to a lesser degree, variability in time (see Section 4.8).

Due to the use of VSpecs as variability models, R3.1 is fulfilled. However, there is no support in devising configurations so that R3.2 remains unsatisfied. Dedicated support for determining the order in which changes have to be performed seems to be lacking from CVL, so that R3.3 is unsatisfied. Due to the explicit variant derivation facilities provided by the CVL, R3.4 is satisfied. Unanticipated products can be created by adding further configuration options for variability in space but there is no dedicated mechanism to support their creation internally, so that R3.5 is satisfied only partially.

Zschaler et al. [ZSS^{+}10, SLFG09] introduce Variability Modeling Language Family (VML*) as a family of variability modeling languages created by bootstrapping SPL techniques. Every individual language targets one particular source language. The variability languages are created to offer domain-specific transformation operations to model changes associated with variability in space. However, handling variability in time is out of scope of their approach.

Within VML*, feature models are mentioned as a possible variability model, but the approach itself focuses on manifesting changes in realization assets so that R3.1 is satisfied only partially. No particular mechanism exists to ease the configuration procedure, so that R3.2 is unsatisfied. No application order for the various changes is derived automatically, so that R3.3 is unsatisfied. With its foundation as family of languages for variability management, VML* has dedicated support for product derivation so that R3.4 is satisfied. Furthermore, its transformation nature principally allows realization of unanticipated products, so that R3.5 is partially fulfilled.

Schaefer et al. [SBB^{+}10] introduce *delta modeling* to apply changes associated with variability in space in realization assets through transformation. Its concepts are used as basis for the work of this thesis and have been proposed to model variability in time in other work [DS11, KLL^{+}14], but have not been used in an integrated approach for this purpose. Furthermore, there is work on refactoring [SRS13] and evolving [HRRS12] delta-oriented software families. However, it focuses on modifying the relation of delta modules but not the realization assets targeted by delta modules.

Even though it is principally possible to combine delta modeling with a variability model, the discussed work does not do so explicitly leaving R3.1 unsatisfied. On the level of selecting configurations, there is no support to simplify the procedure so that R3.2 is unsatisfied. However, the order of the relevant delta modules can be determined by a topological sorting utilizing the specified application-order constraints so that R3.3 is satisfied. Using the determined sequence to apply the respective delta modules allows for the creation of individual products of the software family, so that R3.4 is fulfilled. Even though standard delta modeling may principally be used to create unanticipated products by adding further delta modules for evolution, the approach has previously not been used in this regard, so that R3.5 is only partially satisfied.

Kiczales et al. [KLM^{+}97] introduce *Aspect-Oriented Programming (AOP)* to increase modularity by promoting strict separation of cross-cutting concerns. For this purpose, a *pointcut* specifies how an *advice* has to be added at particular *join points* to add additional behavior. Furthermore, AOP may be used to handle configuration and evolution of software families [AJC^{+}07, GV09, FCS^{+}08]. With this procedure, realization assets can principally be modified for variability in space and time. However, the compositional nature of the approach makes removal of elements from realization assets complicated.

Even though respective extensions exist, AOP itself does not employ a variability model, so that R3.1 is only partially satisfied. Easing the configuration process is no concern of AOP, so that R3.2 is unsatisfied. Furthermore, possible interactions between multiple advices affecting the same join point are not resolved automatically, so that R3.3 remains unsatisfied. However, AOP may principally be applied to create products of a software family in the sense of a compositional variability realization mechanism even though it does not provide dedicated facilities for the respective procedures, so that R3.4 is satisfied partially. Finally, unanticipated products may principally be created by adding further pointcuts but removal of existing functionality proves complicated, so that R3.5 is satisfied only partially.

Rumpe and Weisemöller [RW11] describe domain-specific model transformation where custom model modification operations are created for a source language given as metamodel. The approach permits arbitrary modifications to models, e.g., for variability in space and time. However, it does not make a distinction between the intentions of configuration and evolution and, thus, bears the risk of unintentionally damaging a variant during configuration by applying operations intended for evolution that affect larger parts of the system.

Within the approach, no dedicated variability model is employed, so that R3.1 is unsatisfied. In consequence, there are no means to simplify configuration, so that R3.2 remains unsatisfied. A procedure to bring the specified transformations in a particular order has to be specified manually, so that R3.3 is unsatisfied. However, the transformations may principally be used to create products of a software family, which satisfies R3.4. Moreover, transformations do not distinguish variability in space and time so that they may be employed to create unanticipated products as well which satisfies R3.5.

General purpose model transformation approaches such as ATL[2] or ETL[3] suffer from similar problems as the domain-specific model transformation.

Ebraert et al. [EVC$^+$07] use Change-Oriented Programming (ChOP) to capture modifications on realization assets performed during evolution in *change objects*. They use information from change objects, such as their interdependencies, to generate a feature model in order to address variability in space [ECHD09]. **Hendrickson and van der Hoek** [HvdH07] formulate *change sets* on software architectures and associate them with features by specifying explicit relationships within the change sets. These procedures are based solely on the manifestation of dependencies in realization assets. In contrast, the approach of this thesis derives dependencies from the structure of an HFM and allows extending them with technical dependencies from realization level in delta modules.

The feature model synthesized from the manifestation of changes is very fine-grained in nature and tightly aligned with the technical realization of the software family, so that it does not fully reflect the intent of capturing configuration knowledge on a conceptual level, which leaves R3.1 partially unsatisfied. Within the approach, no procedure to ease configuration is supplied, so that R3.2 is not satisfied. Using the interdependencies of change objects, it is possible to determine an application order, which satisfies R3.3. However, product creation is only explicitly supported for versions of single systems and not explicitly for variants of software families so that R3.4 is satisfied only partially. However, due to the focus on variability in time, it is possible to create products that have not explicitly been anticipated which satisfied R3.5.

Conradi and Westfechtel [CW96, CW98] configure versioned products to combine variability in space and time on a realization level. They utilize version control systems and regard coherent parts of functionality in realization artifacts as special versions similar to features of software families. Furthermore, they allow for specification of predefined sequences in which configuration may be performed, such as “product first”, “version first” or “intertwined”.

However, their approach operates exclusively on realization artifacts but does not support configuration on a conceptual level by using a variability model, so that R3.1 is unsatisfied. Even though sequences for the configuration may be specified, they are rigid and have to be defined manually, so that R3.2 and R3.3 are unsatisfied. Finally, unanticipated products may be created due to the foundation of the approach in version control systems, so that R3.5 is satisfied.

Aßmann [Aßm03] defines Invasive Software Composition (ISC) as a fragment-based gray-box composition technique. Within the approach, transformations are used to modify *components*

[2] http://eclipse.org/atl

[3] http://eclipse.org/epsilon/doc/etl

within *fragment containers*. Transformations target specific *hooks*, which may be explicitly declared or implicitly defined within components. A *composer* binds hooks by performing those transformations that target a specific hook. ISC defines operations to add, modify and delete elements so that it may be perceived as a technical realization to manifest changes associated with variability in space and time in realization assets.

However, ISC does not utilize a variability model to conceptually capture configuration knowledge, so that R3.1 is unsatisfied. In consequence, it also does not offer any concepts to ease the configuration procedure of variability in space and time leaving R3.2 unsatisfied. Furthermore, no automatic application order can be derived for the changes to be implemented, so that R3.3 is unsatisfied. Even though ISC may be applied to create individual products of a software family, there is no dedicated mechanism to support this procedure, so that R3.4 is unsatisfied. However, due to its foundation as a transformation procedure, ISC may principally be employed to define unanticipated products, which satisfies R3.5 partially.

Finally, there is work on variability in time in SPLs and SECOs **applying evolution operations** [PGT+13, PCA+13, SHA12], **analyzing the effects of evolution** [PGT+13, LSB+10] or **planning the process of evolution** [SPP+13, SPBL12]. However, the general notion of all these approaches is to view evolution as a transactional and discrete procedure, by which a variability model and associated assets are transformed into an evolved state within a single step. Yet, it may be necessary to represent evolution on a more fine-grained level where individual variable assets have different versions, which may have an impact on configuration knowledge. In consequence, none of these approaches may be used to derive variants with variability in space and time.

6.8. Chapter Summary

In this chapter, an integration of HFMs and evolution delta modules was presented in order to allow for managing of variability in space and time in software families. Features and feature versions of an HFM (or logical expressions thereof) were associated with configuration and evolution delta modules. It was shown how to find relevant delta modules for a particular configuration. Furthermore, it was demonstrated how to derive and evaluate a dependency graph of delta modules containing “apply after” arcs to determine a suitable application sequence for the relevant delta modules. To ease determining configurations, a procedure was introduced to automatically select a suitable combination of versions for a pre-configuration of features and, possibly, feature versions.

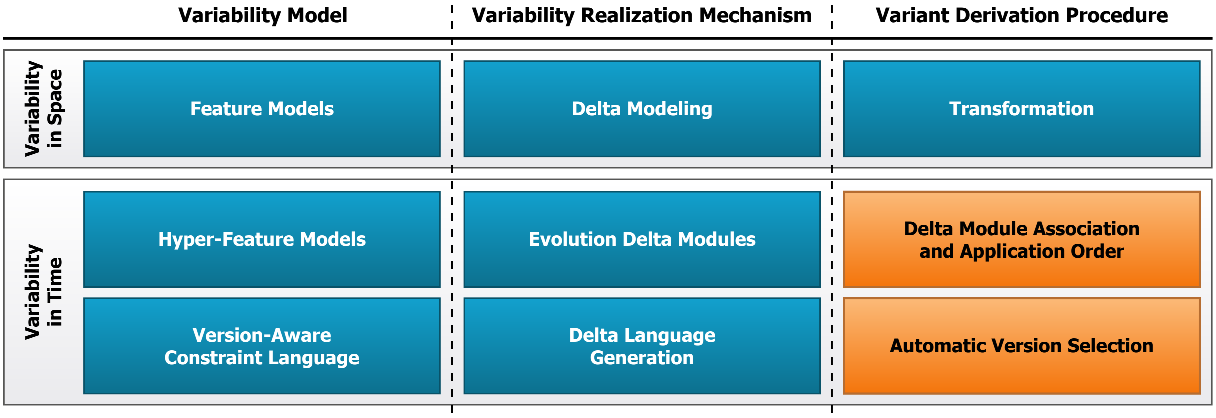

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Hyper-Feature Models
Evolution Delta Modules
Delta Module Association and Application Order
Version-Aware Constraint Language
Delta Language Generation
Automatic Version Selection

Figure 6.7.: Contributions of Chapter 6 in context of the thesis.

With these contributions, it is possible to use a configuration of features and feature versions from an HFM to determine the associated delta modules and to compute a suitable application sequence. As a result, it is possible to derive variants of an SPL or a SECO that contain aspects of variability in space and time in the form of configured functionality (features) at specific revisions (feature versions). Figure 6.7 illustrates the contributions of Chapter 6 in context of the thesis.

Part III.

Realization and Application

7. Realization as Tool Suite DeltaEcore

The concepts for integrated management of variability in space and time presented in this thesis are realized within a tool suite with the name *DeltaEcore*. For one, this tool suite provides means to create suitable delta languages for arbitrary graphical or textual source languages that are described by an EMF Ecore-based metamodel. Furthermore, it allows for the model-based definition of the artifacts mentioned throughout the thesis constituting a software family with variability in space and time, such as Hyper-Feature Models (HFMs), delta modules or the mapping model connecting the former two. Finally, DeltaEcore permits derivation of variants from the software family that encompass aspects of variability in space and time with features in various versions.

Apart from realizing the contributions presented in this thesis, DeltaEcore further improves over existing implementations of delta modeling: For one, DeltaEcore supports strong cohesion of changes related to the same feature(s) but in different realization artifacts by allowing for multiple blocks within one delta module, which each may utilize a different delta dialect (see Section 7.1.2). In addition, DeltaEcore permits the definition of compound delta operations from combinations of standard delta operations to reuse automatically generated interpretation code (see Section 7.1.3). Finally, the custom delta language created from the combination of a source language and its respective delta dialect is synthesized dynamically without the need for repeated code generation (see Section 7.2.3).

Figure 7.1 provides an overview of the application areas of the DeltaEcore tool suite and the following sections elaborate on each area in detail.

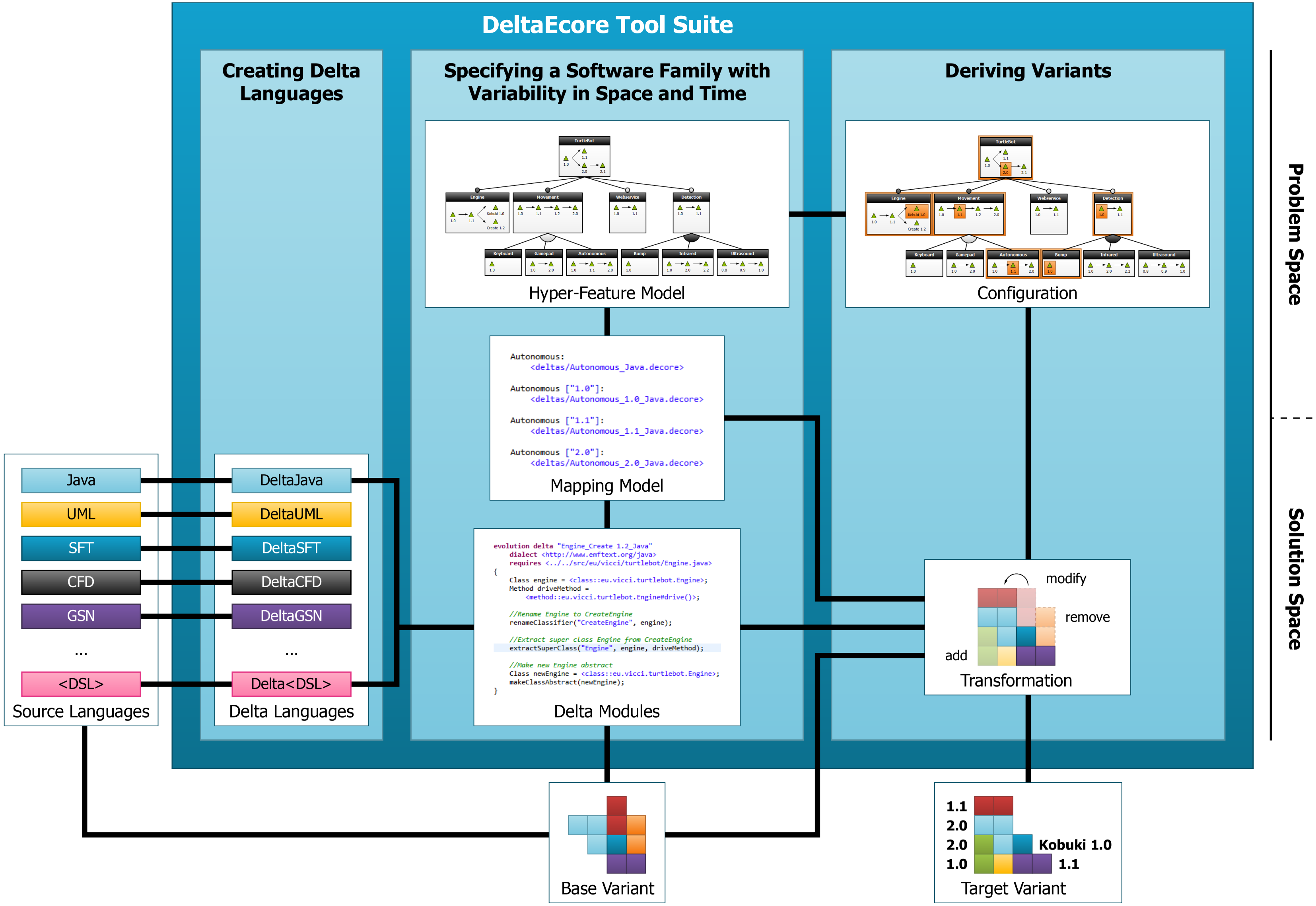
7.1. Creating Delta Languages

As a prerequisite to specifying a software family with variability in space and time, all types of realization assets affected by variability have to be supplied with a suitable delta language. For this purpose, the DeltaEcore tool suite realizes the proposed language creation facilities of Chapter 5. Within DeltaEcore, a *custom delta language* is created by combining the source-language agnostic *common base delta language* with a source-language specific *delta dialect*. In the course of metamodel-based development, these languages are each defined using a metamodel (perceived as abstract syntax) and a concrete syntax for a textual language. Due to significant similarities of selected parts of the common base delta language and delta dialects, the metamodels of the respective languages use a shared base metamodel.

The following sections describe the constituents and the process of creating a custom delta language. First, the shared base metamodel is presented. Second, the common base delta language is introduced. Third, delta dialects and their creation process are discussed.

7.1.1. Shared Base Metamodel

Both the common base delta language and delta dialects use a shared metamodel as foundation. It defines metaclasses used throughout both languages. Figure 7.2 presents the shared base metamodel.



```
Autonomous:
    <deltas/Autonomous_Java.decore>

Autonomous ["1.0"]:
    <deltas/Autonomous_1.0_Java.decore>

Autonomous ["1.1"]:
    <deltas/Autonomous_1.1_Java.decore>

Autonomous ["2.0"]:
    <deltas/Autonomous_2.0_Java.decore>
```

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

Figure 7.1.: Overview of the 3 application areas of the DeltaEcore tool suite and the most essential artifacts for specifying a software family with variability in space and time.

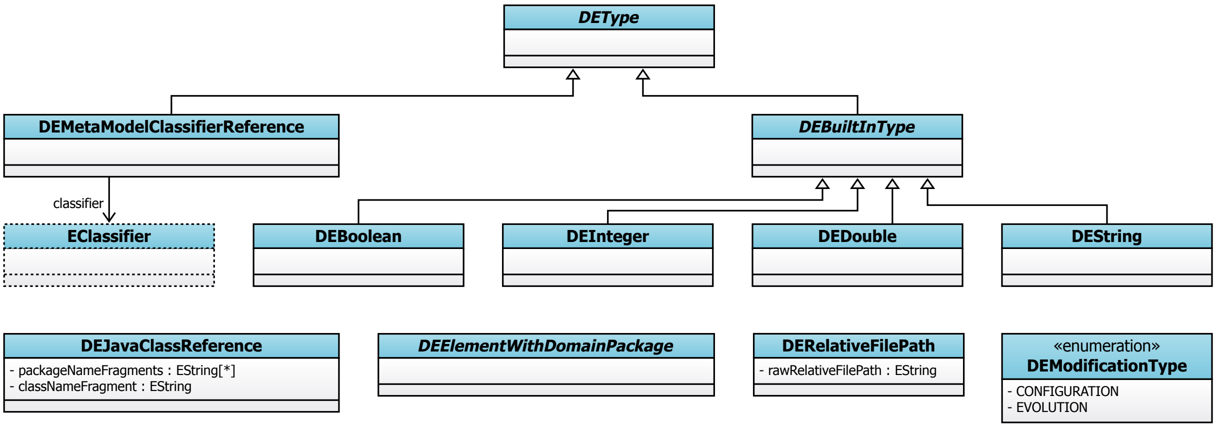


Figure 7.2.: Shared base metamodel for both the common base delta language and delta dialects formulated in EMF Ecore. The metaclass `EClassifier` is defined in the Ecore metametamodel.

The metaclass `DEType` is used to denote types of variables and parameters within delta modules (see Section 7.1.2). The subclass `DEBuiltInType` and its children denote types directly provided by DeltaEcore. Furthermore, the metaclass `DEMetaModelClassifierReference` allows using types defined in the respective metamodel of the source language of a delta dialect (see Section 7.1.3).

In addition, auxiliary metaclasses are defined that are used throughout the metamodels for the common base delta language and delta dialects. The metaclass `DEJavaClassReference` allows specification of a qualified Java class name and resolution to its respective `Class` object. The marker interface `DEElementWithDomainPackage` is used to ease the resolution and validation process of references to foreign metamodels. The metaclass `DERelativeFilePath` allows for specification of a relative file path as string and supports validation of the file's location, as well as resolution to the respective `File` object of Java. Finally, the enumeration `DEModificationType` defines the available modification types (i.e., `configuration` or `evolution`) of delta operations in delta dialects, as well as delta modules, when specifying delta modules in the created custom delta language.

As the common base delta language as well as the delta dialects utilize a textual language, the shared base metamodel defines a partial textual syntax for its metaclasses that may be imported in order to create a uniform representation. Listing 7.1 presents the concrete textual syntax for the shared base metamodel.

7.1.2. Common Base Delta Language

The common base delta language specifies the source-language agnostic parts of a delta language. Most notably, it defines metaclasses for delta modules and the call of delta operations defined within a specific delta dialect (see Section 7.1.3). Figure 7.3 depicts the metamodel of the common base delta language.

The metaclass `DEDelta` defines a delta module consisting of multiple blocks represented by the metaclass `DEDeltaBlock`. Through this mechanism, a single delta module may utilize multiple delta dialects in order to alter realization artifacts that belong to the same feature, but are specified in different source languages (see Section 7.2.3).

In turn, a delta block consists of multiple statements using the common base metaclass `DEStatement`. Most notably, delta operations defined in the referenced delta dialect may be

```
SYNTAXDEF decorebase
FOR <http://deltaecore.org/decorebase/1.0>
//Irrelevant. The language is never instantiated as it merely serves as basis for others.
START DEMetaModelClassifierReference, DEBoolean, DEInteger,
  DEDouble, DEString, DEJavaClassReference, DERelativeFilePath

TOKENS {
  DEFINE WHITESPACE $(' '|'\t'|'\f')+$;
  DEFINE LINEBREAK $('\r\n'|'\r'|'\n')$;

  DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\uffff'))*$;
  DEFINE ML_COMMENT $'/*'.*'*/'$;

  DEFINE IDENTIFIER_TOKEN $('A'..'Z'|'a'..'z'|'_')
    ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')*$;
}

RULES {
  DEMetaModelClassifierReference ::= classifier[IDENTIFIER_TOKEN];

  DEBoolean ::= "Boolean";
  DEInteger ::= "Integer";
  DEDouble ::= "Double";
  DEString ::= "String";

  DEJavaClassReference ::= (packageNameFragments[IDENTIFIER_TOKEN] ".")*
    classNameFragment[IDENTIFIER_TOKEN];

  DERelativeFilePath ::= rawRelativeFilePath['<','>'];
}
```

Listing 7.1: Concrete syntax for the shared base metamodel from Figure 7.2 as specified with EMFText.

invoked by using the metaclass `DEDeltaOperationCall` and assigning values to the operation's parameters with instances of the metaclass `DEArgument`. Furthermore, the metaclass `DEVariableDeclaration` allows for the specification of variables and their values to serve as aliases for the respective values, e.g., as arguments to a delta operation call. Finally, it is possible to directly invoke expressions by using the metaclass `DEStandaloneExpressionStatement`.

Expressions of the common base delta language share the base class `DEExpression2`[1]. Unlike statements, expressions produce a value that may be used further. For one, it is possible to instantiate metaclasses of the source language's metamodel using the metaclass `DEVirtualConstructorCall`, which permits named arguments in arbitrary order for initialization. Furthermore, new objects of data types of the source language can be created using the metaclass `DEDataTypeInitializer` by specifying a string value that can be transformed to an object of the referenced data type. In addition, concrete existing values may be used as expression by employing the metaclass `DEValue`. Its subclass `DELiteral` and the respective child metaclasses allow for the specification of literal values for the built-in types of Figure 7.2, as well as for enumerations. Furthermore, the metaclass `DEModelElementIdentifier` allows for the specification of a string identifier that is resolved to an element of the source model to be altered using a (custom) resolution strategy (see Section 7.1.3). Finally, the metaclass `DEVariableReference` permits referencing previously declared variables in order to use their current value.

[1] The name was chosen to disambiguate the metaclass from `DEExpression` of the expression language used for version-aware constraints and the mapping model (see Figure 7.9).

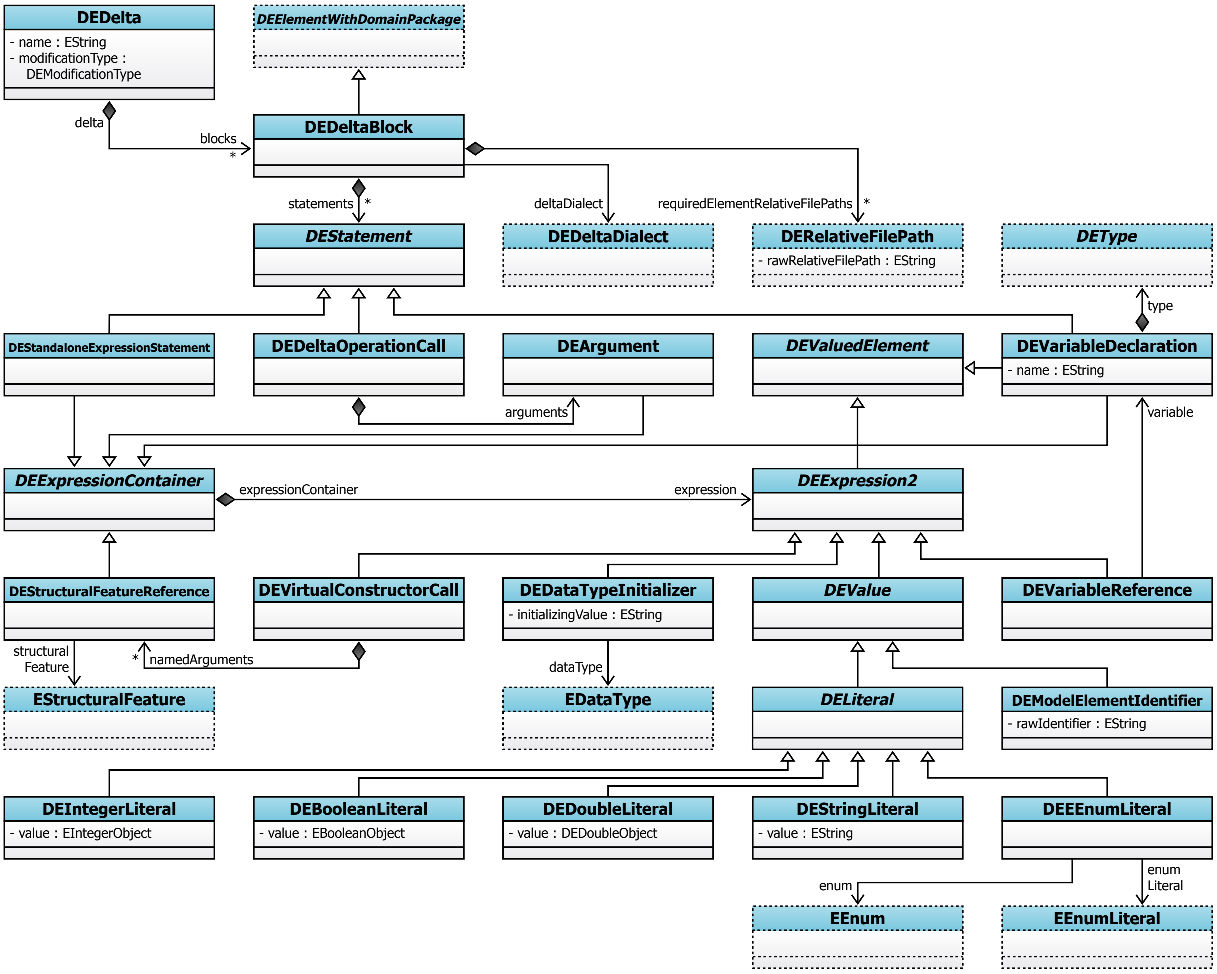


Figure 7.3.: Metamodel for the common base delta language formulated in EMF Ecore. The metaclasses `DEElementWithDomainPackage`, `DERelativeFilePath` and `DEType` are defined in Figure 7.2. The metaclass `DEDeltaDialect` is defined in Figure 7.4. The metaclasses `EStructuralFeature`, `EDataType`, `EEnum` and `EEnumLiteral` are defined in the Ecore metametamodel.

To specify language fragments of the common base delta language, a textual language is provided by DeltaEcore. For this purpose, the metamodel in Figure 7.3 is perceived as an abstract syntax. Furthermore, the concrete syntax depicted in Listing 7.2 is added to the metamodel. Using this concrete syntax with the tool EMFText allows for the creation of a parser and an editor for a textual language for delta modules (see Section 7.2.3).

Both the metamodel and the concrete syntax for the common base delta language specify the mere structure of the respective language fragments. Well-formedness of these fragments and especially their static semantics are enforced using constraints defined on the level of the metamodel depicted in Figure 7.3. Through this mechanism, demands on the validity of language fragments in the common base delta language are enforced, such as the correct number and type of arguments in a delta operation call with regard to the definition of the delta operation in the delta dialect.

```
SYNTAXDEF decore
FOR <http://deltaecore.org/decore/1.0>
START DEDelta

IMPORTS {
  decorebase : <http://deltaecore.org/decorebase/1.0>
    WITH SYNTAX decorebase <../../org.deltaecore.core.decorebase/model/DEcoreBase.cs>
}

TOKENS {
  DEFINE BOOLEAN_LITERAL_TOKEN $('true'|'false')$;
  DEFINE INTEGER_LITERAL_TOKEN $('0')|(('-')?('1'..'9')('0'..'9')*)$;
  DEFINE DOUBLE_LITERAL_TOKEN $('-')?('0'..'9')+'.'('0'..'9')*$;
}

RULES {
  DEDelta ::= abstract["abstract" : ""]
    (modificationType[CONFIGURATION : "configuration", EVOLUTION : "evolution"])?
    "delta" name['"','"', '\\'] (blocks (blocks)*)?;

  DEDeltaBlock ::= "dialect" deltaDialect['<','>']
    ("requires" requiredElementRelativeFilePaths ("," requiredElementRelativeFilePaths)*)?
    "{" statements* "}";

  DEDeltaOperationCall ::= operationDefinition[IDENTIFIER_TOKEN]
    "(" (arguments ("," arguments)*)? ")" ";";
  DEArgument ::= expression;

  DEEEnumLiteral ::= enum[IDENTIFIER_TOKEN] "." enumLiteral[IDENTIFIER_TOKEN];

  DEBooleanLiteral ::= value[BOOLEAN_LITERAL_TOKEN];
  DEIntegerLiteral ::= value[INTEGER_LITERAL_TOKEN];
  DEDoubleLiteral ::= value[DOUBLE_LITERAL_TOKEN];
  DEStringLiteral ::= value['"', '"', '\\'];

  DEModelElementIdentifier ::= rawIdentifier['<','>'];

  DEVariableDeclaration ::= type name[IDENTIFIER_TOKEN] ("=" expression ) ";";

  DEVirtualConstructorCall ::= "new" type "(" (namedArguments ("," namedArguments)*)? ")";
  DEStructuralFeatureReference ::= structuralFeature[IDENTIFIER_TOKEN] ":" expression;

  DEDataTypeInitializer ::= "new" dataType[IDENTIFIER_TOKEN]
    "(" initializingValue['"','"', '\\'] ")";

  DEStandaloneExpressionStatement ::= expression ";";
  DEVariableReference ::= variable[IDENTIFIER_TOKEN];
}
```

Listing 7.2: Concrete syntax for the metamodel of the common base delta language from Figure 7.3 as specified with EMFText.

7.1.3. Delta Dialects

To create a custom delta language for a source language, the common base delta language has to be tied to the metamodel of the source language. This is achieved by providing a delta dialect that specifies suitable delta operations for altering the source language. Delta dialects are defined by the metamodel depicted in Figure 7.4.

The metaclass `DEDeltaDialect` allows definition of a delta dialect that references the root package of the source language's metamodel (`domainPackage`) and defines a set of suitable delta operations (`deltaOperationDefinitions`). All definitions of delta operations share the base metaclass `DEDeltaOperationDefinition`. The direct subclasses are `DEStandardDeltaOperationDefinition` and `DECustomDeltaOperationDefinition` respectively allowing for the definition of standard and custom delta operations (see Section 5.4). Concrete ancestors of `DEStandardDeltaOperationDefinition` are defined for set/unset, add/insert/remove, modify and detach delta operations in accordance with the structure of the language creation facilities described in Section 5.4. Standard delta operations use a defined number, sequence and types of parameters, whereas custom delta operations may specify an arbitrary number of parameters in arbitrary sequence and of arbitrary types. To define parameters, the metaclass `DEParameter` provides various subclasses: The metaclass `DENamedParameter` may be used to specify a parameter whose arguments take a concrete value of an explicitly specified type. Furthermore, the metaclass `DEAbstractModelElementParameter` and its concrete subclasses permit referencing a classifier of the source language's metamodel with a particular specified reference or attribute. Using this metaclass, it is possible to target a particular reference or attribute for modification, e.g., when defining an add/insert/remove or set/unset/modify standard delta operation. The type for the respective arguments is derived from the given reference or attribute.

Within the DeltaEcore tool suite, delta dialects are specified in a textual language. For this purpose, the metamodel depicted in Figure 7.4 is perceived as abstract syntax and augmented with the concrete syntax of Listing 7.3 for EMFText.

To demonstrate the specification of delta dialects, EMF Ecore as a notation for metamodels is used as an example of a source language: As the metamodel of Ecore, i.e., the metametamodel of most source languages, is itself defined in Ecore, it is possible to perceive the Ecore metamodel itself as a source language. This is useful as EMF Ecore metamodels may be realization assets subjected to variability themselves (see Section 8.2 and Section 8.3). Using this fact, the delta language for EMF Ecore can be created using the tool suite DeltaEcore. Due to the name of its source language, the name of the delta language is *DeltaEcore* which is similar to the name of the presented tool suite, so that the name "DeltaEcore" may be ambiguous. For purposes of disambiguation, the usage of "DeltaEcore" to reference the delta language for Ecore metamodels is qualified explicitly. The complete delta dialect for EMF Ecore is presented in Listing 7.4.

The textual representation of a delta dialect is split up into 2 blocks for `configuration` and `deltaOperations`. The `configuration` block (Lines 3–5) defines the basic setup information of the delta dialect. In particular, the source language's metamodel has to be specified by giving the respective URI. The URI is internally resolved to retrieve the root package of the metamodel.

Furthermore, it is possible to optionally provide a custom `identifierResolver`—a Java class used to resolve references to elements within the model (Line 5). The default implementation uses attributes flagged as ID in Ecore to resolve references. However, it may be necessary to use custom identifiers, such as with hierarchically structured models without unique identifiers. The characteristics of the identifiers depend on the source language, so that the implementation of the respective identifier resolver is delegated to the creator of the delta language if the standard behavior does not suffice. For the example of Ecore as a source language, this means that various different

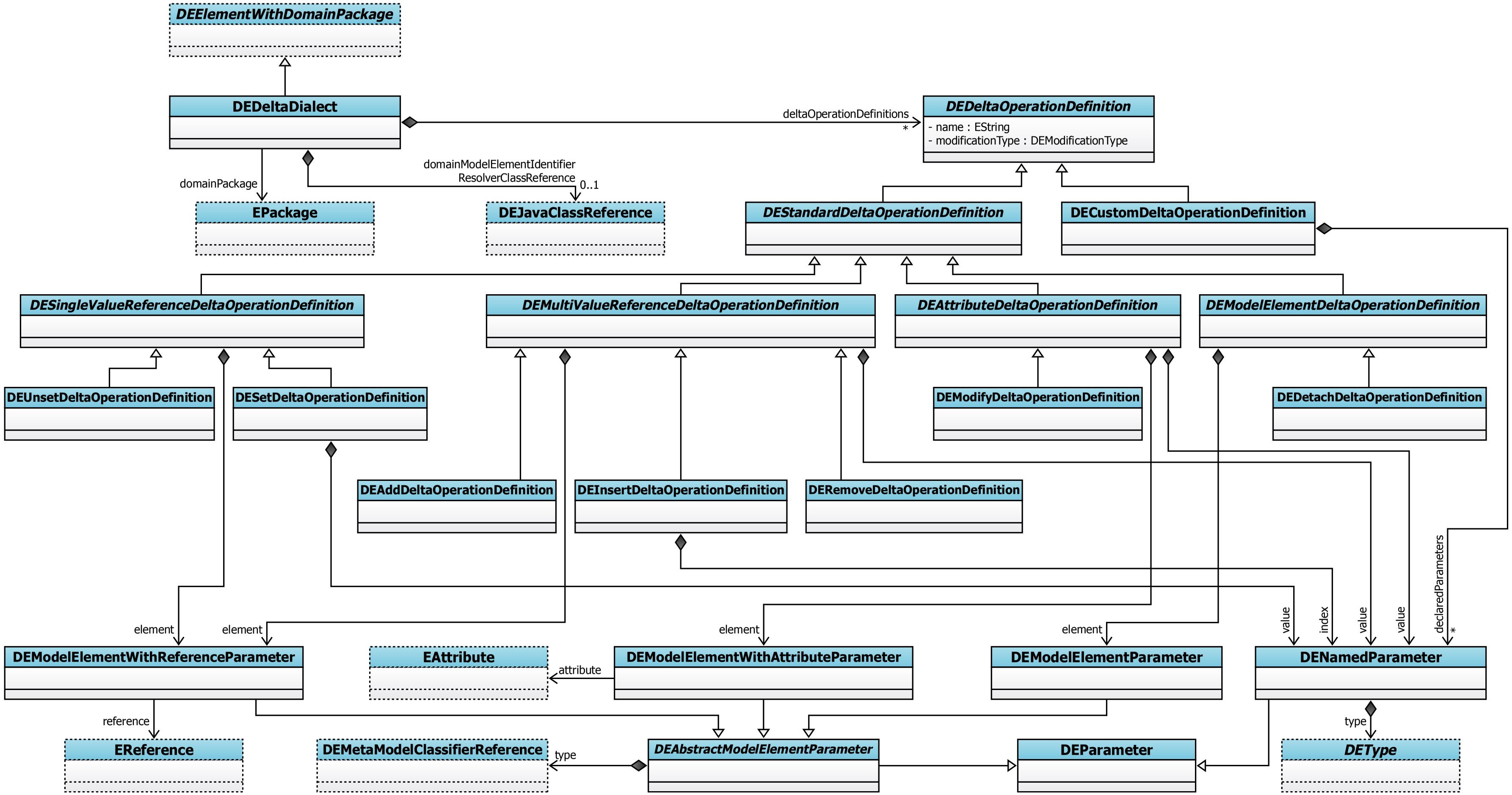

DEElementWithDomainPackage
DEDeltaDialect
deltaOperationDefinitions
*
DEDeltaOperationDefinition
- name : EString
- modificationType : DEModificationType
domainPackage
domainModelElementIdentifier
ResolverClassReference
0..1
EPackage
DEJavaClassReference
DEStandardDeltaOperationDefinition
DECustomDeltaOperationDefinition
DESingleValueReferenceDeltaOperationDefinition
DEMultiValueReferenceDeltaOperationDefinition
DEAttributeDeltaOperationDefinition
DEModelElementDeltaOperationDefinition
DEUnsetDeltaOperationDefinition
DESetDeltaOperationDefinition
DEModifyDeltaOperationDefinition
DEDetachDeltaOperationDefinition
DEAddDeltaOperationDefinition
DEInsertDeltaOperationDefinition
DERemoveDeltaOperationDefinition
element
element
element
element
value
index
value
value
declaredParameters
*
DEModelElementWithReferenceParameter
EAttribute
attribute
DEModelElementWithAttributeParameter
DEModelElementParameter
DENamedParameter
reference
type
EReference
DEMetaModelClassifierReference
type
DEAbstractModelElementParameter
DEParameter
DEType

Figure 7.4.: Metamodel for delta dialects formulated in EMF Ecore. The metaclasses DEJavaClassReference and DEType are defined in Figure 7.2. The metaclasses EPackage, EReference and EAttribute are defined in the Ecore metametamodel.

```
SYNTAXDEF decoredialect
FOR <http://deltaecore.org/decoredialect/1.0>
START DEDeltaDialect

IMPORTS {
  decorebase : <http://deltaecore.org/decorebase/1.0>
    WITH SYNTAX decorebase <../../org.deltaecore.core.decorebase/model/DEcoreBase.cs>
}

RULES {
  DEDeltaDialect ::= "deltaDialect"
    "{"
      "configuration" ":"
      "metaModel" ":" domainPackage[’<’,’>’] ";"
      ("identifierResolver" ":" domainModelElementIdentifierResolverClassReference ";")?
      "deltaOperations" ":" deltaOperationDefinitions*
    "}";


  DESetDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "setOperation" name[IDENTIFIER_TOKEN] "(" value "," element ")" ";";
  DEUnsetDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "unsetOperation" name[IDENTIFIER_TOKEN] "(" element ")" ";";

  DEAddDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "addOperation" name[IDENTIFIER_TOKEN] "(" value "," element ")" ";";
  DEInsertDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "insertOperation" name[IDENTIFIER_TOKEN] "(" value "," element "," index ")" ";";
  DERemoveDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "removeOperation" name[IDENTIFIER_TOKEN] "(" value "," element ")" ";";

  DEModifyDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "modifyOperation" name[IDENTIFIER_TOKEN] "(" value "," element ")" ";";

  DEDetachDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "detachOperation" name[IDENTIFIER_TOKEN] "(" element ")" ";";

  DECustomDeltaOperationDefinition ::=
    (modificationType[CONFIGURATION : "", EVOLUTION : "evolution"])?
    "customOperation" name[IDENTIFIER_TOKEN]
    "(" (declaredParameters ("," declaredParameters)*)? ")" ";";


  DENamedParameter ::= type name[IDENTIFIER_TOKEN];
  DEModelElementParameter ::= type name[IDENTIFIER_TOKEN];
  DEModelElementWithReferenceParameter ::= type "[" reference[IDENTIFIER_TOKEN] "]"
    name[IDENTIFIER_TOKEN];
  DEModelElementWithAttributeParameter ::= type "[" attribute[IDENTIFIER_TOKEN] "]"
    name[IDENTIFIER_TOKEN];
}
```

Listing 7.3: Concrete syntax for the metamodel of delta dialects from Figure 7.4 as specified with EMFText.

```
deltaDialect
{
  configuration:
    metaModel: <http://www.eclipse.org/emf/2002/Ecore>;
    identifierResolver: org.eclipse.emf.ecore.delta.resolver.EcoreIDResolver;

  deltaOperations:
    //EAttribute
    addOperation addEAttribute(EAttribute eAttribute, EClass [eStructuralFeatures] eClass);
    configuration detachOperation removeEAttribute(EAttribute eAttribute);

    //EClassifier
    evolution customOperation setEClassifierPackage(EPackage ePackage,
      EClassifier eClassifier);

    //EClass
    configuration addOperation addEClass(EClass eClass, EPackage [eClassifiers] ePackage);
    evolution modifyOperation setEClassName(String newName, EClass [name] eClass);
    modifyOperation setEClassAbstract(Boolean isAbstract, EClass [abstract] eClass);
    modifyOperation setEClassInterface(Boolean isInterface, EClass [interface] eClass);
    addOperation addESuperType(EClass superTypeEClass, EClass [eSuperTypes] eClass);
    removeOperation removeESuperType(EClass eClass, EClass [eSuperTypes] eSuperClass);
    detachOperation removeEClass(EClass eClass);

    //EDataType
    addOperation addEDataType(EDataType eDataType, EPackage [eClassifiers] ePackage);
    detachOperation removeEDataType(EDataType eDataType);

    //EEnum
    addOperation addEEnum(EEnum eEnum, EPackage [eClassifiers] ePackage);
    detachOperation removeEEnum(EEnum eEnum);

    //EEnumLiteral
    addOperation addEEnumLiteral(EEnumLiteral eEnumLiteral, EEnum [eLiterals] eEnum);
    evolution modifyOperation setEEnumLiteralName(String name,
      EEnumLiteral [name] eEnumLiteral);
    evolution modifyOperation setEEnumLiteralLiteral(String literal,
      EEnumLiteral [literal] eEnumLiteral);
    detachOperation removeEEnumLiteral(EEnumLiteral eEnumLiteral);

    //EOperation
    addOperation addEOperation(EOperation eOperation, EClass [eOperations] eClass);
    customOperation setEOperationImplementation(EOperation eOperation,
      String implementation);
    detachOperation removeEOperation(EOperation eOperation);

    //EPackage
    addOperation addEPackage(EPackage eSubPackage, EPackage [eSubpackages] ePackage);
    evolution detachOperation removeEPackage(EPackage ePackage);

    //EReference
    addOperation addEReference(EReference eReference, EClass [eStructuralFeatures] eClass);
    customOperation makeEReferencesOpposite(EReference reference1, EReference reference2);
    evolution modifyOperation setEReferenceName(String name, EReference [name] eReference);
    modifyOperation setEReferenceLowerBound(Integer lowerBound,
      EReference [lowerBound] eReference);
    modifyOperation setEReferenceUpperBound(Integer upperBound,
      EReference [upperBound] eReference);
    detachOperation removeEReference(EReference eReference);
}
```

Listing 7.4: Delta dialect for Ecore metamodels.

referencable entities have to be distinguished (e.g., `EClass`, `EAttribute`, `EOperation` etc.) and that names may have to be qualified, in order to disambiguate similarly named entities in different packages. Hence, identifiers such as `root.sub.EClass1`, as well as `root.EClass2.operation1()`, are valid and need to be resolved to the respective objects by the identifier resolver.

The `deltaOperations` block (Lines 7-59) allows for the definition of delta operations suitable for altering artifacts of the source language. All 7 types of standard delta operations of Section 5.4 may be used with the respective defined number, sequence and types of parameters (e.g., Lines 9, 10, 17). Furthermore, custom delta operations may be defined with arbitrary number, sequence and types of parameters (e.g., Lines 13–14,43–44). For all delta operation definitions, arbitrary names may be chosen provided that they are unique within the delta dialect (i.e., overloading is not supported).

For each of the operation definitions, the creator of the delta dialect has to decide whether to define a configuration (e.g., Lines 10, 17) or an evolution (e.g., Lines 13, 18) delta operation where the latter may exclusively be used in evolution delta modules (see Section 5.3). If no explicit modification type is provided, the delta operation is assumed to be a configuration delta operation (e.g., Lines 9, 19, 20).

To make a delta dialect usable for variant derivation, the semantics of the specified delta operations have to be defined. Within the DeltaEcore tool suite, this is achieved by providing an interpreter for the respective delta dialect. Due to the model-based nature of the delta dialects and the specified source language, as well as the defined semantics of standard delta operations, it is possible to generate large parts of the delta dialect interpreter in the form of Java source code.

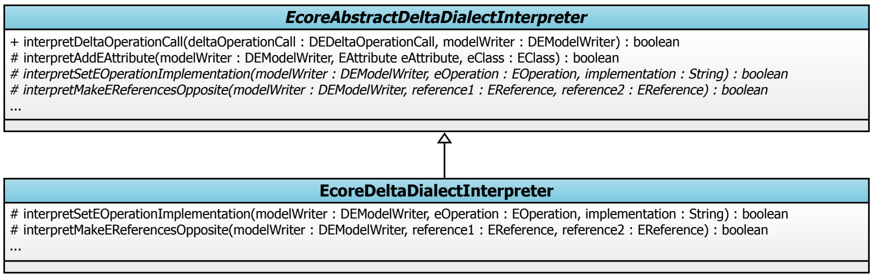

EcoreAbstractDeltaDialectInterpreter
+ interpretDeltaOperationCall(deltaOperationCall : DEDeltaOperationCall, modelWriter : DEModelWriter) : boolean
# interpretAddEAttribute(modelWriter : DEModelWriter, EAttribute eAttribute, eClass : EClass) : boolean
# interpretSetEOperationImplementation(modelWriter : DEModelWriter, eOperation : EOperation, implementation : String) : boolean
# interpretMakeEReferencesOpposite(modelWriter : DEModelWriter, reference1 : EReference, reference2 : EReference) : boolean
...
EcoreDeltaDialectInterpreter
# interpretSetEOperationImplementation(modelWriter : DEModelWriter, eOperation : EOperation, implementation : String) : boolean
# interpretMakeEReferencesOpposite(modelWriter : DEModelWriter, reference1 : EReference, reference2 : EReference) : boolean
...

Figure 7.5.: Class diagram for the delta dialect interpreter generated for the delta dialect for EMF Ecore using the Generation Gap [Vli98] pattern to separate generated and manually implemented source code.

For reasons of a clean separation between generated and manually implemented source code, the Generation Gap [Vli98] pattern is utilized for the creation of delta dialect interpreters: An abstract base class contains all generated content and a subclass defines all manual implementations. The names of these 2 classes are synthesized from the name of the source language. In the example of Ecore as a source language, the classes of the delta dialect interpreter are named `EcoreAbstractDeltaDialectInterpreter` and `EcoreDeltaDialectInterpreter` as depicted in Figure 7.5. The abstract base class contains generated methods to interpret each of the delta operations defined in the delta dialect. Furthermore, the entry point for the execution of the delta dialect interpreter is a method that receives an object of the (generic) type `DEDeltaOperationCall` and performs a dispatch to the suitable method of the interpreter by analyzing the called delta operation's name. An excerpt of the abstract part of the delta dialect interpreter for EMF Ecore as source language is presented as example in Listing 7.5[2].

[2] The class `DEModelWriter` is a placeholder for future extension of the tool suite DeltaEcore to allow for fine-grained control over and recording of all model accesses in the course of applying delta operations.

```java
package org.eclipse.emf.ecore.delta;

//...

public abstract class EcoreAbstractDeltaDialectInterpreter
  implements DEDeltaDialectInterpreter {

  @Override
  public boolean interpretDeltaOperationCall(DEDeltaOperationCall deltaOperationCall,
    DEModelWriter modelWriter) {

    DEDeltaOperationDefinition deltaOperationDefinition =
      deltaOperationCall.getOperationDefinition();
    String deltaOperationName = deltaOperationDefinition.getName();

    List<DEArgument> arguments = deltaOperationCall.getArguments();

    if (deltaOperationName.equals("addEAttribute")) {
      EAttribute eAttribute = (EAttribute) arguments.get(0).getExpression().getValue();
      EClass eClass = (EClass) arguments.get(1).getExpression().getValue();

      return interpretAddEAttribute(modelWriter, eAttribute, eClass);
    }

    if (deltaOperationName.equals("setEOperationImplementation")) {
      EOperation eOperation = (EOperation) arguments.get(0).getExpression().getValue();
      String implementation = (String) arguments.get(1).getExpression().getValue();

      return interpretSetEOperationImplementation(modelWriter, eOperation, implementation);
    }

    if (deltaOperationName.equals("makeEReferencesOpposite")) {
      EReference reference1 = (EReference) arguments.get(0).getExpression().getValue();
      EReference reference2 = (EReference) arguments.get(1).getExpression().getValue();

      return interpretMakeEReferencesOpposite(modelWriter, reference1, reference2);
    }

    //...
  }

  protected boolean interpretAddEAttribute(DEModelWriter modelWriter, EAttribute eAttribute,
    EClass eClass) {

    //Add eAttribute to eClass.eStructuralFeatures
    modelWriter.addValue(eClass, 21, eAttribute);
    return true;
  }

  abstract protected boolean interpretSetEOperationImplementation(DEModelWriter modelWriter,
    EOperation eOperation, String implementation);
  abstract protected boolean interpretMakeEReferencesOpposite(DEModelWriter modelWriter,
    EReference reference1, EReference reference2);

  //...
}
```

Listing 7.5: Excerpt from the abstract part of the delta dialect interpreter for Ecore metamodels.

```java
package org.eclipse.emf.ecore.delta;

//...

//This class is generated only once and will NOT be overwritten.
//Changed abstract methods of the base class have to be implemented manually.
public class EcoreDeltaDialectInterpreter extends EcoreAbstractDeltaDialectInterpreter {

  @Override
  protected boolean interpretSetEOperationImplementation(DEModelWriter modelWriter,
    EOperation eOperation, String implementation) {

    //EAnnotation
    EAnnotation annotation = EcoreFactory.eINSTANCE.createEAnnotation();
    annotation.setSource("http://www.eclipse.org/emf/2002/GenModel");

    //DetailsEntry
    Entry<String, String> detailsEntry = new EAnnotationDetailsEntry("body", implementation);

    EMap<String, String> details = annotation.getDetails();
    details.add(detailsEntry);

    replaceEAnnotation(annotation, eOperation);

    return true;
  }

  @Override
  protected boolean interpretMakeEReferencesOpposite(DEModelWriter modelWriter,
    EReference reference1, EReference reference2) {

    reference1.setEOpposite(reference2);
    reference2.setEOpposite(reference1);

    return true;
  }

  //...
}
```

Listing 7.6: Excerpt from the concrete part of the delta dialect interpreter for Ecore metamodels.

For standard delta operations, the implementation of the methods interpreting the respective operations can be generated fully automatically due to the operations' defined semantics. For custom delta operations, the implementation has to be provided manually and is thus delegated to the concrete subclass of the abstract interpreter. Hence, the abstract interpreter contains an abstract declaration of the methods used to interpret custom delta operations. Furthermore, on first invocation of the generation procedure, a stub for the concrete subclass of the interpreter is generated that contains an empty concrete declaration of the abstract methods of the abstract interpreter. The interpretation of the respective custom delta operation is performed by supplying Java code within the body of the respective method. It is possible to perform arbitrary modifications on the supplied source elements but it is further possible to reuse other delta operations, such as the generated standard delta operations, to build compound delta operations. An excerpt of the concrete part of the delta dialect interpreter for EMF Ecore as source language is presented as example in Listing 7.6.

For the example delta dialect for Ecore as source language as presented in Listing 7.4, the implementation of 26 of the 29 defined delta operations can be generated fully automatically. For the remaining 3 custom delta operations, the semantics have to be specified manually by implementing the respective stub methods of the delta dialect interpreter. Furthermore, the 26 standard delta operations could be determined fully automatically by using the delta language generation procedure described in Section 5.4 (excess methods were removed from the delta dialect).

Using the artifacts and procedures described above, it is possible to create custom delta languages within DeltaEcore for arbitrary source languages providing an EMF Ecore-based metamodel. Due to the definition of language agnostic parts of a delta language within the common base delta language, solely a delta dialect needs to be defined for a source language in order to retrieve a suitable editor for delta modules, integration with HFMs and a variant derivation procedure that seamlessly integrates all delta languages of DeltaEcore. Usage of the created delta languages is demonstrated in Section 7.2.3 as part of specifying a software family encompassing variability in space and time.

7.2. Specifying a Software Family with Variability in Space and Time

To specify a software family encompassing variability in space and time with the approach presented in this thesis, a number of artifacts are required: An HFM and its version-aware constraints capture the conceptual level of variability in space and time. Furthermore, configuration and evolution delta modules manifest the effects of variability in space and time in realization assets and an application-order constraint model represents demands on the order of the delta modules. A mapping model associates combinations of features and feature versions from an HFM with sets of delta modules, so that a conceptual configuration from the HFM may be resolved to a set of relevant delta modules. Figure 7.6 illustrates the relation of the artifacts of a software family encompassing variability in space and time. The following sections explain specification of each type of artifact within the DeltaEcore tool suite as implementation of the concepts presented in the thesis.

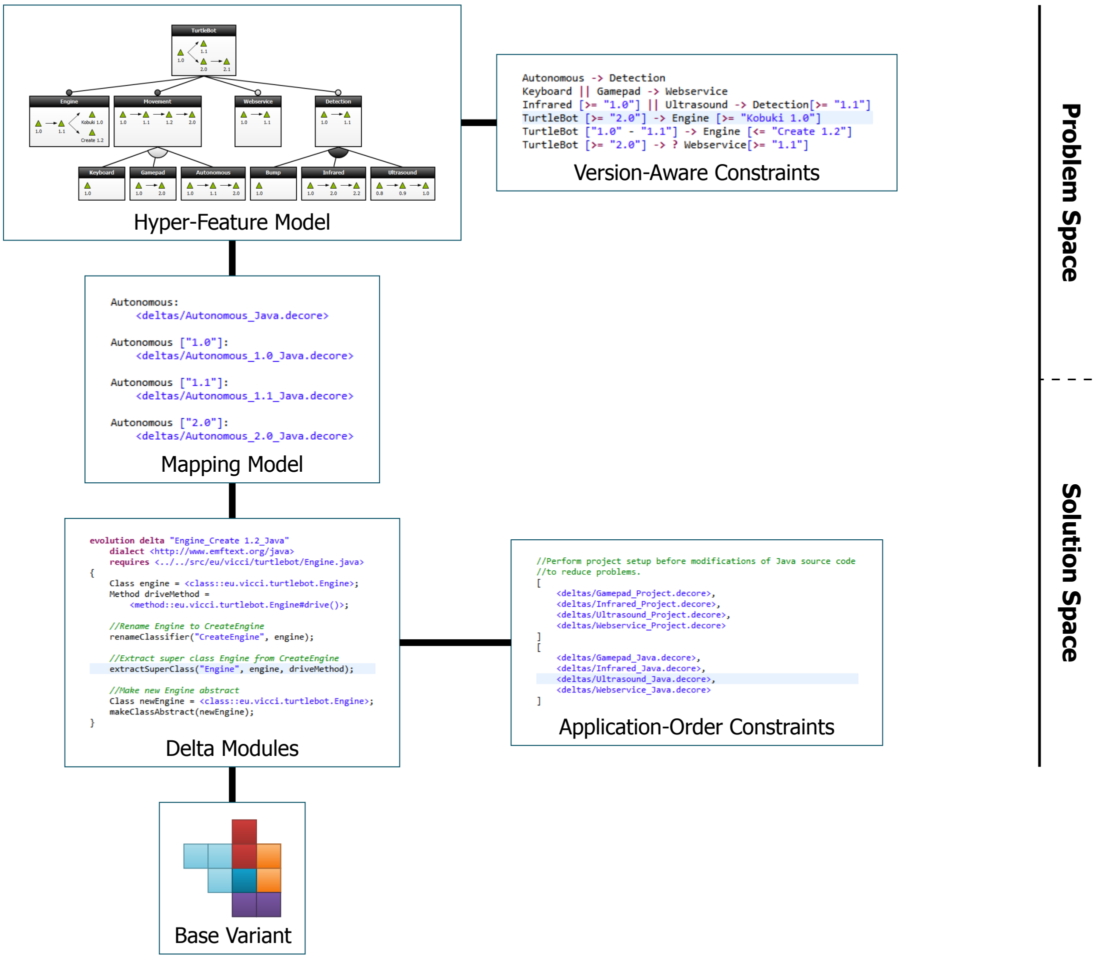
7.2.1. Hyper-Feature Models

To conceptually represent configuration knowledge regarding variability in space and time, HFMs may be used. Within the DeltaEcore tool suite, the notation for HFMs is defined by the metamodel depicted in Figure 7.7.

According to the metamodel, a `HyperFeatureModel` consists of a root feature of type `DEFeature`. Both `DEFeature` and `DEGroup` extend `DECardinalityBasedElement`, which enables them to specify minimum and maximum cardinalities. In addition, a feature is identified by a name. Each feature may specify an arbitrary number of groups (including none) as children and each group may contain between 1 and unlimited features, which spans a tree with alternating levels of features and groups. To realize versions as first-class elements, a dedicated metaclass `DEVersion` is defined in the metamodel, which may further be characterized by a string-based version number to support even non-numeric versions, such as “Kepler” or “Luna” as known from Eclipse. Development lines of versions are captured within a dedicated association relating versions of the same feature (`supersedingVersions/supersededVersion`).

To define a concrete HFM as used by a particular software family, an instance of this metamodel has to be created. The preferred way[3] of specifying the respective model within the DeltaEcore tool suite is to utilize the supplied graphical editor as depicted in Figure 7.8. The editor supports the visual representation used throughout the example figures of this thesis.

[3] As an alternative, it is also possible to utilize the generic tree-based editor generated by EMF to specify models of the metamodel depicted in Figure 7.7 to represent HFMs.



```
Autonomous -> Detection
Keyboard || Gamepad -> Webservice
Infrared [>= "1.0"] || Ultrasound -> Detection[>= "1.1"]
TurtleBot [>= "2.0"] -> Engine [>= "Kobuki 1.0"]
TurtleBot ["1.0" - "1.1"] -> Engine [<= "Create 1.2"]
TurtleBot [>= "2.0"] -> ? Webservice[>= "1.1"]
```

```
Autonomous:
    <deltas/Autonomous_Java.decore>

Autonomous ["1.0"]:
    <deltas/Autonomous_1.0_Java.decore>

Autonomous ["1.1"]:
    <deltas/Autonomous_1.1_Java.decore>

Autonomous ["2.0"]:
    <deltas/Autonomous_2.0_Java.decore>
```

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

```
//Perform project setup before modifications of Java source code
//to reduce problems.
[
    <deltas/Gamepad_Project.decore>,
    <deltas/Infrared_Project.decore>,
    <deltas/Ultrasound_Project.decore>,
    <deltas/Webservice_Project.decore>
]
[
    <deltas/Gamepad_Java.decore>,
    <deltas/Infrared_Java.decore>,
    <deltas/Ultrasound_Java.decore>,
    <deltas/Webservice_Java.decore>
]
```

Figure 7.6.: Relation of the artifacts of a software family encompassing variability in space and time within the DeltaEcore tool suite.

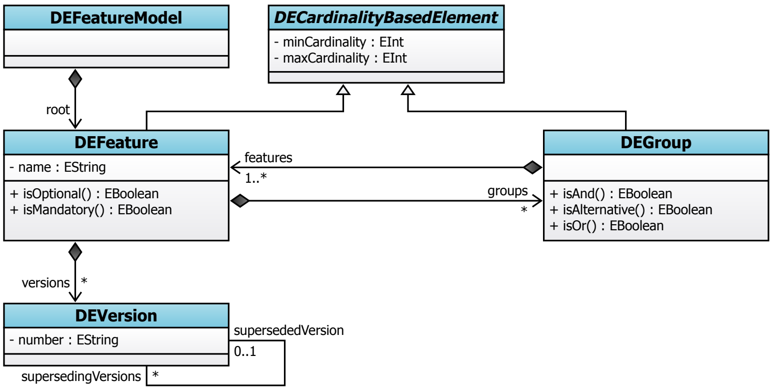


Figure 7.7.: Metamodel for HFMs formulated in EMF Ecore.

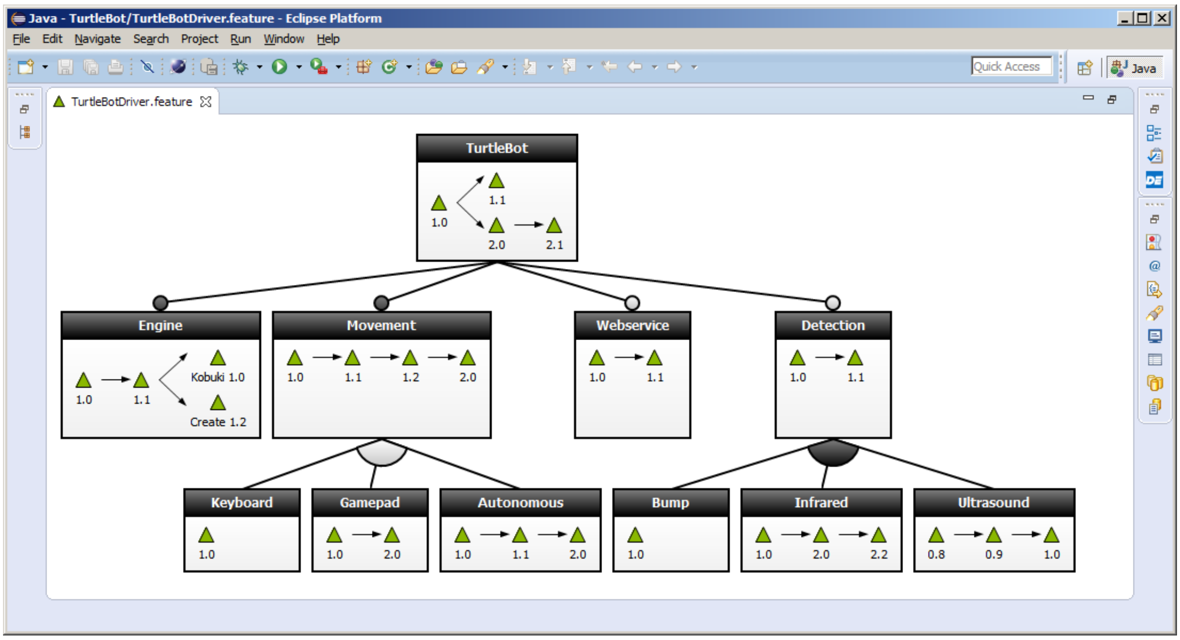


Figure 7.8.: Screenshot of the graphical editor for HFMs within the DeltaEcore tool suite.

7.2.2. Version-Aware Constraints

In addition to an HFM, it is further possible to specify version-aware constraints to restrain the set of valid configurations. For this purpose, logical expressions over features and feature versions have to be formulated. For this purpose, a dedicated metamodel for an expression language is provided as specified in Figure 7.9, which is also used by the mapping model (see below).

The metamodel may be perceived as the abstract syntax of a textual language (see Section 3.1) when a concrete syntax is supplied. For the expression language, this concrete syntax is specified for the tool EMFText as depicted in Listing 7.7. Using this concrete syntax as input, an editor for the respective textual language may be generated, which parses the provided text to an instance of the metamodel of Figure 7.9.

On basis of the metamodel for expressions, a dedicated metamodel for the version-aware constraint language is defined, as depicted in Figure 7.10.

The metamodel defines entities to specify a constraint model (`DEConstraintModel`) and its constraints (`DEConstraint`). The latter consists of nesting binary, unary and atomic expressions as defined by the respective metaclasses, where the atomic expressions allow referencing features of the HFM and constraining their versions.

Within the DeltaEcore tool suite, a textual language is provided to define version-aware constraints. For this purpose, a concrete syntax is supplied for the tool EMFText that reuses the concrete syntax of the expression language presented in Listing 7.7. The concrete syntax for the version-aware constraint language is presented in Listing 7.8 and application of the respective generated editor is demonstrated in Figure 7.11.

7.2.3. Delta Modules

To manifest the changes associated with features and feature versions of an HFM, it is necessary to specify delta modules to alter realization assets. Within the DeltaEcore tool suite, dedicated

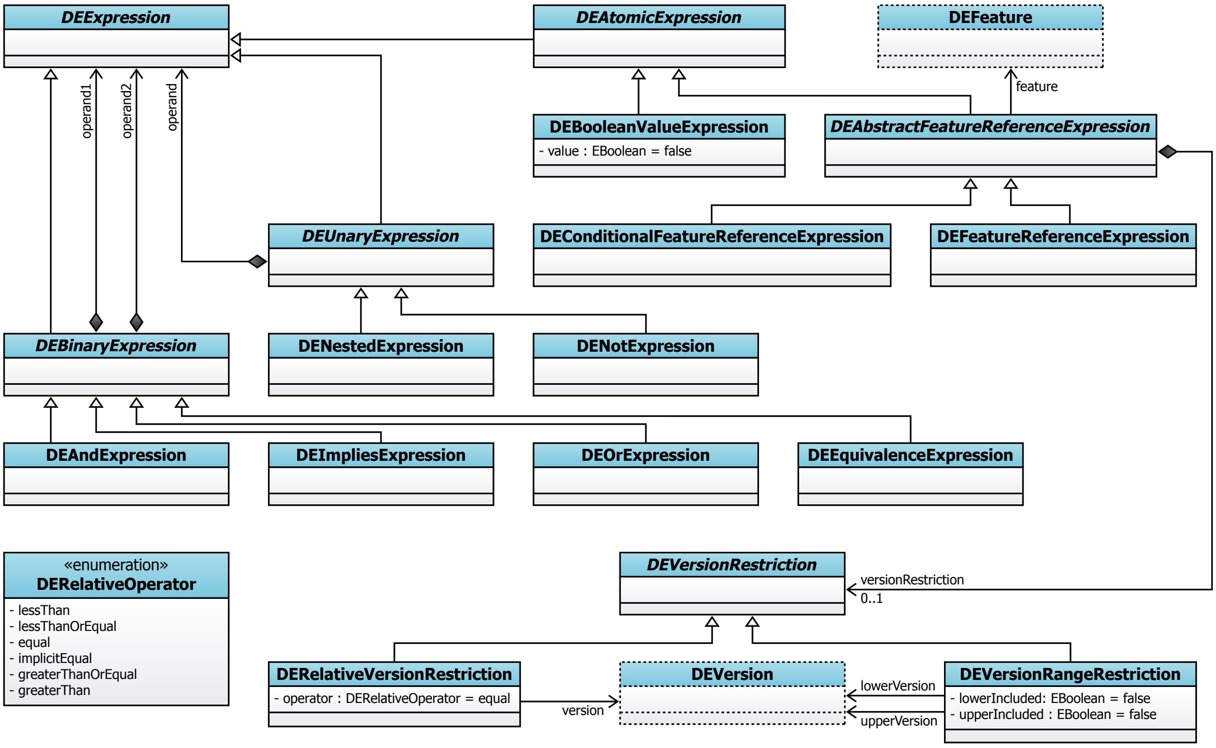


Figure 7.9.: Metamodel for the shared expression metamodel formulated in EMF Ecore. The metaclasses `DEFeature` and `DEVersion` are defined in Figure 7.7

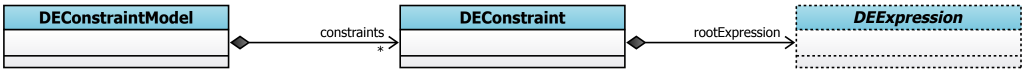
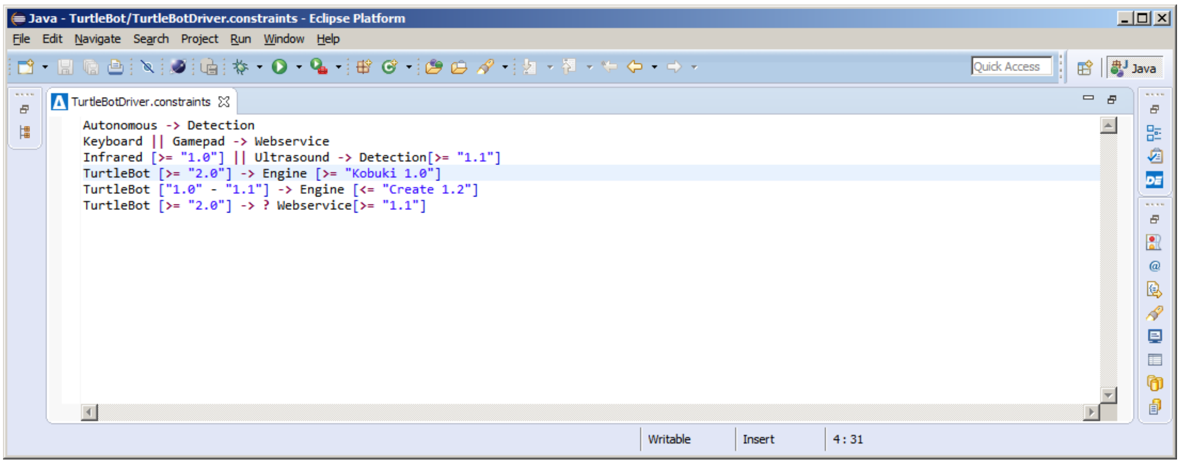


Figure 7.10.: Metamodel for the version-aware constraint language formulated in EMF Ecore. The metaclass `DEExpression` is defined in Figure 7.9.


Java - TurtleBot/TurtleBotDriver.constraints - Eclipse Platform
File
Edit
Navigate
Search
Project
Run
Window
Help
Quick Access
Java
TurtleBotDriver.constraints
Writable
Insert
4 : 31

```
Autonomous -> Detection
Keyboard || Gamepad -> Webservice
Infrared [>= "1.0"] || Ultrasound -> Detection[>= "1.1"]
TurtleBot [>= "2.0"] -> Engine [>= "Kobuki 1.0"]
TurtleBot ["1.0" - "1.1"] -> Engine [<= "Create 1.2"]
TurtleBot [>= "2.0"] -> ? Webservice[>= "1.1"]
```

Figure 7.11.: Screenshot of the textual editor for the version-aware constraint language within the DeltaEcore tool suite.

```
SYNTAXDEF expression
FOR <http://deltaecore.org/feature/expression/1.0> <Expression.genmodel>
START DEExpression

TOKENS {
  DEFINE IDENTIFIER_TOKEN $('A'..'Z'|'a'..'z'|'_')('A'..'Z'|'a'..'z'|'_'|'0'..'9')*$;

  DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\uffff'))*$;
  DEFINE ML_COMMENT $'/*'.*'*/'$;

  DEFINE LINEBREAK $('\r\n'|'\r'|'\n')$;
  DEFINE WHITESPACE $(' '|'\t'|'\f')$;
}

RULES {
  @Operator(type="binary_left_associative",  weight="1", superclass="DEExpression")
  DEEquivalenceExpression ::= operand1 "<->" operand2;

  @Operator(type="binary_left_associative", weight="2", superclass="DEExpression")
  DEImpliesExpression ::= operand1 "->" operand2;

  @Operator(type="binary_left_associative", weight="3", superclass="DEExpression")
  DEOrExpression ::= operand1 "||" operand2;

  @Operator(type="binary_left_associative", weight="4", superclass="DEExpression")
  DEAndExpression ::= operand1 "&&" operand2;

  @Operator(type="unary_prefix", weight="5", superclass="DEExpression")
  DENotExpression ::= "!" operand;

  @Operator(type="primitive", weight="6", superclass="DEExpression")
  DENestedExpression ::= "(" operand ")";


  @Operator(type="primitive", weight="6", superclass="DEExpression")
  DEFeatureReferenceExpression ::= (feature['"', '"'] | feature[]) (versionRestriction)?;

  @Operator(type="primitive", weight="6", superclass="DEExpression")
  DEConditionalFeatureReferenceExpression ::= "?" (feature['"', '"'] | feature[])
    versionRestriction;

  @Operator(type="primitive", weight="6", superclass="DEExpression")
  DEBooleanValueExpression ::= value["true" : "false"];

  DERelativeVersionRestriction ::= "[" operator[lessThan : "<", lessThanOrEqual : "<=",
    equal : "=", implicitEqual : "", greaterThanOrEqual : ">=", greaterThan : ">"]
    version['"','"'] "]";
  DEVersionRangeRestriction ::= "[" lowerIncluded["" : "^"] lowerVersion['"','"'] "-"
    upperIncluded["" : "^"] upperVersion['"','"'] "]";
}
```

Listing 7.7: Concrete syntax for the metamodel of the expression language from Figure 7.9 as specified with EMFText.

```
SYNTAXDEF constraints
FOR <http://deltaecore.org/feature/constraint/1.0>
START DEConstraintModel

IMPORTS {
  expression : <http://deltaecore.org/feature/expression/1.0>
    WITH SYNTAX expression <../../org.deltaecore.feature.expression/model/Expression.cs>
}

RULES {
  DEConstraintModel ::= constraints*;

  DEConstraint ::= rootExpression;
}
```

Listing 7.8: Concrete syntax for the metamodel of the version-aware constraint language from Figure 7.10 as specified with EMFText.

delta languages may be supplied for arbitrary source languages (see Section 7.1). These languages may be used to create delta modules for individual realization artifacts affected by variability in space and time. Changes associated with variability in space are defined in configuration delta modules and those related to variability in time are specified in evolution delta modules.

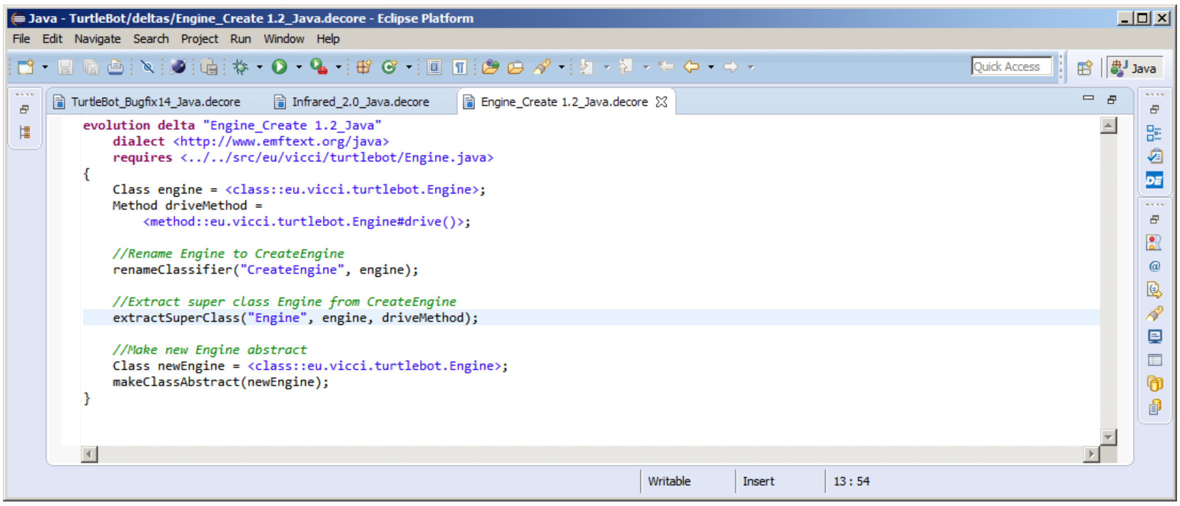
Both these types of delta modules are founded on the metamodel of the common base delta language depicted in Figure 7.3. The concrete language for the delta modules is synthesized dynamically by combining the metamodel of the common base delta language with a reference to an instance of the metamodel depicted in Figure 7.4 for the delta dialect used within the delta module. In particular, no specific metamodel has to be generated. Through this mechanism, types of elements defined within the source language's metamodel become available to the delta language and dynamic virtual constructors to create new instances of these types are made available. Furthermore, the delta operations defined within the delta dialect are provided for use within the delta modules where evolution delta operations may exclusively be used within evolution delta modules (see below). Likewise, the concrete textual syntax is a combination of that for the common base delta language as shown in Listing 7.2 as well as the defined delta operations of the delta dialect with their respective names and parameters. The DeltaEcore tool suite supplies an editor to specify delta modules in this textual language as depicted in Figure 7.12.

7.2.3.1. Configuration Delta Modules

To manifest changes associated with variability in space, configuration delta modules may be used. Listing 7.9 repeats the configuration delta module from Listing 6.1 to demonstrate specification of configuration delta modules.

In Line 1, the delta module is assigned the name "Gamepad_Java" which is used for informative purposes throughout the variant derivation process. Furthermore, using the keyword `configuration`, the delta module is explicitly designated as realizing changes associated with variability in space. In consequence, delta operations that were marked as evolution delta operations within the delta dialect (see Section 7.1.3) are not available for use. Alternatively, a delta module may implicitly be designated as configuration delta module by omitting the modification type.

In Line 2, the delta dialect to employ is specified by providing the URI of the source language's metamodel. The URI is also used as identifier for the delta dialect that permits modification of artifacts of the source language.


Java - TurtleBot/deltas/Engine_Create 1.2_Java.decore - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Quick Access
Java
TurtleBot_Bugfix14_Java.decore
Infrared_2.0_Java.decore
Engine_Create 1.2_Java.decore
Writable
Insert
13 : 54

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

Figure 7.12.: Screenshot of the textual editor for delta modules within the DeltaEcore tool suite.

In Line 3, the realization artifact that is subjected to variability is specified by providing the (relative or absolute) path of its containing file. The EMF facilities determine automatically how to retrieve a model form of the artifact by the file's extension provided that a suitable Ecore metamodel for this file type was supplied. Should more than a single artifact be required, it is possible to provide a list of files, e.g., when moving elements from one artifact to another. Furthermore, other delta modules that necessarily have to be applied before the delta module may be specified as required by giving their respective path in the file system.

In Lines 4–14, the body of the delta module is specified where a sequence of statements are supplied. For one, delta operations may be invoked directly giving their name and providing elements referenced by their respective identifiers as arguments (e.g., Line 5). Alternatively, values of elements of the altered realization artifacts may be stored in variables (e.g., Lines 7, 8) in order to use them as arguments to delta operation calls (e.g., Line 11).

```
1  configuration delta "Gamepad_Java"
2    dialect <http://www.emftext.org/java>
3    requires <../src/eu/vicci/turtlebot/Movement.java>
4  {
5    Package p = <package::eu.vicci.turtlebot>;
6    createClass("public class Gamepad {
7                    //...
8                }", p);
9
10   Class gamepad = <class::eu.vicci.turtlebot.Gamepad>;
11   Class movement = <class::eu.vicci.turtlebot.Movement>;
12
13   //Set Movement as super class of Gamepad
14   setSuperClassOfClass(movement, gamepad);
15
16   //...
17 }
```

Listing 7.9: Repetition of the example configuration delta module from Listing 6.1 that enables the feature `Gamepad` in Java source code.

7.2.3.2. Evolution Delta Modules

To manifest changes associated with variability in time, evolution delta modules may be used. Listing 7.10 repeats the evolution delta module from Listing 6.3 to demonstrate specification of configuration delta modules.

```
 1  evolution delta "Engine_Create 1.2_Java"
 2    dialect <http://www.emftext.org/java>
 3    requires <../src/eu/vicci/turtlebot/Engine.java>
 4  {
 5    Class engine = <class::eu.vicci.turtlebot.Engine>;
 6    Method driveMethod = <method::eu.vicci.turtlebot.Engine#drive()>;
 7
 8    //Rename Engine to CreateEngine
 9    renameNamedElement("CreateEngine", engine);
10
11    //Extract super class Engine from CreateEngine
12    extractSuperClass("Engine", engine, [driveMethod]);
13
14    //Make new Engine abstract
15    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
16    setAbstractModifier(true, newEngine);
17  }
```

Listing 7.10: Repetition of the example evolution delta module from Listing 6.3 that performs an update of feature `Engine` to version *Create 1.2*.

The structure of evolution delta modules is similar to that of configuration delta modules as described above. In Line 1, a name is assigned to the delta module and it is designated as evolution delta module by using the keyword `evolution` for the modification type. In Line 2, the delta dialect to use is imported by specification of the source language's URI. In Line 3, the respective realization artifact to modify is specified. Finally, in Lines 4–18, the body of the delta module allows specification of various statements such as variable declarations (e.g., Lines 5, 6, 16) as well as calls to delta operations (e.g., Lines 10, 13, 17). However, unlike with configuration delta modules, evolution delta modules support the full set of delta operations specified in the delta dialect encompassing both configuration and evolution delta operations.

7.2.3.3. Delta Modules with Multiple Blocks

The realization of individual features or their evolution may crosscut different realization assets that may possibly utilize different source languages, e.g., when a feature equally affects the realization as Java source code, the safety certification material within an SFT and the documentation in the DocBook manual. In consequence, the different source languages of the targeted artifacts demand the use of different delta dialects. When targeting only a single artifact with a single delta dialect per delta module, this leads to fragmentation of delta modules over various files that logically have such high cohesion that it may be beneficial to not separate them. For this purpose, delta modules within the DeltaEcore tool suite introduce the concept of potentially multiple *blocks* per delta module (see Section 7.1.2). Each block may alter an arbitrary number of realization artifacts with one particular delta dialect. Further blocks in one delta module may use other delta dialects to alter artifacts of different source languages where the changes have such high cohesion with that of other blocks that they should be specified within the same delta module. Furthermore, it is possible to use similar delta dialects in multiple blocks to alter realization artifacts of similar source languages in isolation. Listing 7.11 demonstrates the usage of multiple blocks within a single delta module by defining two blocks (Lines 3–10 and 12–27) for the delta modules of Listing 6.1 and Listing 6.2 that were previously specified separately.

```
1  configuration delta "Gamepad"
2
3  dialect <http://vicci.eu/eclipseproject/1.0>
4    requires <../../.classpath>
5  {
6    //Add dependencies on gamepad driver class library
7    addRequiredLibrary("lib/lwjgl2.8.5/jar/lwjgl.jar");
8    addRequiredLibrary("lib/lwjgl2.8.5/jar/lwjgl_util.jar");
9    addRequiredLibrary("lib/lwjgl2.8.5/jar/jinput.jar");
10 }
11
12 dialect <http://www.emftext.org/java>
13   requires <../../src/eu/vicci/turtlebot/Movement.java>
14 {
15   Package p = <package::eu.vicci.turtlebot>;
16   createClass("public class Gamepad {
17                   //...
18               }", p);
19
20   Class gamepad = <class::eu.vicci.turtlebot.Gamepad>;
21   Class movement = <class::eu.vicci.turtlebot.Movement>;
22
23   //Set Movement as super class of Gamepad
24   setSuperClassOfClass(movement, gamepad);
25
26   //...
27 }
```

Listing 7.11: Example of a configuration delta module using multiple blocks with different delta dialects to enable the feature `Gamepad` of the TurtleBot driver software in both the Eclipse project setup and Java source code.

7.2.4. Application-Order Constraints

Besides the “apply after” arcs stemming from the structure of the HFM and the requires relations of delta modules (see Section 6.4), explicit application-order constraints may govern demands on possible orderings of delta modules: Not all delta modules may be completely independent of one another. For example, if one delta module creates elements that are altered by another delta module, the latter one has to be applied after the prior one *if* both delta modules are used for a particular variant. Hence, this demand on possible sequences of delta modules is not a requires relation but instead an application-order constraint (see Section 3.2.2). The DeltaEcore tool suite allows specification of application-order constraints using a dedicated language founded on the metamodel depicted in Figure 7.13.

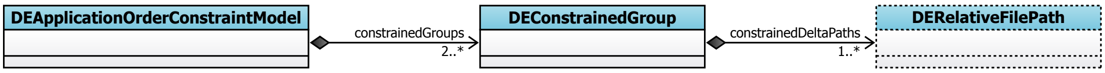


Figure 7.13.: Metamodel for application-order constraints formulated in EMF Ecore. The metaclass `DERelativeFilePath` is defined in Figure 7.2.

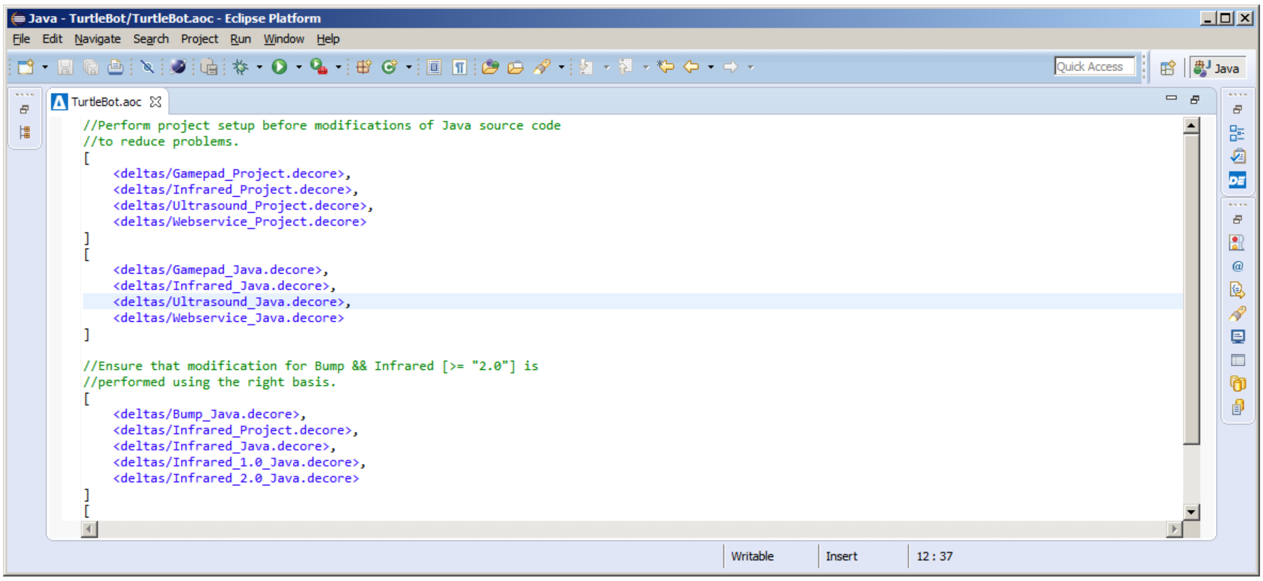
As more than a single delta module may share similar application-order constraints, delta modules may be collected in *constrained groups* by enclosing them with brackets. All delta modules listed within one constrained group demand that they are applied after the delta modules listed in the previous constrained group. Furthermore, this demand on ordering is transitive so that the delta modules of one group have to be applied after the delta modules of all predecessor groups. However, within a single group, the order of delta modules is perceived as being unspecified so that it may be defined freely by topologically sorting delta modules (see Section 7.3.3).

For the metamodel of Figure 7.13, a concrete textual syntax was defined in order to allow specification of application-order constraints with a textual language. Listing 7.12 depicts the concrete syntax for application-order constraints.

```
SYNTAXDEF aoc
FOR <http://deltaecore.org/applicationorderconstraint/1.0>
START DEApplicationOrderConstraintModel

IMPORTS {
  decorebase : <http://deltaecore.org/decorebase/1.0>
    WITH SYNTAX decorebase <../../org.deltaecore.core.decorebase/model/DEcoreBase.cs>
}

RULES {
  DEApplicationOrderConstraintModel ::= constrainedGroups constrainedGroups+;

  DEConstrainedGroup ::= "[" constrainedDeltaPaths ("," constrainedDeltaPaths)* "]";
}
```

Listing 7.12: Concrete syntax for the metamodel of application-order constraints from Figure 7.13 as specified with EMFText.

The DeltaEcore tool suite offers and editor to create models for application-order constraints using this textual language. Figure 7.14 shows an example of using this editor to specify the application-order constraints of the TurtleBot driver software.



```
//Perform project setup before modifications of Java source code
//to reduce problems.
[
    <deltas/Gamepad_Project.decore>,
    <deltas/Infrared_Project.decore>,
    <deltas/Ultrasound_Project.decore>,
    <deltas/Webservice_Project.decore>
]
[
    <deltas/Gamepad_Java.decore>,
    <deltas/Infrared_Java.decore>,
    <deltas/Ultrasound_Java.decore>,
    <deltas/Webservice_Java.decore>
]

//Ensure that modification for Bump && Infrared [>= "2.0"] is
//performed using the right basis.
[
    <deltas/Bump_Java.decore>,
    <deltas/Infrared_Project.decore>,
    <deltas/Infrared_Java.decore>,
    <deltas/Infrared_1.0_Java.decore>,
    <deltas/Infrared_2.0_Java.decore>
]
[
```

Figure 7.14.: Screenshot of the textual editor for application-order constraints within the DeltaEcore tool suite.

7.2.5. Mapping Models

A mapping model associates combinations of features and feature versions with sets of delta modules in various mapping entries. Each mapping entry resembles a logical implication consisting of a logical expression over features and feature versions as condition and a collection of references to existing delta modules as conclusion. The condition of a mapping entry is specified using the language constructs of Definition 18 similarly to the version-aware constraint language. The conclusion of a mapping entry is a collection of references to files containing delta modules. The delta modules of the conclusion have to be applied to create a variant if the respective configuration satisfies the condition posed by the logical expression over features and feature versions. A mapping model is defined as instance of the metamodel presented in Figure 7.15.

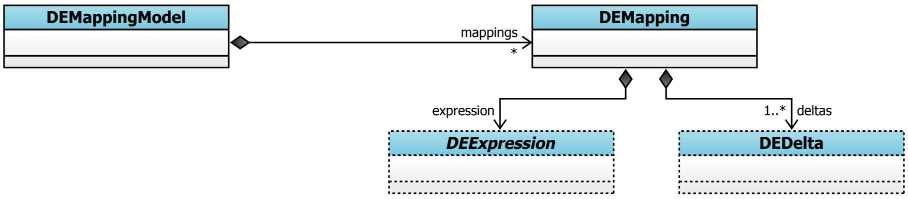


Figure 7.15.: Metamodel for the mapping of logical combinations of features and feature versions to sets of delta modules. The metaclass `DEDelta` is defined in Figure 7.3 and `DEExpression` is defined in the shared expression metamodel depicted in Figure 7.9.

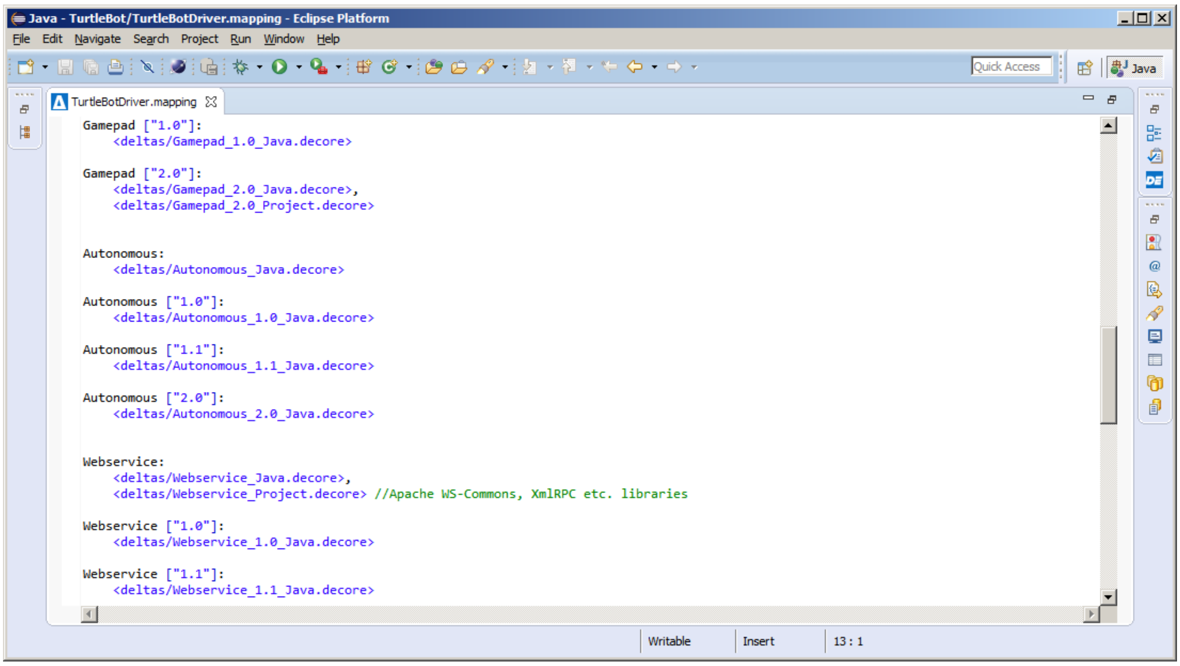
The metamodel defines metaclasses for the mapping model (`DEMappingModel`) and its mapping entries (`DEMapping`). The latter metaclass has references representing the mapping entry's condition as logical expression (`expression`) and its conclusion as set of delta modules (`deltas`). Due to the striking similarity of the condition part of a mapping entry and the constructs of the version-aware constraint language, significant parts of the realization of the version-aware constraint language may be reused to define mapping entries. For this purpose, the metaclass `DEExpression` of the shared expression metamodel depicted in Figure 7.9 is reused for the condition of a mapping entry.

Within DeltaEcore, mapping models are specified textually. For this purpose, the metamodel of Figure 7.15 is perceived as abstract syntax and the concrete textual syntax of Listing 7.13 is added in order to define a textual language.

```
SYNTAXDEF mapping
FOR <http://deltaecore.org/feature/mapping/1.0>
START DEMappingModel

IMPORTS {
  expression : <http://deltaecore.org/feature/expression/1.0>
    WITH SYNTAX expression <../../org.deltaecore.feature.expression/model/Expression.cs>
}

RULES {
  DEMappingModel ::= mappings*;

  DEMapping ::= expression ":" deltas['<','>'] ("," deltas['<','>'])*;
}
```

Listing 7.13: Concrete syntax for the metamodel of mapping models from Figure 7.15 as specified with EMFText.

Using the generated textual language, it is possible to define a mapping model that connects the previously created HFM and delta modules. Figure 7.16 shows an example of specifying a mapping model with the textual editor of DeltaEcore.


Java - TurtleBot/TurtleBotDriver.mapping - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Quick Access
Java
TurtleBotDriver.mapping
Writable
Insert
13 : 1

```
Gamepad ["1.0"]:
    <deltas/Gamepad_1.0_Java.decore>

Gamepad ["2.0"]:
    <deltas/Gamepad_2.0_Java.decore>,
    <deltas/Gamepad_2.0_Project.decore>


Autonomous:
    <deltas/Autonomous_Java.decore>

Autonomous ["1.0"]:
    <deltas/Autonomous_1.0_Java.decore>

Autonomous ["1.1"]:
    <deltas/Autonomous_1.1_Java.decore>

Autonomous ["2.0"]:
    <deltas/Autonomous_2.0_Java.decore>


Webservice:
    <deltas/Webservice_Java.decore>,
    <deltas/Webservice_Project.decore> //Apache WS-Commons, XmlRPC etc. libraries

Webservice ["1.0"]:
    <deltas/Webservice_1.0_Java.decore>

Webservice ["1.1"]:
    <deltas/Webservice_1.1_Java.decore>
```

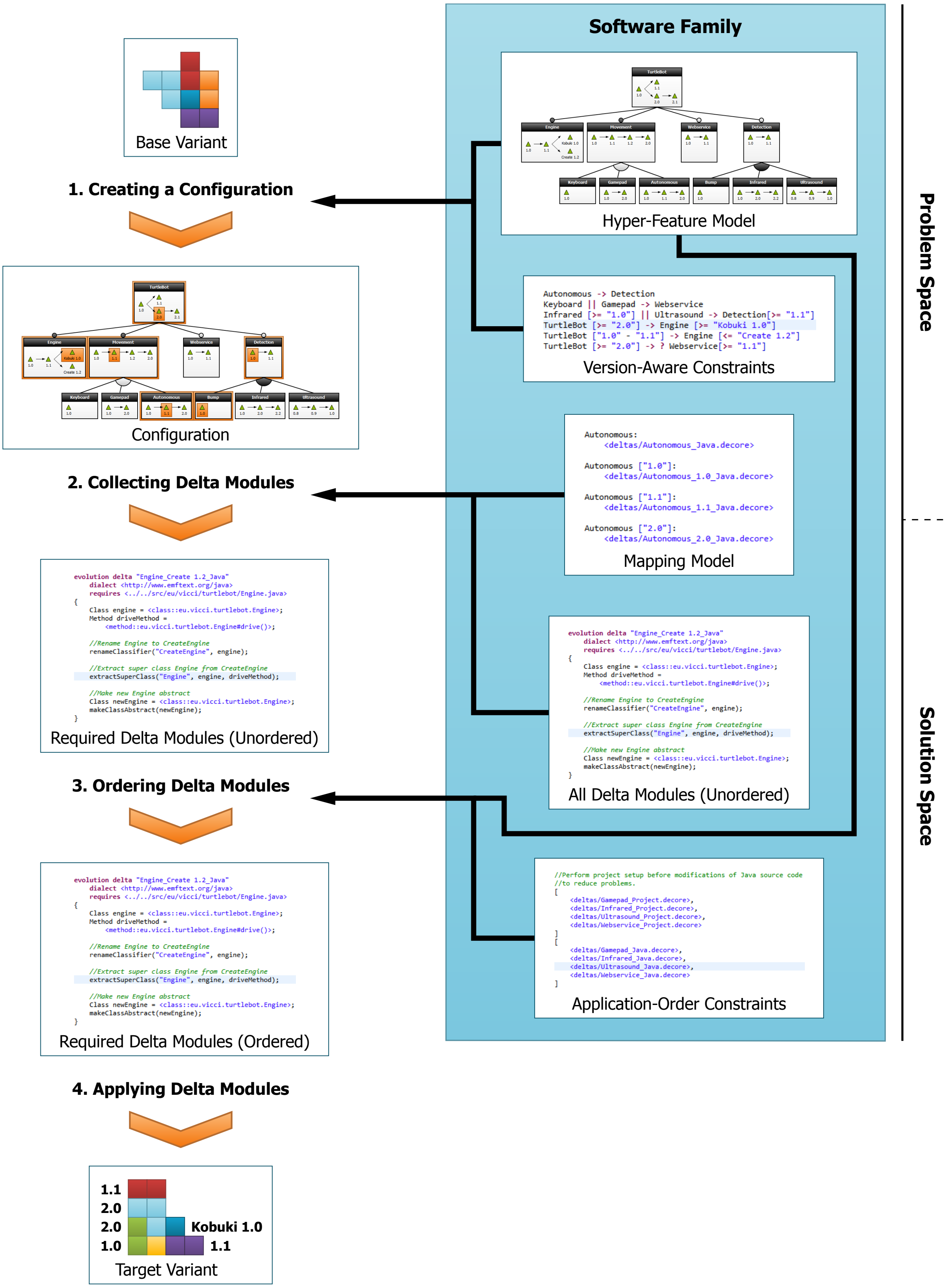
Figure 7.16.: Screenshot of the textual editor for the mapping of (combinations of) features and feature versions from an HFM to (sets of) delta modules within the DeltaEcore tool suite.

7.3. Deriving Variants

To capitalize on the integrated approach for managing variability in space and time, the artifacts created using the methods of Section 7.2 need to be utilized to derive concrete variants of a software family. This procedure consists of 4 steps: First, a configuration from the HFM has to be determined. Second, all delta modules relevant for that configuration have to be collected. Third, all order constraints for the delta modules have to be evaluated to determine a suitable application sequence. Fourth, all relevant delta modules and their respective delta operations have to be applied in order to transform the base variant of the software family into the variant representing the realization of the initially specified configuration including aspects of both variability in space and time. Figure 7.17 visualizes this process and the following sections explain how each of these steps is performed within the DeltaEcore tool suite.

7.3.1. Creating a Configuration

To define a configuration of an HFM, a subset of all features and versions has to be selected that is valid with regard to the configuration knowledge specified by the HFM and its accompanying version-aware constraints. Within the DeltaEcore tool suite, configurations are represented as models conforming to the metamodel presented in Figure 7.18. As both features and versions may appear in configurations of HFMs, there are metaclasses representing the selection of


Base Variant
1. Creating a Configuration
Configuration
TurtleBot
Engine
Kobuki 1.0
Create 1.2
Movement
Webservice
Detection
Keyboard
Gamepad
Autonomous
Bump
Infrared
Ultrasound
2. Collecting Delta Modules
Required Delta Modules (Unordered)
3. Ordering Delta Modules
Required Delta Modules (Ordered)
4. Applying Delta Modules
1.1
2.0
2.0
1.0
Kobuki 1.0
1.1
Target Variant
Software Family
Hyper-Feature Model
Version-Aware Constraints
Mapping Model
All Delta Modules (Unordered)
Application-Order Constraints
Problem Space
Solution Space

Required Delta Modules (Unordered):

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

Required Delta Modules (Ordered):

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

Version-Aware Constraints:

```
Autonomous -> Detection
Keyboard || Gamepad -> Webservice
Infrared [>= "1.0"] || Ultrasound -> Detection[>= "1.1"]
TurtleBot [>= "2.0"] -> Engine [>= "Kobuki 1.0"]
TurtleBot ["1.0" - "1.1"] -> Engine [<= "Create 1.2"]
TurtleBot [>= "2.0"] -> ? Webservice[>= "1.1"]
```

Mapping Model:

```
Autonomous:
    <deltas/Autonomous_Java.decore>

Autonomous ["1.0"]:
    <deltas/Autonomous_1.0_Java.decore>

Autonomous ["1.1"]:
    <deltas/Autonomous_1.1_Java.decore>

Autonomous ["2.0"]:
    <deltas/Autonomous_2.0_Java.decore>
```

All Delta Modules (Unordered):

```
evolution delta "Engine_Create 1.2_Java"
    dialect <http://www.emftext.org/java>
    requires <../../src/eu/vicci/turtlebot/Engine.java>
{
    Class engine = <class::eu.vicci.turtlebot.Engine>;
    Method driveMethod =
        <method::eu.vicci.turtlebot.Engine#drive()>;

    //Rename Engine to CreateEngine
    renameClassifier("CreateEngine", engine);

    //Extract super class Engine from CreateEngine
    extractSuperClass("Engine", engine, driveMethod);

    //Make new Engine abstract
    Class newEngine = <class::eu.vicci.turtlebot.Engine>;
    makeClassAbstract(newEngine);
}
```

Application-Order Constraints:

```
//Perform project setup before modifications of Java source code
//to reduce problems.
[
    <deltas/Gamepad_Project.decore>,
    <deltas/Infrared_Project.decore>,
    <deltas/Ultrasound_Project.decore>,
    <deltas/Webservice_Project.decore>
]
[
    <deltas/Gamepad_Java.decore>,
    <deltas/Infrared_Java.decore>,
    <deltas/Ultrasound_Java.decore>,
    <deltas/Webservice_Java.decore>
]
```

Figure 7.17.: Variant derivation procedure within DeltaEcore consisting of 4 steps with usage of the respective artifacts of the software family.

each type of element (DEFeatureSelection and DEVersionSelection, respectively). Each of these metaclasses extends the marker interface DEConfigurationArtifact that is used for all elements present in a configuration (DEConfiguration).

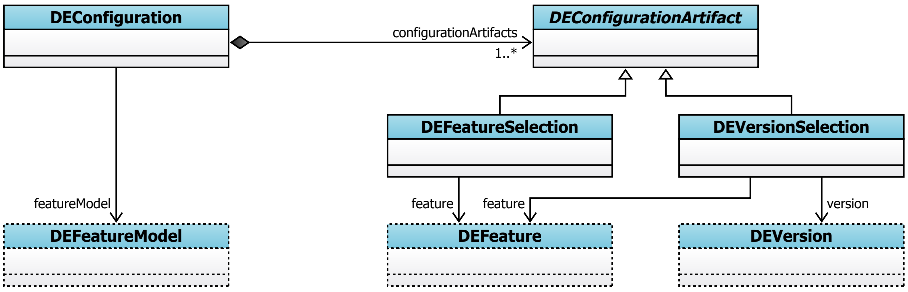
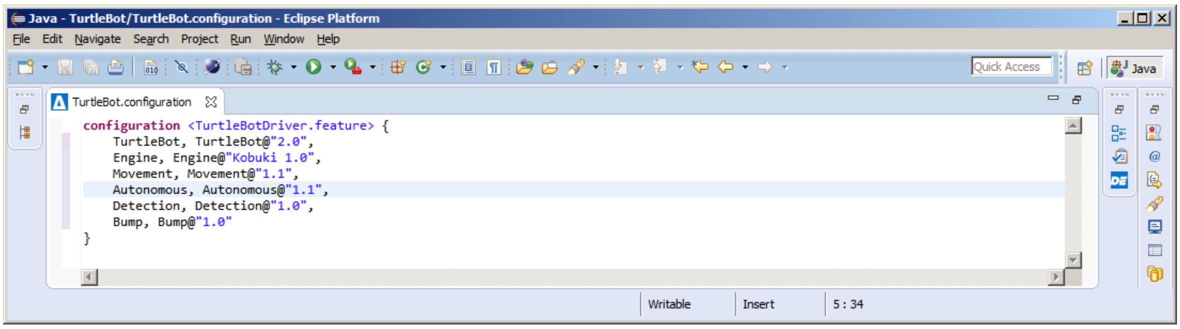


Figure 7.18.: Metamodel for configurations of HFMs formulated in EMF Ecore. The metaclasses DEFeatureModel, DEFeature and DEVersion are defined in Figure 7.7.

A configuration in DeltaEcore may be specified either textually or graphically. For the textual definition of a configuration, the metamodel of Figure 7.18 is perceived as abstract syntax of a textual language. It is further augmented with the concrete textual syntax specified in Listing 7.14 for the tool EMFText in order to create a suitable editor capable of parsing textual input to the respective configuration models. Figure 7.19 shows an example of using the textual editor.

```
SYNTAXDEF configuration
FOR <http://deltaecore.org/feature/configuration/1.0>
START DEConfiguration

RULES {
  DEConfiguration ::= "configuration" featureModel['<','>'] "{"
    configurationArtifacts ("," configurationArtifacts)*
  "}";

  DEFeatureSelection ::= (feature['"', '"'] | feature[]);
  DEVersionSelection ::= (feature['"', '"'] | feature[]) "@" version['"','"'];
}
```

Listing 7.14: Concrete syntax for the metamodel for configurations of HFMs from Figure 7.18 as specified with EMFText.


Java - TurtleBot/TurtleBot.configuration - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Quick Access
Java
TurtleBot.configuration

```
configuration <TurtleBotDriver.feature> {
    TurtleBot, TurtleBot@"2.0",
    Engine, Engine@"Kobuki 1.0",
    Movement, Movement@"1.1",
    Autonomous, Autonomous@"1.1",
    Detection, Detection@"1.0",
    Bump, Bump@"1.0"
}
```

Figure 7.19.: Screenshot of using the textual language to define configurations within the DeltaEcore tool suite.

Alternatively, it is further possible to specify the model of a configuration visually. For this purpose, the graphical editor for HFMs offers a configuration mode. In this mode, individual elements of a configuration (i.e., features or feature versions) may be added or subtracted by clicking the respective visual representation. Users are aided by automatically adding immediately required elements to a configuration, e.g., when adding a feature, its parent feature is (transitively) added to the configuration as well. Figure 7.20 shows an example of using the graphical editor in configuration mode.

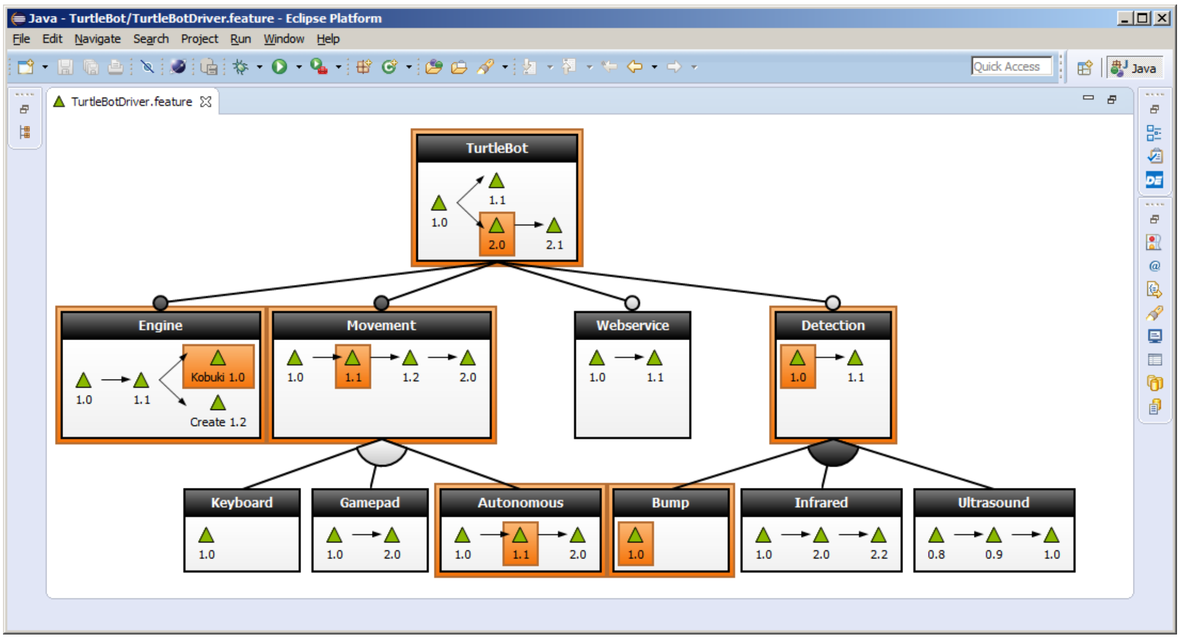


Figure 7.20.: Screenshot of using the configuration mode of the graphical editor for HFMs to define configurations within the DeltaEcore tool suite.

Regardless of the utilized means for specifying a configuration, the created model of the configuration is checked for validity regarding the constraints imposed on a valid configuration in Definition 17. For this purpose, the configuration is validated against the models for HFM and version-aware constraints. Violations of configuration constraints are reported to users by the editor.

Furthermore, it is possible to utilize the automatic version selection procedure presented in Section 6.3 in order to complete a partial configuration consisting of a valid pre-selection of features with a suitable constellation of versions. For this procedure, the models of the HFM, the version-aware constraints and the selected pre-configuration are translated to a suitable input format for the CSP solver Choco [JRL^{+}08]. The active objective function (see Section 6.3) governs the means of what constitutes an optimal solution. The CSP solver enumerates all possible solutions for valid version constellations, but only maintains the one with the currently best value with regard to the objective function. When completing the procedure, the result is a constellation of versions that is optimal with regard to the objective function and that obeys both the rules of the HFM as well as the version-aware constraints. When adding these versions to the pre-configuration of features, a valid configuration can be retrieved. Hence, when using the automatic version selection procedure, only the dimension for variability in space has to be considered in the manual configuration process and the dimension for variability in time is handled automatically.

7.3.2. Collecting Delta Modules

Depending on the concrete configuration, a set of relevant delta modules may be determined, which needs to be applied in order to create the respective variant. To collect all (transitively) required delta modules, an iterative procedure with 3 steps is performed:

First, the mapping model is processed in the sense that, for all mapping entries, the condition is evaluated for the features and feature versions included in the configuration. For this purpose, the semantics defined in Definition 19 for the constructs of the version-aware constraint language is used to evaluate the conditions of mapping entries. For each condition that is satisfied, the set of delta modules in the conclusion of the mapping entry is added to the set of all relevant delta modules.

Second, the structure of the HFM is evaluated. Each feature version demands its predecessor version on the same development line to have been applied beforehand. Hence, the delta modules directly associated with the predecessor version are relevant for variant derivation as well. In consequence, the mapping model is again evaluated for mapping entries that solely use a relative version restriction demanding the version to be equal to one of the transitively reachable predecessor versions. The respective delta modules from the conclusion of the mapping entry are added to the set of relevant delta modules.

Third, each delta module in the set of relevant delta modules has its requires relations evaluated. If the target of one of the requires relations specified within the delta module is a delta module itself (as opposed to a realization artifact), it is added to the set of relevant delta modules. This step is repeated until the set of relevant delta modules does not change in anymore, which signals that all required delta modules have been determined.

7.3.3. Ordering Delta Modules

The delta modules required to create one variant may not necessarily be applied in an arbitrary order as demands on the sequence of application may exist. The potential order in which the required delta modules may be applied can be affected by 3 factors:

1. the structure of the HFM,
2. the requires relations specified internal to the delta modules and
3. the application-order constraints specified external to the delta modules.

The HFM imposes constraints on the order of delta modules in two ways (see Section 6.4): First, delta modules of an initial version require the delta modules of their defining feature. Second, delta modules of a version require the delta modules of their predecessor version. For both these rules, an “apply after” relation is created from each delta module to the one that has to be applied before.

Furthermore, each block of a delta module may define multiple other delta modules that have to be applied before the containing delta module within a requires relation. This implicitly creates an order constraint that is captured by introducing “apply after” relations from the requiring to all required delta modules.

Finally, the application-order constraint model (see Section 7.2.4) may define constrained groups of delta modules that need to be applied after the delta modules of their previous constrained group (and, thus, transitively after all predecessor groups). For all delta modules of one constrained group, “apply after” relations are introduced to all delta modules of the previous constrained group.

These “apply after” relations may be used to determine a suitable application sequence in which the delta modules are applied to retrieve the concrete variant for the configuration.

For this purpose, the respective delta modules are perceived as nodes of a graph connected by edges formed by the "apply after" relation. As a result, establishing a suitable application sequence for delta modules may be solved by performing a topological sorting and choosing one possible path through the graph.

Algorithmically, topological sorting is performed as follows [Kah62]: Those nodes are selected from the partial order that do not have any incoming "apply after" relations. These nodes are appended to a result list (in arbitrary order) and their outgoing edges are removed from the partial order. The process is repeated with the reduced number of delta modules and "apply after" relations of the altered partial order until no more nodes to process are present. After the algorithm terminates, the result list contains one valid sequence of all possible ones described by the partial order on the delta modules spanned by the structure of the HFM, requires relations and application-order constraints.

7.3.4. Applying Delta Modules

With the determined application sequence of the relevant delta modules, it is possible to apply the delta modules and their contained calls to delta operations to create the target variant associated with the initially specified configuration.

First, each delta module has its requires relations processed in the sense that all required realization artifacts (i.e., not the required delta modules) are determined and collected in a set.

Second, the artifacts within this set, as well as the artifacts of the base variant, are loaded collectively to ensure proper resolution of cross references and to avoid multiple instances of the same artifact being present in memory, which would lead to inconsistencies during transformation.

Third, the determined order of delta modules is used to process each individual delta module. For each delta module, the contained blocks are processed sequentially. Within each block, the statements of the body are evaluated sequentially as well. Identifiers used to retrieve elements from the altered realization asset as part of statements are resolved using the identifier resolver of the delta dialect or the standard mechanism of using EMF Ecore IDs, if no identifier resolver was specified (see Section 7.1.3). Interpreting calls to delta operations leads to transformations of the base model. The concrete semantics of individual delta operations are determined by the respective delta interpreter, which performs a transformation of the model elements supplied as arguments in a dedicated Java method to interpret a particular delta operation (see Section 7.1.3).

Performing this operation for all blocks of all delta modules in the determined sequence yields the target variant for the initially specified configuration. The variant contains the realization of features and feature versions as manifestation of variability in space and time according to the transformations of the base variant specified in the relevant delta modules.

8. Evaluation

The evaluation demonstrates the feasibility of the integrated approach for managing variability in space and time presented within this thesis. For this purpose, three individual case studies are performed: First, the TurtleBot driver software used as running example throughout the thesis is inspected. In contrast to the case studies performed for the individual contributions of Chapter 4, Chapter 5, Chapter 6, this case study examines suitability of the integration of the individual contributions and considers realization artifacts of all languages mentioned in Chapter 1 including the 4 languages for safety certification SFT, CFD, CL and the GSN. Second, variability in space and time of a metamodel family for role-based modeling and programming languages is handled using the integrated approach of the thesis. Third, an SPL of feature modeling notations and their respective constraint languages is subjected to the presented concepts.

The goal of the evaluation is to determine suitability of the integrated approach for managing variability in space and time of the thesis by finding answers to the following research questions:

RQ1 Variability Model
Is it possible to adequately capture the conceptual configuration knowledge for variability in space and time using HFMs and the version-aware constraint language?

RQ2 Variability Realization Mechanism
Is it possible to adequately represent the manifestation of changes associated with variability in space and time using both the introduced language generation facilities to create delta languages as well as the created delta languages to specify configuration and evolution delta modules?

RQ3 Variant Derivation Procedure
Is it possible to use the variant derivation procedure to create products of the software family encompassing combinations of different features and their versions?

Each of the case studies first explains the general setting of the addressed problem. Then, it elaborates on variability in space within the respective software families. After that, it discusses the effects of variability in time caused by evolution of the software family. Finally, it demonstrates how both these dimensions can be managed with the integrated approach of this thesis. The results of the individual case studies and especially the discussion of their suitability with regard to the posed research questions are summarized in Section 8.4 as last part of the evaluation.

8.1. Configurable TurtleBot Driver Software

The TurtleBot is a domestic service robot mostly used to collect and deliver small items (see Chapter 1). To control the robot's operation, a driver software is required to execute basic functions (e.g., moving in particular directions). Due to the various possible configurations of the robot and its software, a variety of different components and functionality has to be supported. Furthermore, revisions of the robot's design make it necessary to support various versions of hardware components serving similar purposes (e.g., different engines). When considering the limited resources of the robot, such as CPU speed or battery life, it seems feasible to not provide one monolithic driver but rather custom-tailored drivers for each robot. In consequence, the driver software for the TurtleBot developed by the software development group of TU Dresden is perceived not as a single software but as a software family. Due to the effects of different configuration options as well as hardware revisions as part of evolution, the driver is subject to both variability in space and time.

Due to these reasons, the TurtleBot driver software has been used both as running example of the thesis and as subject of the case studies performed in Chapter 4, Chapter 5 and Chapter 6. In contrast to the previous case studies, which demonstrated feasibility of the respective presented contributions in isolation, the case study of this section inspects the combination of the individual contributions to an integrated approach for managing variability in space and time in software families. For the sake of completeness, the basic properties of the TurtleBot driver software regarding variability in space and time are briefly recaptured in the following sections before discussing their integrated management. Table 8.1 lists the essential characteristics of the case study.

Monitored Period	1.5 years	**Delta Languages**	7
Features	11	**Delta Modules**	46
Versions	29	**Configuration Delta Modules**	15 (33%)
Constraints	6	**Evolution Delta Modules**	31 (67%)

Table 8.1.: Essential characteristics of the case study on the configurable TurtleBot driver software.

8.1.1. Variability in Space

On a conceptual level, variability in space of the TurtleBot driver is represented by the feature model depicted in Figure 8.1. The feature `TurtleBot` represents the driver's core functionality and the mandatory `Engine` is responsible for providing low-level access to locomotion functionality. Furthermore, `Movement` groups different options for logical control over the robot's movement, which may be used as alternatives. It is possible to control the robot remotely by `Keyboard` or `Gamepad` as well as to use `Autonomous` driving of the robot to a specified target. An optional `Webservice` allows communication with the robot over WiFi. Furthermore, various means of `Detection` for obstacles in the way of the robot are presented that may be used in arbitrary combinations such as the `Bump`, `Infrared` and `Ultrasound` sensors. For safe operation of the robot in autonomous mode, an obstacle detection mechanism is mandatory, as expressed by the constraint $\text{Autonomous} \rightarrow \text{Detection}$. Finally, remote control over the robot demands usage of the webservice to connect to the robot via WiFi as expressed by the constraint $\text{Keyboard} \vee \text{Gamepad} \rightarrow \text{Webservice}$.

On a realization level, the TurtleBot driver software is realized by artifacts of 7 different languages: First, the deployed source code is written in Java. Second, the source code is organized in Eclipse project files (which utilize an XML dialect) that have to be altered depending on the provided configuration. Third, documentation of the driver software is written in DocBook markup that may later be used to create PDF documents. In addition, there are 4 notations used to specify safety certification material in order to document that the operation of the robot is considered sufficiently safe for particular scenarios: Software Fault Trees (SFTs), Component Fault Diagrams (CFDs), Checklists (CLs) and the Goal Structuring Notation (GSN) (see Section 1.3).

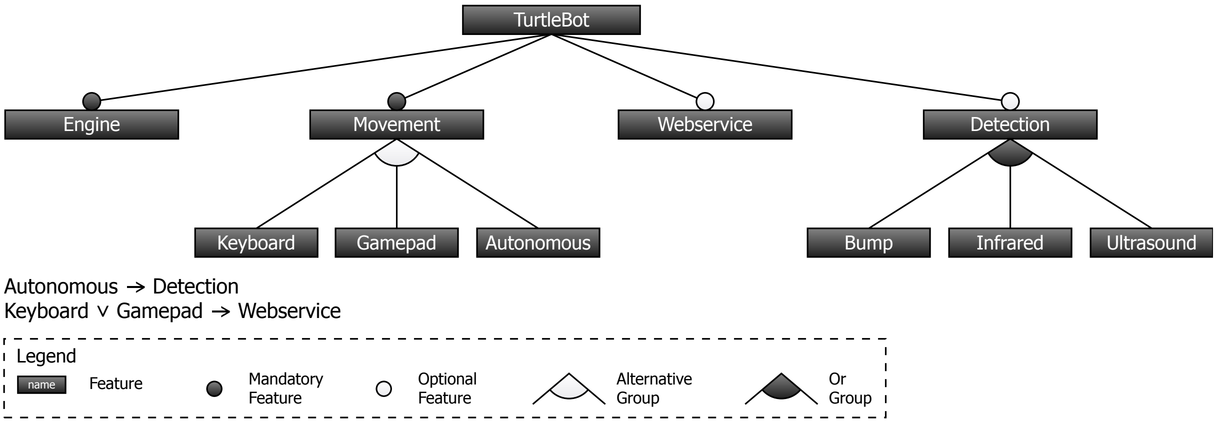


Figure 8.1.: Feature model of the configurable TurtleBot driver software.

Artifacts of each of these languages are subjected to variability in space as they are affected by configuration of the driver software: Java source code realizes different functionality which may entail a change of the project setup within Eclipse. Furthermore, the documentation material written in DocBook markup has to explain the actually configured functionality. Finally, safety certification material consisting of SFTs, CFDs, CLs and the GSN has to reflect the logic of the currently configured driver. To manifest the changes associated with variability in space within these artifacts, delta languages for all 7 source languages were created with the language generation facilities presented in Chapter 5 (see below). The respective delta dialects can be found in Appendix B.1, Appendix B.2, Appendix B.3, Appendix B.4, Appendix B.5, Appendix B.6 and Appendix B.7.

8.1.2. Variability in Time

Besides variability in space resulting from configuration, the driver software for the TurtleBot is further subject to variability in time resulting from evolution. By implementing new functionality or fixing existing defects, realization assets are altered, which creates new versions. Some of the new versions entail altered dependencies on artifacts or create incompatibilities with them. In consequence, the effects of variability in time affect the configuration knowledge on both a realization as well as a conceptual level. To address this issue, the configuration knowledge is modeled conceptually within an HFM as well as manifested in realization assets using suitable evolution delta modules.

Figure 8.2 repeats the HFM from Figure 6.6, which depicts the most recent state of configuration knowledge for the TurtleBot driver. In addition to the features representing variability in space, feature versions were added for all features representing variability in time. Furthermore, various version-aware constraints capture dependencies and incompatibilities of combinations of features and version ranges.

The evolution of the TurtleBot driver can roughly be grouped into 4 stages as marked by the annotation in Figure 8.2: The first stage introduced the features `Keyboard`, `Webservice` and `Infrared`. The second stage introduced the features `Gamepad` and `Ultrasound`. Furthermore, it performed updates of `Engine`, `Movement`, `Autonomous`, `Detection` and `Infrared`. The third stage updated `TurtleBot`, `Movement`, `Gamepad` and `Ultrasound`. Finally, the fourth stage performed updates of `TurtleBot`, `Engine`, `Movement`, `Autonomous`, `Infrared` and `Ultrasound`. Furthermore, the first 3 stages also had an effect on the version-aware constraints as depicted in Figure 8.2.

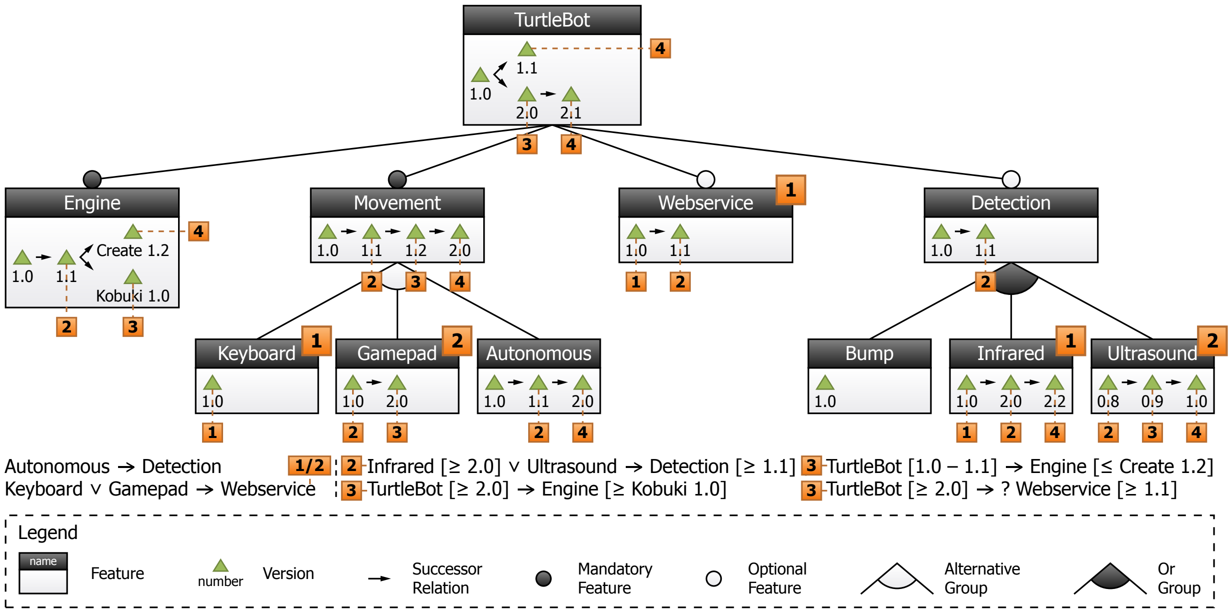


Figure 8.2.: HFM of the configurable TurtleBot driver software. Features, feature versions and version-aware constraints are annotated with the evolution stage they were introduced.

Along with these changes on the conceptual level, modifications were performed on the realization assets of the TurtleBot driver as part of variability in time within each evolution stage. To manifest the respective changes in realization assets in accordance with the feature versions of the HFM, appropriate evolution delta modules were specified. Apart from the delta operations also used to handle variability in space, there were operations that had to be specified exclusively for addressing variability in time, such as change of names (being part of identifiers) or certain semantic preserving refactorings, such as extracting a Java class.

8.1.3. Integrated Management of Variability in Space and Time

Using the approach presented within this thesis, it was possible to integrate both these dimensions of a software family with variability in space and time. For this purpose, combinations of features and feature versions of the HFM of Figure 8.2 were mapped to sets of configuration and evolution delta modules, respectively. Within the case study, a total of 46 delta modules was created out of which 15 were configuration delta modules and 31 were evolution delta modules. These numbers are equivalent to those of the case study presented in Section 6.6, even though it used merely 2 delta languages in contrast to the 7 of this case study. The lack of a difference in number of delta modules is due to the fact that the DeltaEcore tool suite allows specification of multiple blocks within a single delta module that each may use a different delta language. Hence, logically cohesive changes on realization artifacts of different source languages can be expressed within a single delta module. As the safety certification material specified using SFTs, CFDs, CLs and the GSN, as well as the documentation material in DocBook, are directly related to the realization within Java, all changes could be integrated into previously existing delta modules. Hence, no new delta modules were required even though additional realization assets of 5 source languages were considered. A variant may be created by selecting a configuration from the HFM, determining required delta modules as well as their order and applying their delta operations sequentially to a base variant in order to retrieve a target variant with variability in space and time as described in Chapter 6.

The base variant of the TurtleBot driver software consists of various artifacts of the aforementioned 7 source languages. As example, an excerpt of the Java class `Engine` is presented in Listing 8.1 as it appears within the base variant.

```java
package eu.vicci.turtlebot;

// ...

public class Engine {
  private boolean connected;

  // ...

  public Engine() {
    connected = false;
  }

  public void connect(CommPortIdentifier id) {
    // ... (performing connect)
    connected = true;
  }

  // ...

  public void drive() {
    // ... (driving with defined settings)
  }

  public void disconnect() {
    // ... (performing disconnect)
    connected = false;
  }
}
```

Listing 8.1: Excerpt from the base variant of the Java class `Engine` of the configurable TurtleBot driver software.

As the HFM specified multiple versions of the feature `Engine` associated with this class, multiple evolution delta modules were specified realizing the modifications of the respective versions. As an example, Listing 8.2 shows the evolution delta module for version *1.1* of the `Engine`, which adds functionality to the class that allows querying the current logical driving state of the engine as perceived by the robot. For this purpose, an enumeration with the distinguished driving states is added in Lines 7–10. Furthermore, a field holding the actual driving state is created in Line 12. Finally, a method is added to retrieve the field's value in Lines 14–16. The remainder of integrating the driving-state logic is omitted from the example.

In the subsequent version *Create 1.2* of the `Engine`, this state of the class `Engine` is altered further. Listing 8.3 shows the delta module associated with this version. It adds functionality to address both the LEDs and the speaker that are built into the platform to allow for basic communication of the robot (Lines 19–34). As this functionality is specific to the Create engine, the class structure was refactored to separate the more general functionality and to encapsulate it in a separate class. For this purpose, the original class was renamed to `CreateEngine` (Line 9) before a super class containing the `drive()` method was extracted, which again was called `Engine` (Line 12) and which was made abstract (Lines 15–16).

```
evolution delta "Engine_1.1_Java"
  dialect <http://www.emftext.org/java>
  requires <../src/eu/vicci/turtlebot/Engine.java>
{
  Class c = <class::eu.vicci.turtlebot.Engine>;

  createInternalEnumeration("public enum DrivingStates {
                               Stopped, Endless, DriveToTicksForward, DriveToTicksBackward,
                               SpinToTicksLeft, SpinLeft, SpinToTicksRight, SpinRight
                             }", c);

  createField("private DrivingStates drivingState;", c);

  createMethod("public DrivingStates getDrivingState() {
                  return drivingState;
                }", c);

  //...
}
```

Listing 8.2: Example evolution delta module that updates the `Engine` class to version *1.1* of the `Engine` feature.

```
evolution delta "Engine_Create 1.2_Java"
  dialect <http://www.emftext.org/java>
  requires <../src/eu/vicci/turtlebot/Engine.java>
{
  Class engine = <class::eu.vicci.turtlebot.Engine>;
  Method driveMethod = <method::eu.vicci.turtlebot.Engine#drive()>;

  //Rename Engine to CreateEngine
  renameNamedElement("CreateEngine", engine);

  //Extract super class Engine from CreateEngine
  extractSuperClass("Engine", engine, [driveMethod]);

  //Make new Engine abstract
  Class newEngine = <class::eu.vicci.turtlebot.Engine>;
  setAbstractModifier(true, newEngine);

  //Add additional components
  Class createEngine = <class::eu.vicci.turtlebot.CreateEngine>;

  createImport("import eu.vicci.turtlebot.components.Led;", createEngine);
  createImport("import eu.vicci.turtlebot.components.Sound;", createEngine);

  createField("private Led led;", createEngine);
  createField("private Sound sound;", createEngine);

  Constructor constructor = <constructor::eu.vicci.turtlebot.CreateEngine#CreateEngine()>;
  implementConstructor(constructor, "original(); led = new Led(); sound = new Sound();");

  ClassMethod connectMethod = <method::eu.vicci.turtlebot.CreateEngine#connect(..)>;
  implementMethod(connectMethod, "original(); led.enable(); sound.enable();");

  ClassMethod disconnectMethod = <method::eu.vicci.turtlebot.CreateEngine#disconnect()>;
  implementMethod(disconnectMethod, "led.disable(); sound.disable(); original();");

  //...
}
```

Listing 8.3: Example evolution delta module that updates the `Engine` class to version *Create 1.2* of the `Engine` feature producing the class `CreateEngine` in the process.

Hence, applying these delta modules to the base variant of the `Engine` results in two classes `Engine` and `CreateEngine` with the latter being a subclass of the prior. As an example, Listing 8.4 depicts the class `CreateEngine` with the changes performed by the evolution delta modules manifested in Java code.

```java
package eu.vicci.turtlebot;

// ...

public class CreateEngine extends Engine {
  private boolean connected;

  // ...

  public CreateEngine() {
    connected = false;
    led = new Led();
    sound = new Sound();
  }

  public void connect(CommPortIdentifier id) {
    // ... (performing connect)
    connected = true;
    led.enable();
    sound.enable();
  }

  // ...

  public void disconnect() {
    led.disable();
    sound.disable();
    // ... (performing disconnect)
    connected = false;
  }

  public enum DrivingStates {
    Stopped, Endless, DriveToTicksForward, DriveToTicksBackward, SpinToTicksLeft, SpinLeft,
    SpinToTicksRight, SpinRight
  }

  private DrivingStates drivingState;

  public DrivingStates getDrivingState() {
    return drivingState;
  }

  private Led led;
  private Sound sound;
}
```

Listing 8.4: Excerpt from the variant of the Java class `CreateEngine` of the configurable TurtleBot driver software for version *Create 1.2* of the `Engine` feature.

Using these and the remaining delta modules along with the HFM, it was possible to derive a multitude of different variants containing combinations of features in different versions. As part of the case study, variants were derived for the state at the end of each evolution stage including the original state before the first evolution stage. Furthermore, variants were created for combinations of features and feature versions that had not been anticipated explicitly. These variants were, in part, determined by manually creating valid configurations and then generating the respective products. In addition, the automatic version selection procedure of Section 6.3 was employed to automatically complete a preselection of features to a valid configuration containing appropriate feature versions. The realization assets of all resulting variants were manually inspected for validity and subjected to a test suite checking their correct operation.

8.2. Metamodel Family for Role-Based Modeling and Programming Languages

Role-based modeling in the sense of Bachman [Bac73] has been proposed almost 40 years ago as a means to model complex and dynamic domains by capturing both context-dependent and collaborative behavior of objects. Various modeling [Bac73, Hal06, Gui05, MT08] and programming languages [BBT06, BGE07, Her03] have been defined to utilize roles. Even though the different approaches are founded on a common basic understanding of roles, they differ in supported modeling concepts and offered language constructs as well as the utilized implementation technology and are not based on a common metamodel. Within the approach, the term *Compartment* was chosen as an umbrella term for the competing terms *Context* [Dey01], *Environment* [ZZ06], *Institution* [BBT06], *Team* [Her05] and *Ensemble* [HK14], which are used throughout the literature to group those roles that are supposed to collaborate within one particular application instance.

On a conceptual level, the various different approaches to role modeling form a family of related software systems due to their significant similarities and explicitly defined variabilities. To also reflect this on a realization level, a software family was created [KLG$^+$14]. The single asset of the solution space of the software family is a metamodel based on EMF Ecore. The software family may be used to derive metamodels containing a selection of features that represents the different modeling concepts and language constructs along the dimension of variability in space. As the family of metamodels is also subject to further development as part of evolution, it also contains aspects of variability in time that can be captured as feature versions. Table 8.2 lists the essential characteristics of the case study.

Monitored Period	1.5 months
Features	48
Versions	106
Constraints	5

Delta Languages	1
Delta Modules	52
Configuration Delta Modules	28 (54%)
Evolution Delta Modules	24 (46%)

Table 8.2.: Essential characteristics of the case study on the metamodel family for role-based modeling and programming languages.

8.2.1. Variability in Space

Figure 8.3 presents the feature model of the software family of role-based modeling and programming languages. In total, the feature model defines 48 configurable features that affect availability of various modeling concepts and language constructs in the respective variants. The majority of features affects availability of elements in the metamodel and their concrete characteristics which both can be captured structurally. However, the features `DifferentRolesSimultanously`, `ByUnrelatedPlayers`, `RoleDependentPlayerFeatures` and `RoleInheritance` correspond to invariants for concrete models as instances of the metamodel [KLG$^+$14]. These invariants cannot be represented structurally in the metamodel, but would have to be defined as well-formedness rules, e.g., in the form of modeling constraints. Neither the original work on the

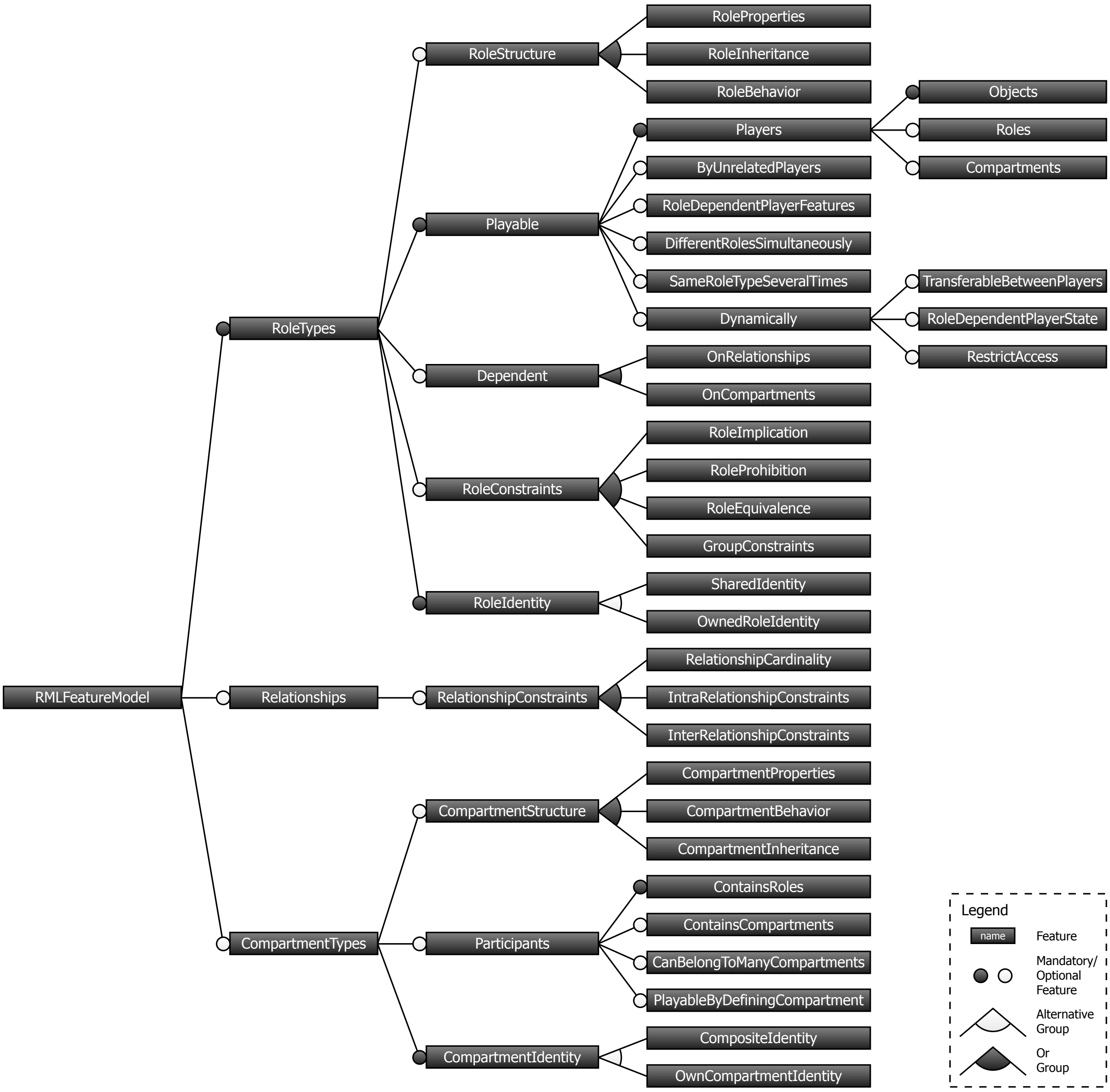


Figure 8.3.: Feature model of the metamodel family for role-based modeling and programming languages.

metamodel family [KLG$^+$14] nor the case study presented in this section support the realization of these features, but exclude them as future work.

The set of possible configurations described by the feature model is further restrained by a number of cross-tree constraints as listed in Formula 12. Constraints (1) to (4) consider the domain of role-based modeling and programming languages and stem from the original work in [KLG$^+$14]. Constraint (1) states that the dependence on relationships requires relationships to be part of a configuration. Likewise, constraint (2) states that the dependence on compartments requires compartments to be part of a configuration. Furthermore, constraint (3) ensures that configuration of role implications entails the configuration role equivalences. Finally, constraint (4) formulates that the use of role compartments as players requires the presence of compartment types.

Formula 12: Constraints of the feature model of the metamodel family for role-based modeling and programming languages presented in Figure 8.3.

(1) OnRelationships ↔ Relationships
(2) OnCompartments ↔ CompartmentTypes
(3) RoleImplication → RoleEquivalence
(4) Compartments → CompartmentTypes

8.2.2. Variability in Time

In the original presentation of the metamodel family for role-based modeling and programming languages [KLG$^+$14], only the dimension of variability in space was modeled explicitly. However, over the course of time, changes on the metamodel family had to be performed as part of evolution. At the time, these changes were tracked manually along with the respective affected features. With clients utilizing products of the metamodel family from different times, in order to provide ongoing compatibility, it is necessary to also support variability in time to create products of the metamodel for different versions of features. For this purpose, the tracked changes were realized by extending the software family's feature model to an HFM with feature versions and associated evolution delta modules that perform transformation of the metamodel. The original state of the metamodel family is represented by the initial version *2014-05-30* of each feature. The evolution of the metamodel family can roughly be distinguished into three stages that utilize a timestamp as designation: *2014-06-16*, *2014-06-17* and *2014-07-14*. Versions have a number equivalent to the designation of the respective evolution stage they were introduced in. Due to its size, the HFM of the metamodel family for role-based modeling and programming languages was arranged horizontally and split up into Figure 8.4 and Figure 8.5.

In total, 106 feature versions are present for the 48 features in the HFM of the metamodel family for role modeling. Further development of the original state of the metamodel family was performed by two tightly integrated developers. As a result of the strictly sequential nature of changes to the metamodel, the resulting feature versions of each feature are arranged sequentially along a single development line without branching. In detail, the original state of the metamodel family and the three evolution stages encompassed the following functionality and changes:

The **original state** of the metamodel allowed for selecting features with the rules defined in Figure 8.3. When defining a configuration with the features `OnRelationships`, `RoleConstraints`, `CompartmentInheritance`, `IntraRelationshipConstraint`, `InterRelationshipConstraint`, `RoleGroup` and `RoleInheritance`, EClasses with similar names are created. Furthermore, the respective incoming and outgoing EReferences of these classes are created with selection of the respective feature(s) of the reference ends. Additional EReferences are created or modified depending on combinations of various features, e.g., the `filler` EReference of `Fulfillment` points to either `NaturalType`, `RigidType` or the generic `Type` depending on combination of (de-)selection of features `Roles` and `Compartments`. Finally, the inheritance hierarchy in the metamodel may be altered by changing super types of various EClasses, e.g., the EClass `RoleType` only inherits (indirectly) from `Type` if at least one of the features `RoleProperties` and `RoleBehavior` is selected.

The **evolution stage *2014-06-16*** was mostly concerned with fixing minor defects of the original state of the metamodel family. The EEnumLiteral `InseperablePart` created by feature `OnRelationships` was renamed to "InseparablePart". The EClasses `IntraRelationshipConstraints` and `Irreflexiv` created by `IntraRelationshipConstraints`

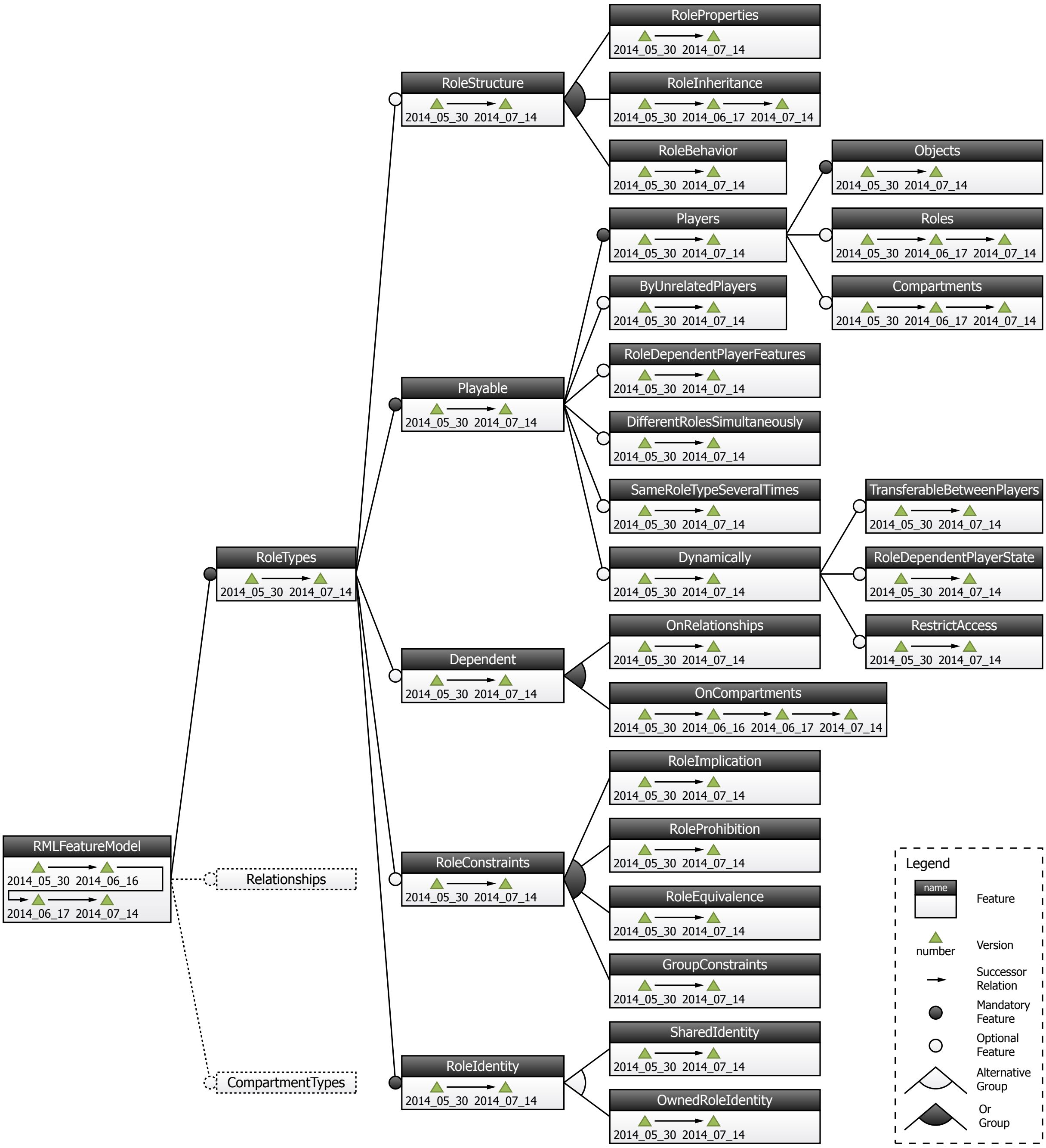


Figure 8.4.: HFM of the metamodel family for role-based modeling and programming languages (1/2).

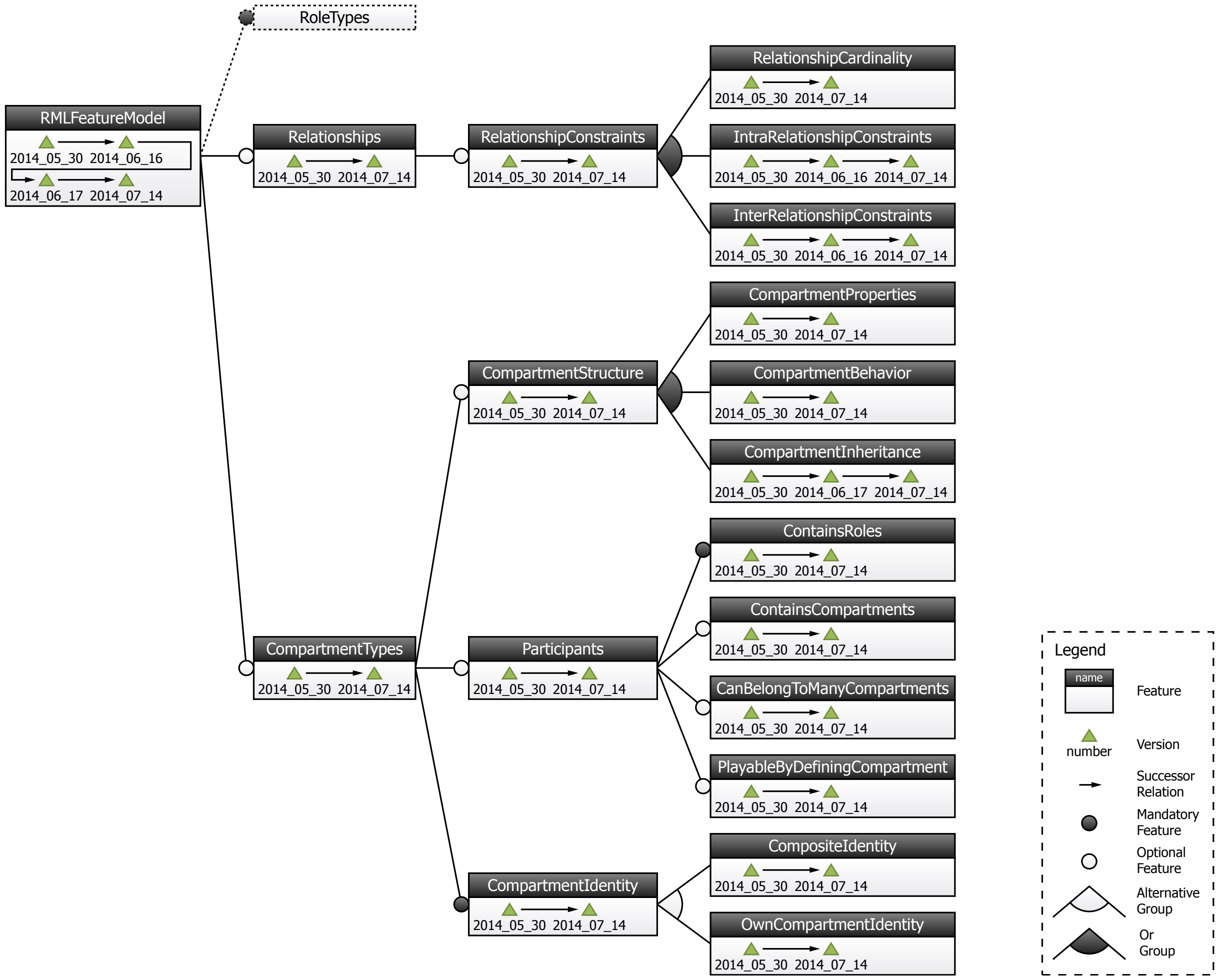


Figure 8.5.: HFM of the metamodel family for role-based modeling and programming languages (2/2).

were renamed to “IntraRelationshipConstraint” and “Irreflexive”, respectively. Furthermore, names of the EReferences `outcoming` of both EClasses `RelationTarget` and `Relation` were changed to “outgoing”.

The **evolution stage *2014-06-17*** was mostly concerned with fixing conceptual problems and shortcomings of the previous revisions of the metamodel family. With selection of both features `CompartmentInheritance` and `OnCompartments` simultaneously, two EReferences `super` and `sub` from EClass `CompartmentType` to EClass `CompartmentInheritance` are created. However, closer analysis revealed that the correct direction of these references should be from `CompartmentInheritance` to `CompartmentType`. As a result, the direction of the respective EReferences was inverted provided that feature `OnCompartments` is selected. Similar defects were identified in association with the feature `RoleInheritance` (between EClasses `RoleType` and `RoleInheritance`) as well as the root feature `RMLFeatureModel` (between EClasses `NaturalType` and `NaturalInheritance`). These problems were also solved by inverting the respective EReferences. Finally, the previously unaddressed combination of deselecting feature `Compartments` and selecting feature `Roles` was addressed by creating an abstract EClass `Player`, which inherits from `NaturalType` as well as `RoleType` and which serves as target of the EReference `filler` of EClass `Fulfillment`.

The **evolution stage *2014-07-14*** relocated all metamodel elements from an excess sub-package `crom_l1.CompartmentRoleObjectModel` to its parent package `crom_l1` and deleted the then redundant sub-package. This procedure had substantial impact on the metamodel family, as the change of the containing package inadvertently alters the hierarchical identifier in the qualified name of the affected elements. Due to this reason, previously specified references to metamodel elements were no longer resolvable.

In consequence, selection of feature version combinations of the different evolution stages has to be constrained. For this purpose, a baseline is established for evolution stage *2014-07-14*. Hence, in addition to the constraints of Formula 12, Formula 13 defines constraint (5) to establish a baseline to only allow selection of versions either before or after the *2014-07-14* evolution stage.

Formula 13: Version-aware constraint of the HFM of the metamodel family for role-based modeling and programming languages presented in Figure 8.4 and Figure 8.5.

(5) ?RMLFeatureModel [$\geq$ 2014-07-14] $\leftrightarrow$?RoleTypes [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleStructure [$\geq$ 2014-07-14] $\leftrightarrow$?RoleProperties [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleBehavior [$\geq$ 2014-07-14] $\leftrightarrow$?RoleInheritance [$\geq$ 2014-07-14] $\leftrightarrow$
?Playable [$\geq$ 2014-07-14] $\leftrightarrow$?Players [$\geq$ 2014-07-14] $\leftrightarrow$?Objects [$\geq$ 2014-07-14] $\leftrightarrow$
?Roles [$\geq$ 2014-07-14] $\leftrightarrow$?Compartments [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleDependentPlayerFeatures [$\geq$ 2014-07-14] $\leftrightarrow$
?DifferentRolesSimultaneously [$\geq$ 2014-07-14] $\leftrightarrow$
?SameRoleTypeSeveralTimes [$\geq$ 2014-07-14] $\leftrightarrow$?ByUnrelatedPlayers [$\geq$ 2014-07-14] $\leftrightarrow$
?Dynamically [$\geq$ 2014-07-14] $\leftrightarrow$?TransferableBetweenPlayers [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleDependentPlayerState [$\geq$ 2014-07-14] $\leftrightarrow$?RestrictAccess [$\geq$ 2014-07-14] $\leftrightarrow$
?Dependent [$\geq$ 2014-07-14] $\leftrightarrow$?OnCompartments [$\geq$ 2014-07-14] $\leftrightarrow$
?OnRelationships [$\geq$ 2014-07-14] $\leftrightarrow$?RoleConstraints [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleImplication [$\geq$ 2014-07-14] $\leftrightarrow$?RoleProhibition [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleEquivalence [$\geq$ 2014-07-14] $\leftrightarrow$?GroupConstraints [$\geq$ 2014-07-14] $\leftrightarrow$
?RoleIdentity [$\geq$ 2014-07-14] $\leftrightarrow$?SharedIdentity [$\geq$ 2014-07-14] $\leftrightarrow$
?OwnedRoleIdentity [$\geq$ 2014-07-14] $\leftrightarrow$?Relationships [$\geq$ 2014-07-14] $\leftrightarrow$
?RelationshipConstraints [$\geq$ 2014-07-14] $\leftrightarrow$?RelationshipCardinality [$\geq$ 2014-07-14] $\leftrightarrow$
?IntraRelationshipConstraints [$\geq$ 2014-07-14] $\leftrightarrow$
?InterRelationshipConstraints [$\geq$ 2014-07-14] $\leftrightarrow$?CompartmentTypes [$\geq$ 2014-07-14] $\leftrightarrow$
?CompartmentStructure [$\geq$ 2014-07-14] $\leftrightarrow$?CompartmentProperties [$\geq$ 2014-07-14] $\leftrightarrow$
?CompartmentBehavior [$\geq$ 2014-07-14] $\leftrightarrow$?CompartmentInheritance [$\geq$ 2014-07-14] $\leftrightarrow$
?Participants [$\geq$ 2014-07-14] $\leftrightarrow$?ContainsRoles [$\geq$ 2014-07-14] $\leftrightarrow$
?ContainsCompartments [$\geq$ 2014-07-14] $\leftrightarrow$
?CanBelongToManyCompartments [$\geq$ 2014-07-14] $\leftrightarrow$
?PlayableByDefiningCompartment [$\geq$ 2014-07-14] $\leftrightarrow$
?CompartmentIdentity [$\geq$ 2014-07-14] $\leftrightarrow$?CompositeIdentity [$\geq$ 2014-07-14] $\leftrightarrow$
?OwnCompartmentIdentity [$\geq$ 2014-07-14]

8.2.3. Integrated Management of Variability in Space and Time

To manifest the changes of features and feature versions, both configuration and evolution delta modules were employed to affect the elements provided in the metamodel for role-based modeling and programming languages. Delta operations were used to add, modify and remove elements of the metamodel. The base variant for these transformations is the metamodel depicted in Figure 8.6. To provide a suitable delta language to perform these changes, 27 delta operations[1] were defined to manipulate Ecore metamodels. The respective delta dialect can be found in Appendix B.8.

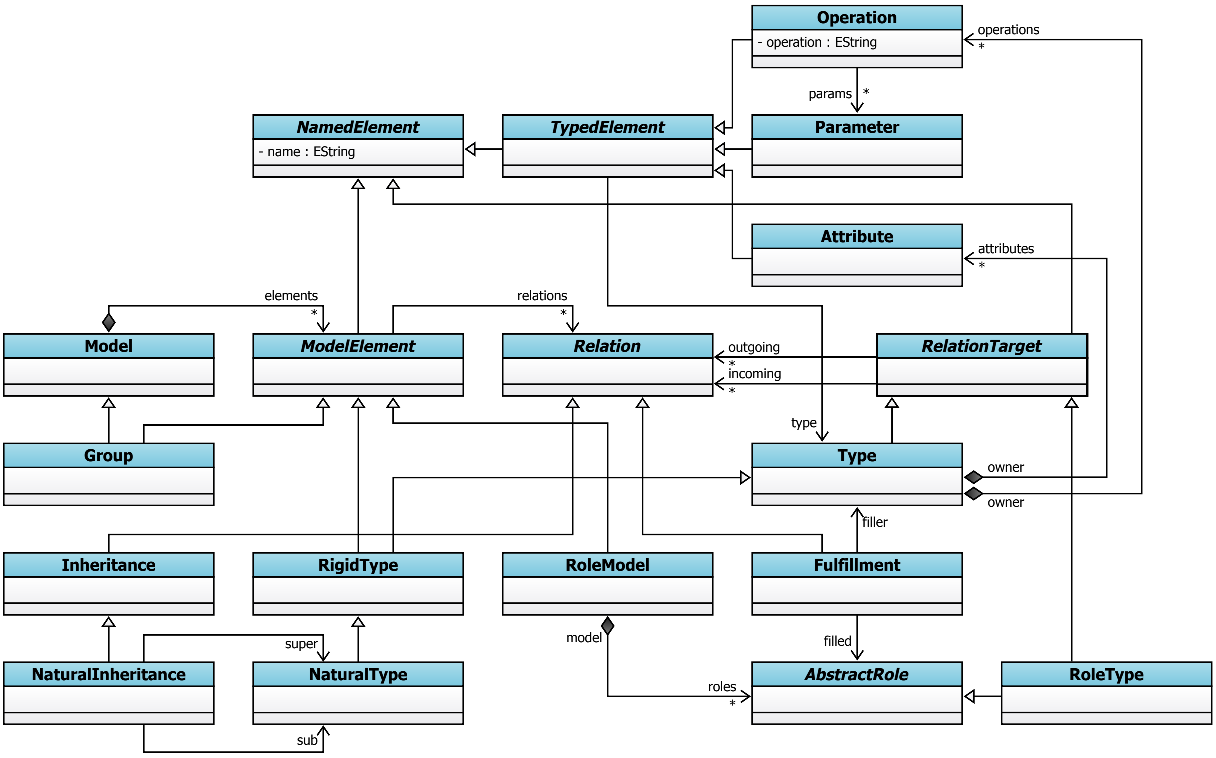


Figure 8.6.: Base variant of the metamodel family for role-based modeling and programming languages.

Using the language DeltaEcore, a total of 52 delta modules was defined for the metamodel family. Of these delta modules, 28 served the purpose of configuration and the remaining 24 were used for evolution. The difference of number of delta modules in comparison to the number of features and feature versions stems from three facts: First, some features are not realized in the metamodel by configuration delta modules. This is due to their nature as invariants on the models, which are specified external to the metamodel in EMF Ecore. Second, some of the feature versions of the *2014-07-14* evolution stage merely serve a logical purpose of establishing a baseline due to the extensive impact of modifications during that evolution stage. Third, some delta modules may be reused in different contexts, e.g., when two configuration delta modules perform (in part) similar modifications on the metamodel.

As an example of realizing changes associated with variability in space, Listing 8.5 shows an excerpt of the configuration delta module that enables the feature `OnRelationships`.

[1] The delta dialect presented in Listing 7.4 contains 29 delta operations. However, 2 of these operations were defined only later as part of the case study performed in Section 8.3.

```
configuration delta "OnRelationships"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/crom_l1.ecore>
{
  EPackage ePackage = <CompartmentRoleObjectModel>;

  //Create Relationship EClass
  EClass relationshipEClass = new EClass(name: "Relationship");
  addEClass(relationshipEClass, ePackage);
  addESuperType(relationshipEClass, <CompartmentRoleObjectModel.Relation>);


  //Create Parthood EEnum
  EEnum parthoodEEnum = new EEnum(name: "Parthood");
  addEEnum(parthoodEEnum, ePackage);

  EEnumLiteral unconstrainedEEnumLiteral =
    new EEnumLiteral(name: "Unconstrained", literal: "Unconstrained", value: 0);
  addEEnumLiteral(unconstrainedEEnumLiteral, parthoodEEnum);

  //...

  EEnumLiteral inseperablePartEEnumLiteral =
    new EEnumLiteral(name: "InseperablePart", literal: "InseperablePart", value: 4);
  addEEnumLiteral(inseperablePartEEnumLiteral, parthoodEEnum);

  EEnumLiteral shareablePartEEnumLiteral =
    new EEnumLiteral(name: "ShareablePart", literal: "ShareablePart", value: 5);
  addEEnumLiteral(shareablePartEEnumLiteral, parthoodEEnum);


  //Create Place EClass...
  EClass placeEClass = new EClass(name: "Place");
  addEClass(placeEClass, ePackage);

  //...and its EAttributes...
  EAttribute lowerEAttribute =
    new EAttribute(name: "lower", eType: <EInt>, lowerBound: 1, upperBound: 1);
  addEAttribute(lowerEAttribute, placeEClass);

  EAttribute upperEAttribute =
    new EAttribute(name: "upper", eType: <EInt>, lowerBound: 1, upperBound: 1);
  addEAttribute(upperEAttribute, placeEClass);

  EAttribute partEAttribute =
    new EAttribute(name: "part", eType: parthoodEEnum, lowerBound: 1, upperBound: 1);
  addEAttribute(partEAttribute, placeEClass);


  //...as well as incoming and outgoing EReferences
  EReference firstEReference = new EReference(eType: placeEClass, name: "first",
    lowerBound: 1, upperBound: 1, containment: false);
  addEReference(firstEReference, relationshipEClass);

  EReference secondEReference = new EReference(eType: placeEClass, name: "second",
    lowerBound: 1, upperBound: 1, containment: false);
  addEReference(secondEReference, relationshipEClass);


  EReference holderEReference = new EReference(eType: <CompartmentRoleObjectModel.RoleType>,
    name: "holder", lowerBound: 1, upperBound: 1, containment: false);
  addEReference(holderEReference, placeEClass);
}
```

Listing 8.5: Example configuration delta module that enables the feature `OnRelationships`.

As an example of the *2014-06-16* evolution stage, Listing 8.6 presents the evolution delta module used to update the feature `OnRelationships`.

```
evolution delta "OnRelationships_2014_06_16"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/crom_l1.ecore>
{
  EEnumLiteral literal = <CompartmentRoleObjectModel.Parthood.InseperablePart>;

  setEEnumLiteralLiteral("InseparablePart", literal);
  setEEnumLiteralName("InseparablePart", literal);
}
```

Listing 8.6: Example evolution delta module that updates the feature `OnRelationships` by correcting a mistake in the name of a previously created EEnumLiteral as part of the evolution stage 2014-06-16.

As an example of the *2014-06-17* evolution stage, Listing 8.7 presents the evolution delta module updating the base variant of the metamodel by inverting the EReferences `super` and `sub` between EClasses `NaturalType` and `NaturalInheritance`.

```
evolution delta "RMLFeatureModel_2014_06_17"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/crom_l1.ecore>
{
  //Reverse references by removing existing and then adding new references.
  removeEReference(<CompartmentRoleObjectModel.NaturalType.super>);
  removeEReference(<CompartmentRoleObjectModel.NaturalType.sub>);


  EClass naturalTypeEClass = <CompartmentRoleObjectModel.NaturalType>;
  EClass naturalInheritance = <CompartmentRoleObjectModel.NaturalInheritance>;

  EReference superEReference = new EReference(eType : naturalTypeEClass, name : "super",
    lowerBound : 1, upperBound : 1, containment : false);
  addEReference(superEReference, naturalInheritance);

  EReference subEReference = new EReference(eType : naturalTypeEClass, name : "sub",
    lowerBound : 1, upperBound : 1, containment : false);
  addEReference(subEReference, naturalInheritance);
}
```

Listing 8.7: Example evolution delta module that updates the base variant of the metamodel by inverting the EReferences `super` and `sub` between EClasses `NaturalType` and `NaturalInheritance` as part of the evolution stage 2014-06-17.

In the **evolution stage *2014-07-14***, the base package of all metamodel elements was changed by relocating elements from a nested sub-package `crom_l1.CompartmentRoleObjectModel` to their parent package `crom_l1` and deleting the then redundant sub-package. For this purpose, a number of evolution delta modules were defined that each move the elements previously created by the configuration delta module associated with the respective feature. Furthermore, an evolution delta module of the base variant was used to relocate all metamodel elements to the parent package and to delete the sub-package. For the latter to be unproblematic, it has to be ensured that all elements have been moved out of the sub-package before deletion. Hence, the evolution delta module of the base variant has to be applied as last evolution delta module of the evolution stage *2014-07-14*. For this purpose, respective application-order constraints were introduced to assure that all other evolution delta modules of the evolution stage *2014-07-14* were applied before the evolution delta module for the base variant, if they are required for a configuration due to selection of the respective feature versions. As an example of the *2014-07-14* evolution stage, Listing 8.8 presents the evolution delta module for feature `OnRelationships` that moves the previously created EClasses `Relationship` and `Place` as well as the EEnum `Parthood` from the sub-package `crom_l1.CompartmentRoleObjectModel` to the parent package `crom_l1`.

```
evolution delta "OnRelationships_2014_07_14"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/crom_l1.ecore>
{
  EClass relationshipEClass = <CompartmentRoleObjectModel.Relationship>;
  setEClassifierPackage(<crom_l1>, relationshipEClass);

  EEnum parthoodEEnum = <CompartmentRoleObjectModel.Parthood>;
  setEClassifierPackage(<crom_l1>, parthoodEEnum);

  EClass placeEClass = <CompartmentRoleObjectModel.Place>;
  setEClassifierPackage(<crom_l1>, placeEClass);
}
```

Listing 8.8: Example evolution delta module that updates the feature `OnRelationships` by moving the classifiers created for the feature to a different package as part of the evolution stage 2014-07-14.

When selecting a configuration consisting of features and feature versions from the HFM and applying the respective delta modules retrieved using the mapping model, it is possible to derive variants of the metamodel family with variability in space and time. Besides manually created configurations, the automatic version selection procedure of Section 6.3 was employed to determine appropriate versions for a pre-selection of features. Older versions would not necessarily have to be maintained for use as they contain obvious defects such as the inverted super/sub references described in evolution stage *2014-06-17*. However, in order to be able to process models created in conformance with previous revisions of the metamodel, these versions may need to be available (e.g., to load and update them to conform to a more recent version of the metamodel). The validity of the created variants was checked by loading instances of the metamodel that were specified at certain points in the development process. These models depend on the presence of certain features at particular versions so that a successful loading with a variant containing these features at the required versions substantiates validity of the created variants. Figure 8.7 depicts a variant of the metamodel presented in Figure 8.6 for the configuration containing all features except `OwnedRoleIdentity` and `OwnCompartmentIdentity`. For each selected feature, the respective *2014-07-14* version was selected to retrieve the most current revision.

8.3. A Software Product Line of Feature Modeling Notations and Constraint Languages

A software family encompasses a set of closely related software systems in terms of common and variable functionality. On a conceptual level, the entirety of all valid configurations

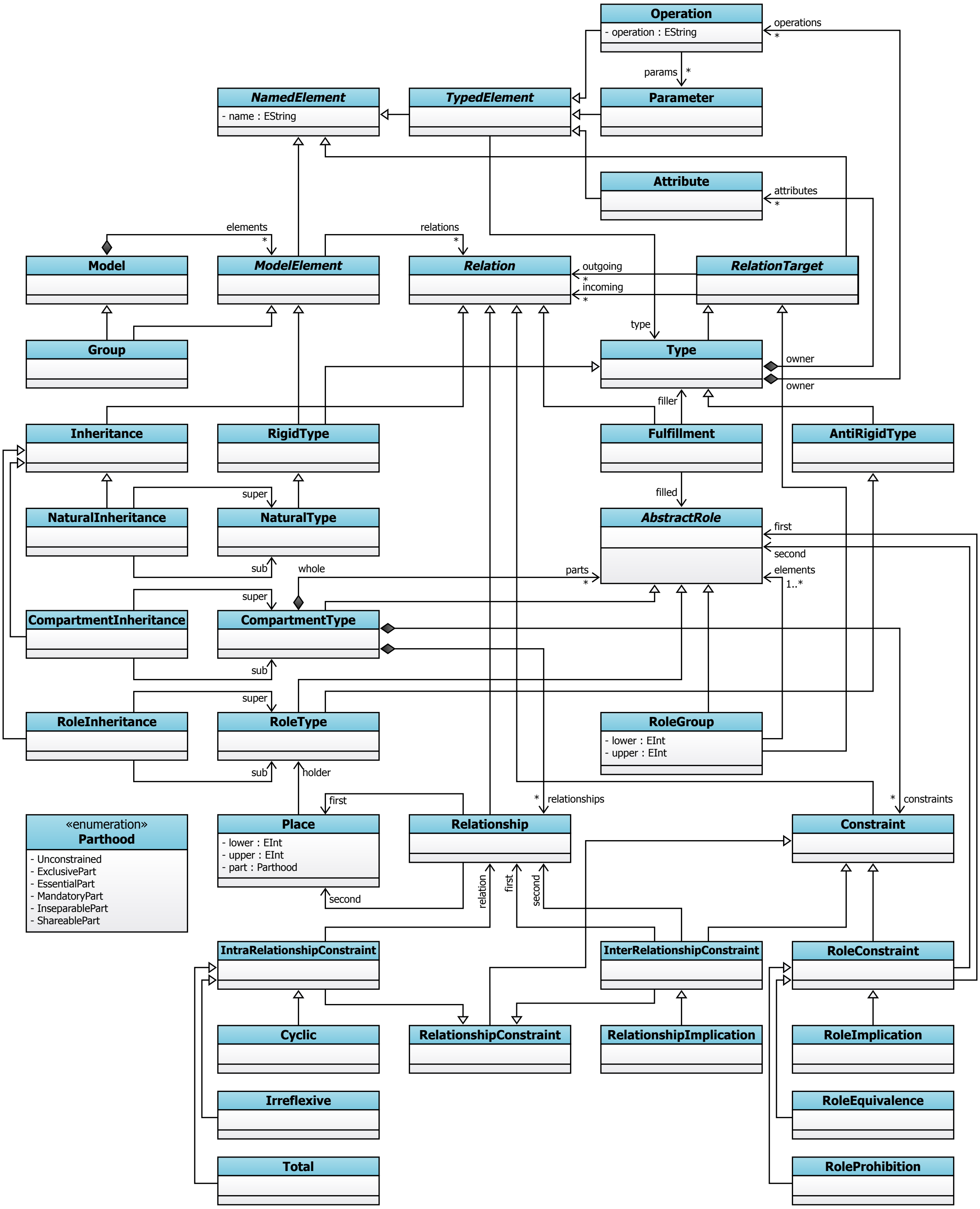


Figure 8.7.: Variant of the metamodel family for role-based modeling and programming languages for the configuration containing all features except `OwnedRoleIdentity` and `OwnCompartmentIdentity` at version *2014-07-14*.

may be captured in a variability model such as a feature model [KCH$^+$90, CE00, BRN$^+$13] with additional cross-tree constraints (see Section 2.2.1).

Ever since the introduction of feature models [KCH$^+$90], a great number of extensions has been made to the original language to address various needs, such as attributes for features with finite and infinite domain [CBUE02, CHE05, KOD10a], cardinalities for groups and individual features [RBSP02, CHE05, ME08, SSA14a] or configurable feature versions [SSA14a, SSA14c]. Likewise, languages for cross-tree constraints may be represented by different means, e.g., using various subsets of propositional logic [Bat05, KOD10b] or the Object Constraint Language (OCL) [CE00, CBUE02]. Furthermore, availability of various constructs of cross-tree constraint languages depends on the concrete notation employed for feature models, e.g., constraints over attributes are only possible with attributed feature models.

Even though variability models are essential for specifying configuration knowledge, this variability in the notations for feature models and cross-tree constraint languages makes it hard to exchange data between different tools, to reuse algorithms for analyses and to combine configuration knowledge from different sources, when the respective notations are defined individually. However, despite the differences in the languages for feature models and cross-tree constraints, the languages encompass a significant level of commonality.

Within the case study, this fact is exploited in order to devise an SPL modeling feature models and cross-tree constraint languages. For this purpose, a feature model is bootstrapped to conceptually capture the configuration knowledge regarding variability in space and then extended to an HFM also representing variability in time regarding the changes of the realization assets. Furthermore, SPL techniques are used to allow configuration of various different concrete languages for feature models and cross-tree constraints.

The approach is based on the state of the art in various works on feature modeling to create a family of feature modeling notations that has similar expressiveness to those of the original approaches. Table 8.3 analyzes 15 selected approaches from the state of the art in feature modeling (including constraint languages) and categorizes them by a number of distinctive characteristics.

For these findings, both conceptual configuration knowledge as well as a concrete model-based realization are presented. The realization consists of 3 main artifacts: For the feature modeling notations, an EMF Ecore metamodel defines valid modeling concepts. Furthermore, for the constraint languages, another Ecore metamodel is provided that encompasses all permissible language constructs. When using a textual representation for constraints (see below), the concepts of this metamodel may be perceived as abstract syntax. For this case, a concrete syntax for the tool EMFText is provided that may be used to generate parser and editor for the respective textual constraint language. To represent the effects of configurable functionality of the software family as well as its evolution, these artifacts are subjected to both variability in space and time. Table 8.4 lists the essential characteristics of the case study.

8.3.1. Variability in Space

Table 8.3 gives information on different language constructs provided by the examined feature modeling notations and their constraint languages. The features representing variability in space of the family of feature modeling notations are aligned with the respective rows in the table. Hence, each row is discussed briefly in the following to elaborate on the features of the software family as the dimension of variability in space.

Mandatory Features represent commonalities that have to be included in a configuration, if their parent feature is selected. All examined approaches support them as a language construct.

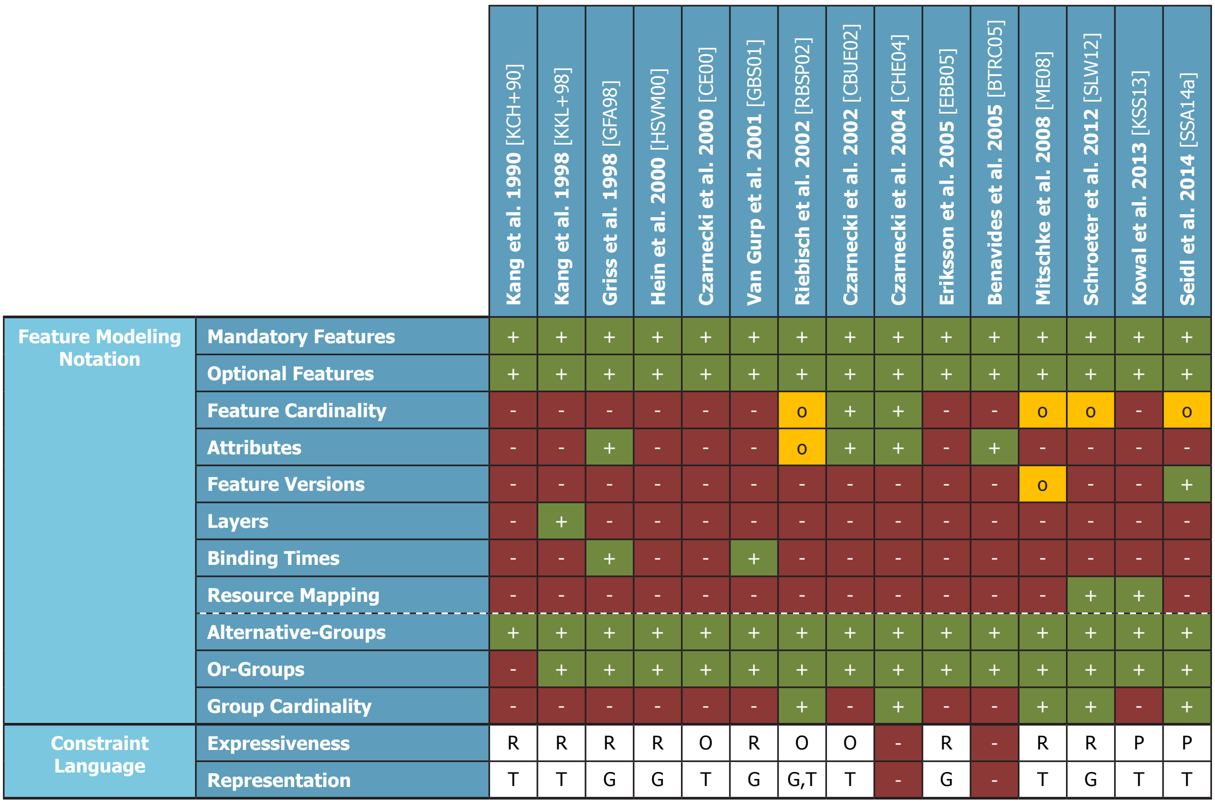
Optional Features represent variabilities that may or may not be included in a configuration. All examined approaches support them as a language construct.

Feature Cardinality specifies a minimum and maximum number for how often a feature may be selected as [m..n]. When using 1 as maximum cardinality [CBUE02, CHE04, RBSP02, ME08, SLW12, SSA14a], feature cardinality may be perceived as alternative to the explicit variation type for mandatory features (i.e., [1..1]) and optional features (i.e., [0..1]). Furthermore, Czarnecki et al. [CBUE02, CHE04] use feature cardinalities to be able to have multiple instances (e.g., [2..5]) of one and the same feature as *cloned features* by allowing maximum cardinalities greater than 1.

Attributes are named variables of features [RBSP02, CBUE02, CHE04, BTRC05] that refine configuration options so that, besides selection of features, concrete values for attributes may be chosen. Czarnecki et al. [CBUE02, CHE04] and Benavides et al. [BTRC05] assign a specific *type* or *domain* to the attributes, which specifies permissible values. In the literature, types of attributes are typically categorized into *discrete* (*finite* or *infinite*) and *continuous* domains.

Feature Versions include variability in time in feature models [SSA14a, ME08]. Seidl et al. [SSA14a, SSA14c] allow specification of multiple feature versions with interdependencies along development lines to make feature versions a configurable unit. Mitschke et al. [ME08] support two versions per feature representing the state of the feature model's structure and its associated implementation, but do not allow using them as configurable units.

Layers of feature models provide a separation of concerns for different sources of variability [KKL+98]. Kang et al. [KKL+98] use a *Capability Layer*, an *Operating Environment Layer*,



		Kang et al. 1990 [KCH+90]	**Kang et al. 1998** [KKL+98]	**Griss et al. 1998** [GFA98]	**Hein et al. 2000** [HSVM00]	**Czarnecki et al. 2000** [CE00]	**Van Gurp et al. 2001** [GBS01]	**Riebisch et al. 2002** [RBSP02]	**Czarnecki et al. 2002** [CBUE02]	**Czarnecki et al. 2004** [CHE04]	**Eriksson et al. 2005** [EBB05]	**Benavides et al. 2005** [BTRC05]	**Mitschke et al. 2008** [ME08]	**Schroeter et al. 2012** [SLW12]	**Kowal et al. 2013** [KSS13]	**Seidl et al. 2014** [SSA14a]
Feature Modeling Notation	**Mandatory Features**	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Optional Features	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Feature Cardinality	-	-	-	-	-	-	o	+	+	-	-	o	o	-	o
	Attributes	-	-	+	-	-	-	o	+	+	-	+	-	-	-	-
	Feature Versions	-	-	-	-	-	-	-	-	-	-	-	o	-	-	+
	Layers	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-
	Binding Times	-	-	+	-	-	+	-	-	-	-	-	-	-	-	-
	Resource Mapping	-	-	-	-	-	-	-	-	-	-	-	-	+	+	-
	Alternative-Groups	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Or-Groups	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Group Cardinality	-	-	-	-	-	-	+	-	+	-	-	+	+	-	+
Constraint Language	**Expressiveness**	R	R	R	R	O	R	O	O	-	R	-	R	R	P	P
	Representation	T	T	G	G	T	G	G,T	T	-	G	-	T	G	T	T

R: Requires/Excludes, O: OCL, P: Propositional Logic, T: Textual, G: Graphical

Table 8.3.: Distinctive characteristics of 15 feature modeling notations and their constraint languages used as reference for the software family.

Monitored Period	1.0 months
Features	57
Versions	64
Constraints	10

Delta Languages	2
Delta Modules	65
Configuration Delta Modules	32 (49%)
Evolution Delta Modules	33 (51%)

Table 8.4.: Essential characteristics of the case study on the SPL of feature modeling notations and constraint languages.

a *Domain Technology Layer* and an *Implementation Technique Layer*. Each layer may contain a separate feature model with relations to feature models of other layers.

Binding Times specify at which time a feature may be or has to be configured. Typical binding times are at *compile time* or *run time* [GFA98, GBS01, BLL+13]. Griss et al. [GFA98] use attributes of features to describe the binding time. Van Gurp et al. [GBS01] use a label on the connector between features to distinguish the binding time.

Resource Mapping allows for the association of arbitrary resources with the features in a feature model [SLW12, KSS13]. Schroeter et al. [SLW12] provide a mapping of features to views, which show only selective parts of a feature model for different stakeholders on the feature model. Kowal et al. [KSS13] map priorities for the configuration and specific hardware to the features.

Alternative-Groups allow for the selection of exactly one of the contained features, which makes them mutually exclusive. All examined approaches support them as a language construct.

Or-Groups allow for the selection of at least one and at most all of the contained features. With the exception of the work by Kang et al. [KCH+90], all examined approaches support them as a language construct.

Group Cardinality specifies the minimum and maximum number of selectable features in a group as [m..n]. Hence, it may be perceived as an alternative to the explicit variation type of groups as alternative-groups (i.e., [1..1]) and or-groups (i.e., [0..n] for groups with n members) [RBSP02, CHE04, SLW12, SSA14a]. In contrast to the explicit variation types, group cardinality supports further restrictions on selections in a group (e.g., [2..5]).

Expressiveness of constraint languages is determined by the employed formalism and its utilized language constructs. For one, mere *requires* and *excludes* relations (R) may be specified [KCH+90, GFA98, HSVM00, GBS01, EBB05, SLW12]. Furthermore, the OCL (O) is used in some approaches [CE00, CBUE02]. In addition, it is possible to utilize propositional logic (P). Depending on the concrete work, different selections of logical expressions are utilized to specify constraints over features [KSS13, SSA14a]. In addition, new language constructs are introduced for special purposes, e.g., to express constraints over feature versions [SSA14a].

Representation of constraints is either graphical or textual. With a graphical (G) representation, additional edges are added between two features to express requires or excludes relationships [GFA98, HSVM00, GBS01, EBB05, SLW12]. Textual representations (T) may be employed for a wider range of formalisms, such as requires and excludes relationships [KCH+90], OCL [CE00, CBUE02] or subsets of propositional logic [KSS13, SSA14a]. In the latter case, there is a further distinction on how logical operators are represented as they may use logical symbols (e.g., $\wedge$, $\vee$), a verbal representation (e.g., *and*, *or*) or a representation known from various programming languages such as Java or C++ (e.g., `&&`, `||`). In some cases [RBSP02], both a textual and a graphical representation of constraints is provided.

With these elements of variability in space, a feature model encompassing this dimension can be assembled as depicted in Figure 8.8.

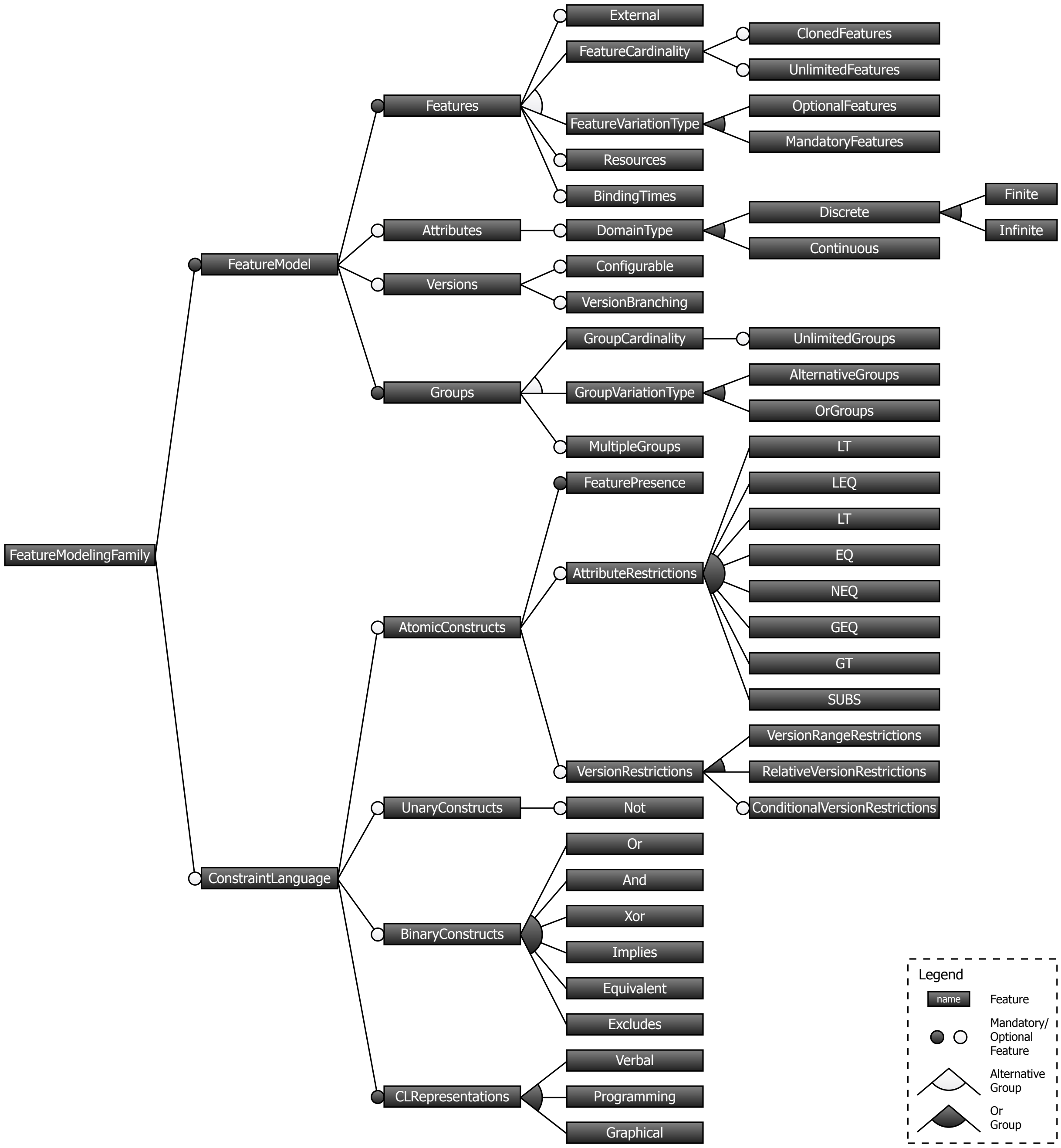


Figure 8.8.: Feature model of the software family for feature modeling notations and constraint languages.

In addition to the configuration rule specified in the feature model, cross-tree constraints further govern concerns of variability in space of the software family for feature modeling notations and constraint languages as shown in Formula 14.

Constraint (1) was introduced as the possibility to specify that a feature may be selected an unlimited number of times demands the support for cloned features. Constraint (2) was specified as the chronological relation of versions can only be captured for configurable versions. Constraints (3) and (4) were added as restrictions for attributes and versions may only be

Formula 14: Constraints of the software family for feature modeling notations and constraint languages referencing the feature model of Figure 8.8.

(1) UnlimitedFeatures → ClonedFeatures
(2) VersionBranching → Configurable
(3) AttributeRestrictions → Attributes
(4) VersionRestrictions → Configurable
(5) Graphical → $(\neg\text{Or} \land \neg\text{And} \land \neg\text{Xor} \land \neg\text{Equivalent} \land \neg\text{Not})$

specified if the respective elements are part of the notation. Constraint (5) was introduced because the graphical representation of cross-tree constraints as additional edges in the feature model is only capable of expressing implications and exclusions. In consequence, all other constructs are prohibited when this representation is selected.

8.3.2. Variability in Time

Over the course of developing the family of feature modeling notations and constraint languages, a number of changes had to be performed as part of the evolution, which represents the dimension of variability in time. These changes affect the metamodel used to define different feature modeling notations, as well as the metamodel and the accompanying concrete textual syntax for the various cross-tree constraint languages.

The **original state** of the software family realized only a fraction of the features presented in Figure 8.8 due to the incremental approach to development. In particular, the features `Resources`, `BindingTimes` and `External` as well as `Configurable` (for versions) and `VersionBranching` were not part of the original software family. Furthermore, variability in space for constraint languages and their textual representation was added as part of evolution as well. The remaining features were part of the original state of the software family but, in part, had to be revised in order to add functionality or to fix defects. In consequence, 4 revisions altered the original state of the software family.

Revision 1 of the software family introduced the `Resources` feature that allows for the association of features with various resources such as hardware, views or other arbitrary elements. Furthermore, the feature `Versions` was refined to have child features `Configurable` and `VersionBranching`. When `Configurable` is selected, feature versions are realized similar to those of [SSA14a] as units that can be selected as part of a configuration (equivalently to HFMs of this thesis). When deselected, feature versions are implemented in the sense of [ME08] as a mere value of each feature being solely informative.

Revision 2 of the software family introduced the `BindingTimes` feature that allows for stating when a feature may be configured (e.g., during compile time or run time). Furthermore, the feature `External` was added in order to support specification of references to features defined in a different feature model. Furthermore, the root feature `FeatureModelingFamily` was altered to allow not only for features but also for feature references within groups and to support an explicit variation for and-groups, where exclusively the variation type of the group's members are evaluated. The latter also has an effect on the feature `GroupVariationType`, which was altered to provide basic handling of the and-group variation type. Finally, a bug in the feature `Attributes` was fixed, where an attribute value would have the wrong type.

Revision 3 of the software family fixed a bug in the feature `BindingTimes`, where the enumeration literal for “run time” as one the standard binding times would have a wrong value. Furthermore, some minor problems in the realization of the feature `Resources` were addressed that appeared when referencing externally defined resources, e.g., from hardware models. Finally, two bugs with discrete attribute values were fixed, where integer values and references to custom-defined enumeration literals could not be represented properly in certain cases.

Revision 4 introduced variability for the constraint language and its textual syntax. Hence, feature `ConstraintLanguage` and all its descendant features were created along with their respective initial versions.

With the original state and the following 4 revisions, the feature model of Figure 8.8 was extended to the HFM depicted in Figure 8.9 and Figure 8.10.

In addition to the constraints of Formula 14, further version-aware constraints had to be defined to represent the dimension of variability in time as depicted in Formula 15.

Formula 15: Additional version-aware constraints of the software family for feature modeling notations and constraint languages referencing the HFM of Figure 8.8.

(6) $\neg\text{Attributes}\ [=0]$
(7) $\neg\text{Resources}\ [=0]$
(8) $\neg\text{Attributes}\ [=2] \wedge \neg\text{Discrete}\ [\geq 0]$
(9) $\neg\text{BindingTimes}\ [=2]$
(10) $\text{External} \rightarrow \text{FeatureModelingFamily}\ [\geq 2]$

Constraints (6) to (9) each were defined to deprecate (combinations of) feature versions that should no longer be used for configuration as the respective realizations contained critical bugs. Constraint (10) was added, as external features depend on the modification to the root feature of the software family performed in revision 2.

8.3.3. Integrated Management of Variability in Space and Time

The base variants for the feature model and the constraint language of the software family largely resemble the metamodels for HFMs and their version-aware constraint language as described within this thesis. These base variants provide language constructs as described by the last column of Table 8.3. Figure 8.11 depicts the base variant of the metamodel for feature models. Furthermore, Figure 8.12 depicts the base variant of the metamodel for constraint languages in propositional logic and Listing 8.9 shows the base variant for the concrete textual syntax for EMFText.

To manifest the changes associated with variability in space and time within the realization assets of the software family, suitable delta languages had to be devised for EMF Ecore and EMFText concrete syntax files. A delta language for Ecore called “DeltaEcore” (not to be confused with the framework of Chapter 7) had already been created as part of the case study described in Section 8.2. Hence, only minor additions to the language were necessary to make it suitable for this use case, e.g., adding a delta operation for making a metaclass an interface. The respective delta dialect can be found in Appendix B.1.

However, the delta language for EMFText concrete syntax files had to be created. As EMFText concrete syntax files are themselves defined using EMF Ecore, the respective delta language could directly be defined using the tool suite DeltaEcore as presented in Chapter 7. The delta dialect features merely 2 configuration delta operations to add new or remove existing syntax

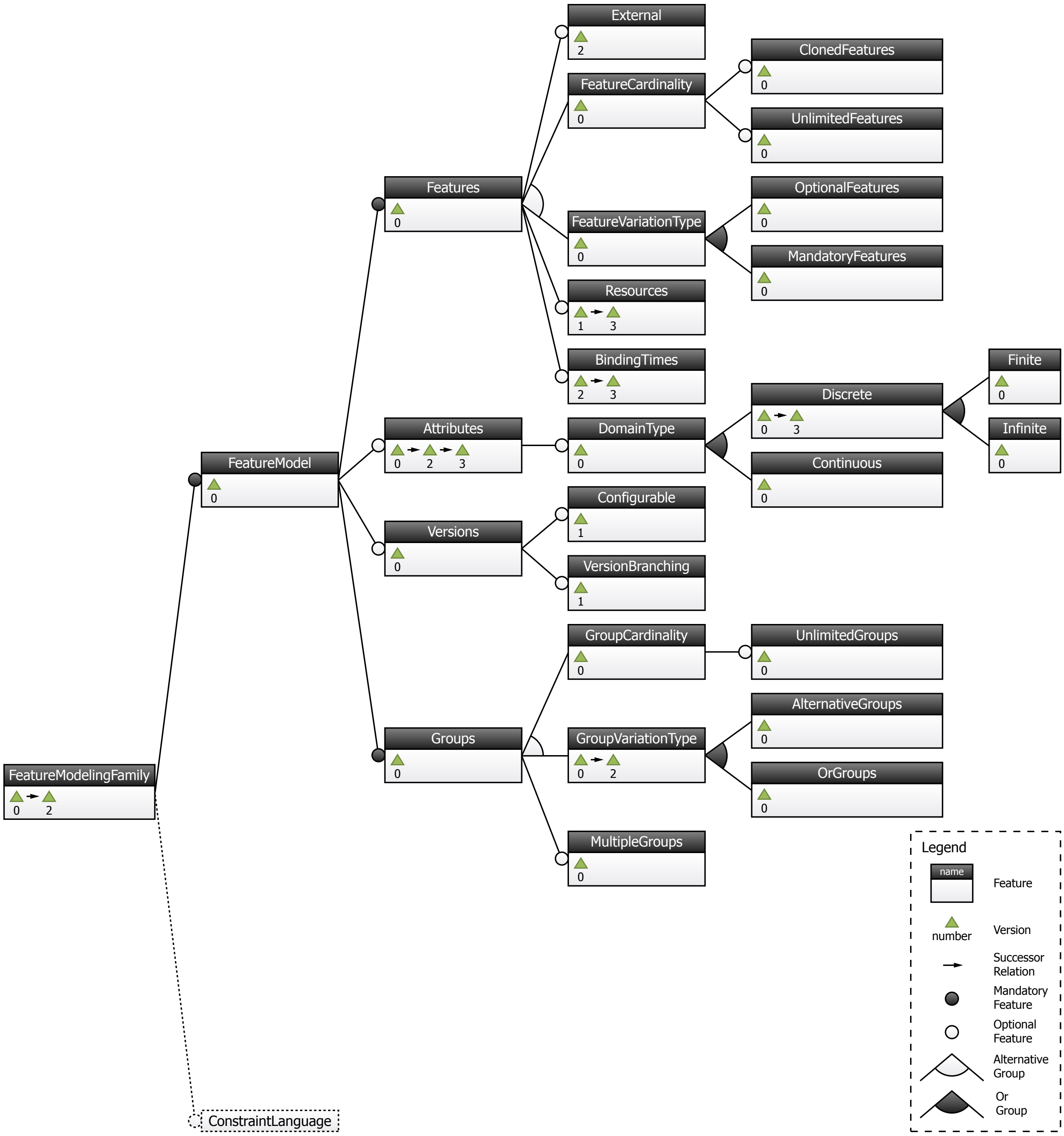


Figure 8.9.: HFM of the software family for feature modeling notations and constraint languages (1/2).

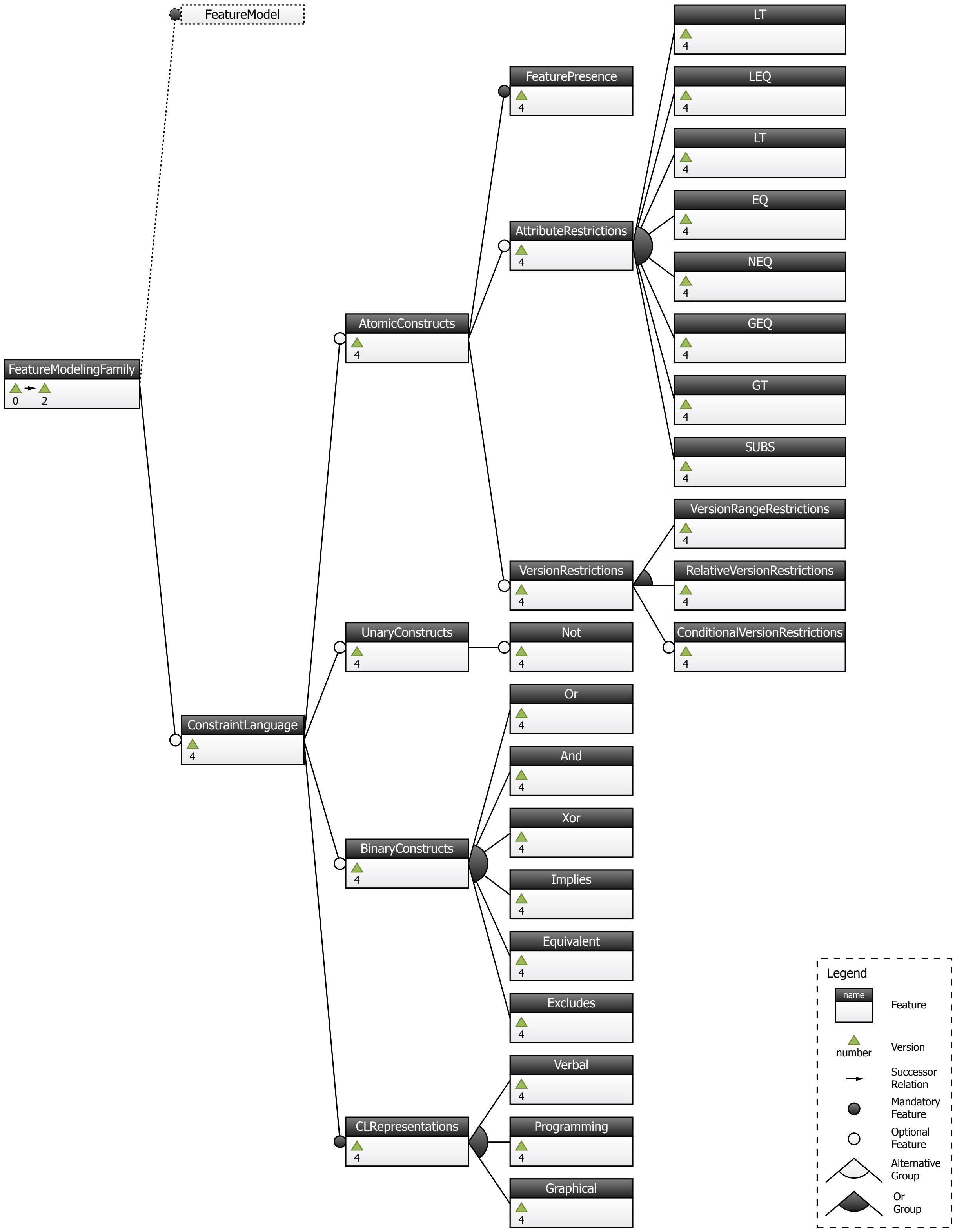


Figure 8.10.: HFM of the software family for feature modeling notations and constraint languages (2/2).

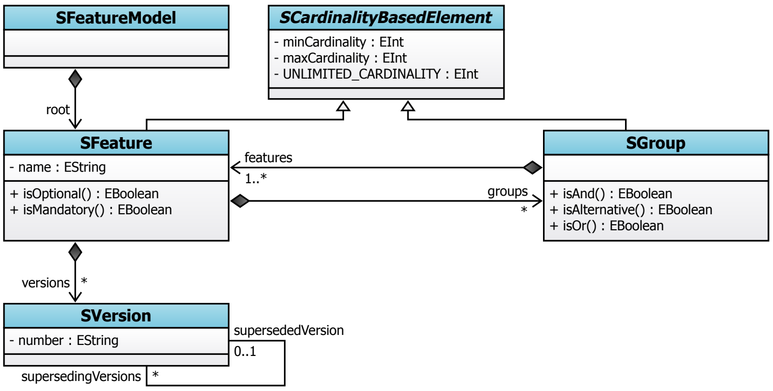


Figure 8.11.: Base variant of the metamodel for the feature modeling notation.

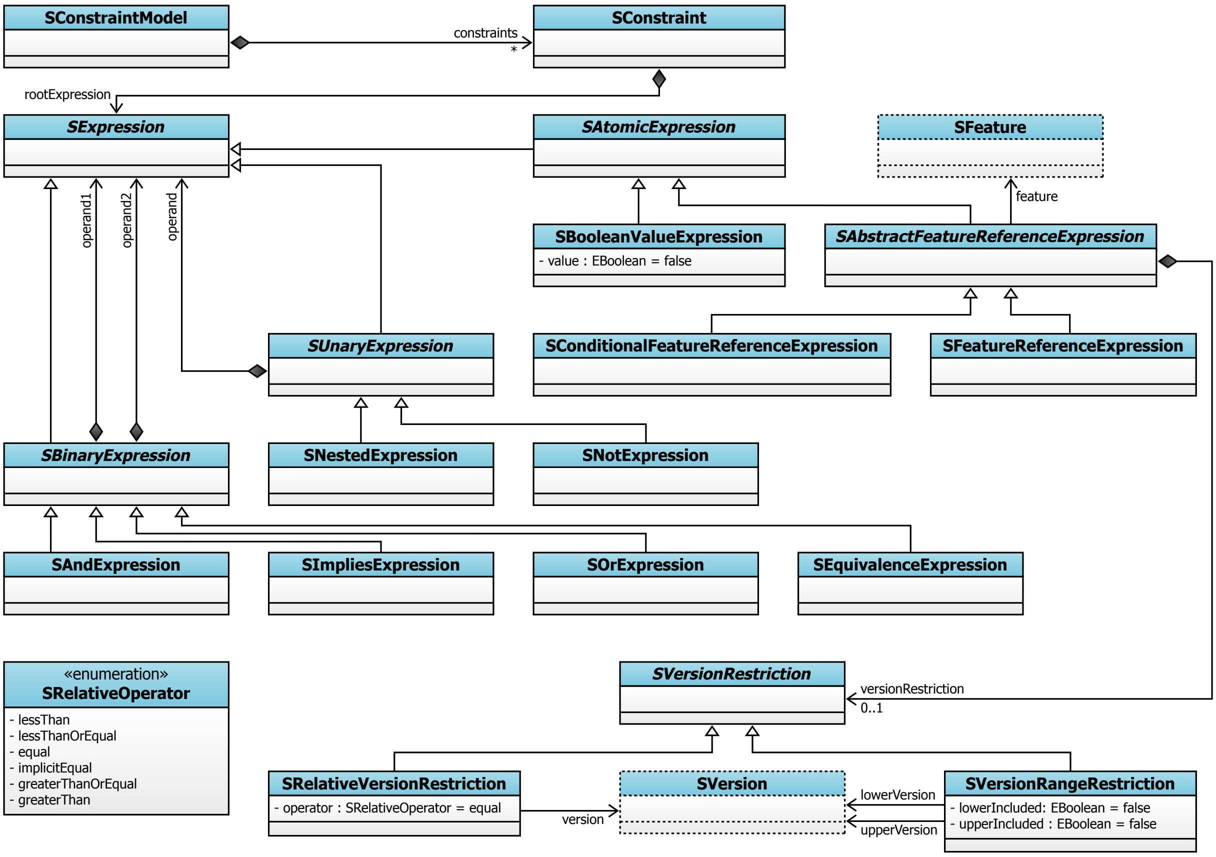


Figure 8.12.: Base variant of the metamodel for the constraint language. The metaclasses SFeature and SVersion are defined in Figure 8.11.

rules. For the prior, a literal representation of the syntax rule as string is parsed on execution of the delta operation and added at the appropriate location of the concrete syntax file as model element. The respective delta dialect can be found in Appendix B.9.

Using these delta languages, a total of 65 delta modules was specified to manifest variability in the realization assets of the software family. Of these delta modules, 32 were configuration delta modules realizing changes to accommodate for the different language constructs of the

```
SYNTAXDEF constraints
FOR <http://www.tu-bs.de/snowflake/constraint/1.0> <Constraint.genmodel>
START SConstraintModel

TOKENS {
  DEFINE INTEGER_TOKEN $('0'..'9')+$;
  DEFINE DOUBLE_TOKEN $('0'..'9')+('.')('0'..'9')+$;
  DEFINE IDENTIFIER_TOKEN $('A'..'Z'|'a'..'z'|'_')('A'..'Z'|'a'..'z'|'_'|'0'..'9')*$;
  DEFINE QUALIFIED_IDENTIFIER_TOKEN IDENTIFIER_TOKEN + $('.')$ + IDENTIFIER_TOKEN;

  DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\uffff'))*$;
  DEFINE ML_COMMENT $'/*'.*'*/'$;

  DEFINE LINEBREAK $('\r\n'|'\r'|'\n')$;
  DEFINE WHITESPACE $(' '|'\t'|'\f')$;
}

RULES {
  SConstraintModel ::= "constraints" "for" featureModel['<','>'] constraints*;

  SConstraint ::= rootExpression;

  @Operator(type="binary_left_associative", weight="2", superclass="SExpression")
  SImpliesExpression ::= operand1 "->" operand2;

  @Operator(type="binary_left_associative", weight="4", superclass="SExpression")
  SOrExpression ::= operand1 "||" operand2;

  @Operator(type="binary_left_associative", weight="6", superclass="SExpression")
  SAndExpression ::= operand1 "&&" operand2;

  @Operator(type="unary_prefix", weight="7", superclass="SExpression")
  SNotExpression ::= "!" operand;

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SNestedExpression ::= "(" operand ")";

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SFeatureReferenceExpression ::= (feature['"', '"'] | feature[]);

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SRelativeVersionRestriction ::= conditional["?" : ""] (feature['"', '"'] | feature[])
    "[" operator[lessThan : "<", lessThanOrEqual : "<=", equal : "=", implicitEqual : "",
    greaterThanOrEqual : ">=", greaterThan : ">"] version['"','"'] "]";

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SVersionRangeRestriction ::= conditional["?" : ""] (feature['"', '"'] | feature[])
    "[" lowerIncluded["" : "^"] lowerVersion['"','"'] "-"
    upperIncluded["" : "^"] upperVersion['"','"'] "]";
}
```

Listing 8.9: Concrete syntax for the metamodel of the constraint language defined in Figure 8.12 as specified with EMFText.

feature modeling notations and constraint languages. For example, Listing 8.10 depicts the configuration delta module enabling the feature `Attributes`, which allows for using attributes in feature models and assigning values to the respective attributes.

Furthermore, there were 33 evolution delta modules coping with variability in time in order to add new functionality and fix defects in the realization of existing features. For example, Listing 8.11 depicts an evolution delta module that performs an update of the software family's root feature as part of revision 2. The performed changes enable other arti-

```
configuration delta "Attributes"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/Feature.ecore>
{
  EClass attributeEClass = new EClass(name: "SAttribute");
  addEClass(attributeEClass, <feature>);

  EAttribute attributeEAttribute = new EAttribute(name : "name",
    eType: <EString>, lowerBound : 1, upperBound : 1);
  addEAttribute(attributeEAttribute, attributeEClass);

  EClass attributeValueEClass = new EClass(name: "SAttributeValue");
  setEClassAbstract(true, attributeValueEClass);
  addEClass(attributeValueEClass, <feature>);

  EClass attributeValueAssignmentEClass = new EClass(name: "SAttributeValueAssignment");
  addESuperType(<SConfigurationArtifact>, attributeValueAssignmentEClass);
  addEClass(attributeValueAssignmentEClass, <feature>);

  EReference attributeEReference = new EReference(eType : attributeEClass,
    name : "attribute", lowerBound : 1, upperBound : 1, containment : false);
  addEReference(attributeEReference, attributeValueAssignmentEClass);

  EReference attributeValueEReference = new EReference(eType : attributeEClass,
    name : "attributeValue", lowerBound : 1, upperBound : 1, containment : true);
  addEReference(attributeValueEReference, attributeValueAssignmentEClass);

  //attributes: SFeature <>->* SAttribute
  EReference attributesEReference = new EReference(eType : attributeEClass,
    name : "attributes", lowerBound : 1, upperBound : -1, containment : true);
  addEReference(attributesEReference, <SFeature>);
}
```

Listing 8.10: Example configuration delta module that enables the feature `Attributes`.

facts than features to be members of groups (e.g., references to external features) and install the option of using and-groups explicitly.

The specified delta modules were assigned to (combinations of) features and feature versions of Figure 8.9 and Figure 8.10. In consequence, configurations defined on the HFM that are valid with regard to the configuration knowledge may be resolved to a set of required delta modules. When creating and evaluating the graph of delta modules with the “apply after” arcs from the HFM (see Section 6.4), it is possible to devise a suitable order for application of the delta modules. Executing all required delta modules according to this order, by executing the delta operations of each delta module sequentially, yields a variant of the software family that contains both variability in space and time. As example, artifacts of a variant are depicted in Figure 8.13, Figure 8.14 and Listing 8.12 realizing the configuration of Formula 16.

To demonstrate that the specified software family subsumes the individual approaches analyzed in Table 8.3, configurations were specified according to the entries in each column of the table to reflect the respective language constructs. These configurations were used to derive variants of the metamodels for the feature modeling notation and its constraint language as well as the concrete syntax file of the textual constraint language. The generated variants were inspected for conformance with the selected configurations as well as for their expressiveness with regard

```
evolution delta "FeatureModelingFamily_2"

dialect <http://www.eclipse.org/emf/2002/Ecore>
  requires <../model/Feature.ecore>
{
  //Extract group artifacts

  //Remove features : SGroup <>—>* SFeature
  EReference featuresEReference = <SGroup.features>;
  removeEReference(featuresEReference);

  //Create SGroupArtifact interface
  EClass groupArtifactEClass = new EClass(name: "SGroupArtifact");
  setEClassInterface(true, groupArtifactEClass);
  setEClassAbstract(true, groupArtifactEClass);
  addEClass(groupArtifactEClass, <feature>);

  //Make SGroupArtifact super class of SFeature
  addESuperType(groupArtifactEClass, <SFeature>);

  //Add groupArtifacts: SGroup <>—>* SGroupArtifact
  EReference groupArtifactsReference = new EReference(eType : groupArtifactEClass,
    name : "groupArtifacts", lowerBound : 0, upperBound : -1, containment : true);
  addEReference(groupArtifactsReference, <SGroup>);

  //Change default implementation of SGroup.isOr()
  String implementation = "return (getMinCardinality() == 1 &&
    getMaxCardinality() == getArtifacts().size());";
  setEOperationImplementation(<SGroup.isOr>, implementation);


  //Add And Groups

  //Add SGroup.isAnd()
  EOperation isAndEOperation = new EOperation(eType : <EBoolean>,
    name : "isAnd", lowerBound : 1, upperBound : 1);
  addEOperation(isAndEOperation, <SGroup>);

  String implementation2 = "int optionalFeatures = 0;
    int mandatoryFeatures = 0;

    for (SGroupArtifact artifact : getArtifacts()) {
      if (artifact instanceof SFeature) {
        SFeature feature = (SFeature) artifact;
        if (feature.isOptional()) {
          optionalFeatures++;
        } else if (feature.isMandatory()) {
          mandatoryFeatures++;
        }
      }
    }

    return (getMinCardinality() <= mandatoryFeatures &&
      getMaxCardinality() >= (mandatoryFeatures + optionalFeatures));";
  setEOperationImplementation(isAndEOperation, implementation2);
}
```

Listing 8.11: Example evolution delta module that updates the root feature `FeatureModelingFamily` as part of revision 2.

Formula 16: Formalization of an example configuration for the software family of feature modeling notations and constraint languages.

$$\mathcal{C} = \{\text{FeatureModelingFamily}, 0\ (\text{FeatureModelingFamily}), \text{FeatureModel}, 0\ (\text{FeatureModel}), \text{Features}, 0\ (\text{Features}), \text{FeatureCardinality}, 0\ (\text{FeatureCardinality}), \text{Attributes}, 3\ (\text{Attributes}), \text{DomainType}, 0\ (\text{DomainType}), \text{Discrete}, 0\ (\text{Discrete}), \text{Finite}, 0\ (\text{Finite}), \text{Infinite}, 0\ (\text{Infinite}), \text{Groups}, 0\ (\text{Groups}), \text{GroupVariationType}, 2\ (\text{GroupVariationType}), \text{AlternativeGroups}, 0\ (\text{AlternativeGroups}), \text{OrGroups}, 0\ (\text{OrGroups}), \text{MultipleGroups}, 0\ (\text{MultipleGroups}), \text{ConstraintLanguage}, 4\ (\text{ConstraintLanguage}), \text{UnaryConstructs}, 4\ (\text{UnaryConstructs}), \text{Not}, 4\ (\text{Not}), \text{BinaryConstructs}, 4\ (\text{BinaryConstructs}), \text{Or}, 4\ (\text{Or}), \text{And}, 4\ (\text{And}), \text{Xor}, 4\ (\text{Xor}), \text{Implies}, 4\ (\text{Implies}), \text{CLRepresentations}, 4\ (\text{CLRepresentations}), \text{Verbal}, 4\ (\text{Verbal})\}$$

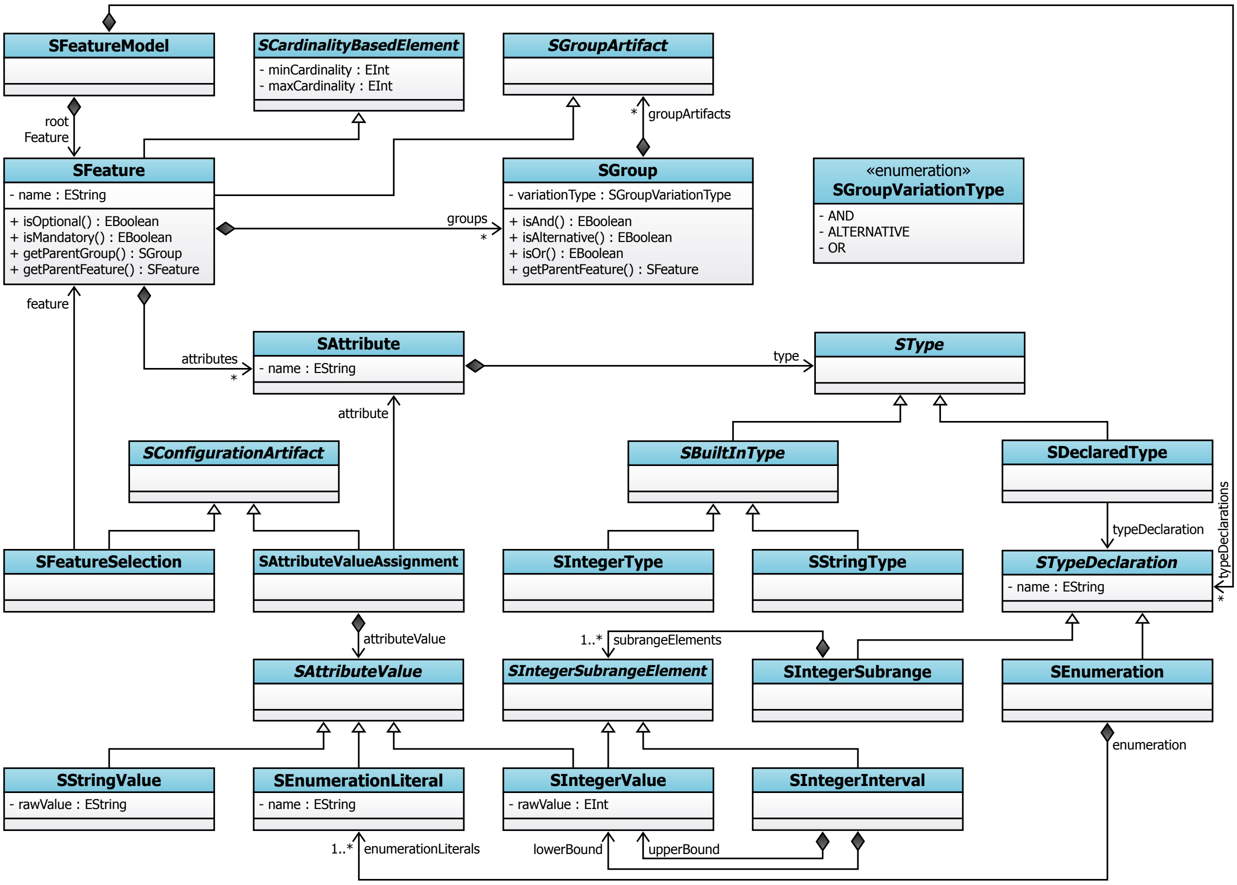


Figure 8.13.: Variant of the metamodel for the feature modeling notation that realizes the configuration of Formula 16.

to the included distinctive characteristics. The variants of the metamodel and the constraint language were further used to recreate examples presented in each of the analyzed works[2].

As these examples were created during the ongoing development of the software family, some of the created models depend on the structure of their respective metamodel as created with a

[2] For papers, all presented examples were recreated. However, [CE00] is a book with over 800 pages so that a representative sample of all presented feature models and constraints was created.

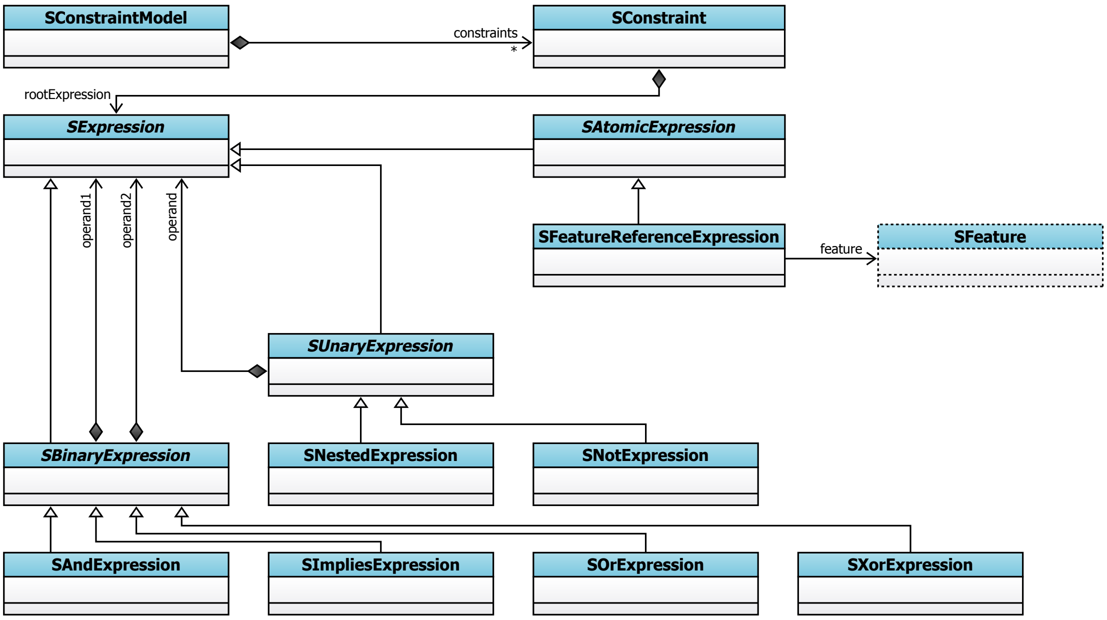


Figure 8.14.: Variant of the metamodel for the constraint language that realizes the configuration of Formula 16. The metaclass `SFeature` is defined in Figure 8.13.

feature in a particular version. To further support these artifacts, variants were derived that contained combinations of features and feature versions that had not been anticipated initially. Through this mechanism, it was possible to import models conforming to older versions of certain features of the software family, exporting them to a (largely) version-agnostic textual intermediate format and finally re-importing them into a variant with newer versions of the feature in question. Through this procedure, it was possible to effectively upgrade artifacts specified with an older version of a feature without breaking compatibility.

8.4. Results and Discussion

The case studies of Section 8.1, Section 8.2 and Section 8.3 each describe a software family encompassing variability in space and time. The TurtleBot driver software is maintained by a core development team and 4 loosely connected extension developers performing largely independent development. The metamodel family for role-based modeling is maintained by 4 different developers who closely coordinate their efforts. The SPL for feature modeling notations is maintained by 2 developers working on different areas of the realization who have to coordinate their work when altering parts of the realization that affect the other one. The development of the software families within the respective case studies was monitored for approximately 1.5 years, 1.5 months and 1 month, respectively. To evaluate the concepts of the thesis, these case studies applied the approach for integrated management of variability in space and time. The following sections present the results of the case studies with regard to the research questions posed in the introduction of Chapter 8.

```
SYNTAXDEF constraints
FOR <http://www.tu-bs.de/snowflake/constraint/1.0> <Constraint.genmodel>
START SConstraintModel

TOKENS {
  DEFINE INTEGER_TOKEN $(’0’..’9’)+$;
  DEFINE DOUBLE_TOKEN $(’0’..’9’)+(’.’)(’0’..’9’)+$;
  DEFINE IDENTIFIER_TOKEN $(’A’..’Z’|’a’..’z’|’_’)(’A’..’Z’|’a’..’z’|’_’|’0’..’9’)*$;
  DEFINE QUALIFIED_IDENTIFIER_TOKEN IDENTIFIER_TOKEN + $(’.’)$ + IDENTIFIER_TOKEN;

  DEFINE SL_COMMENT $’//’(~(’\n’|’\r’|’\uffff’))*$;
  DEFINE ML_COMMENT $’/*’.*’*/’$;

  DEFINE LINEBREAK $(’\r\n’|’\r’|’\n’)$;
  DEFINE WHITESPACE $(’ ’|’\t’|’\f’)$;
}

RULES {
  SConstraintModel ::= "constraints" "for" featureModel[’<’,’>’] constraints*;

  SConstraint ::= rootExpression;

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SNestedExpression ::= "(" operand ")";

  @Operator(type="primitive", weight="8", superclass="SExpression")
  SFeatureReferenceExpression ::= (feature[’"’,’"’] | feature[]);

  @Operator(type="binary_left_associative", weight="5", superclass="SExpression")
  SXorExpression ::= operand1 "xor" operand2;

  @Operator(type="binary_left_associative", weight="4", superclass="SExpression")
  SOrExpression ::= operand1 "or" operand2;

  @Operator(type="binary_left_associative", weight="6", superclass="SExpression")
  SAndExpression ::= operand1 "and" operand2;

  @Operator(type="unary_prefix", weight="7", superclass="SExpression")
  SNotExpression ::= "not" operand;

  @Operator(type="binary_left_associative", weight="2", superclass="SExpression")
  SImpliesExpression ::= operand1 ("implies" | "requires") operand2;
}
```

Listing 8.12: Variant of the concrete syntax for the metamodel of the constraint language variant defined in Figure 8.14 as specified with EMFText that realizes the configuration of Formula 16.

8.4.1. Results and Discussion of RQ1: Variability Model

On a conceptual level, within the case studies, variability in space was represented by multiple features. Furthermore, incremental changes performed on the functionality of a feature were represented as feature versions. For this purpose, HFMs were employed along with version-aware constraints. In total, the case studies defined 116 features and 199 versions. For the individual case studies, this amounts to 11, 48 and 57 features as well as 29, 106 and 64 versions, respectively. On average, the case studies declared 2.63, 2.21 and 1.12 versions per feature. The low number of the latter is due to the relatively short development duration captured within the respective case study of merely one month. Table 8.5 summarizes the essential characteristics of the case studies with regard to conceptual modeling of variability in space and time within a variability model.

	Monitored Period	Features	Versions	Versions/Feature	Constraints
TurtleBot Driver	1.5 years	11	29	29/11 ≈ 2.63	6
Metamodel Family for Role Modeling	1.5 months	48	106	106/48 ≈ 2.21	5
SPL for Feature Modeling Notations	1.0 months	57	64	64/57 ≈ 1.12	10

Table 8.5.: Essential characteristics of the variability models of the case studies in Chapter 8.

For all encountered cases, it was possible to attribute the changes to one particular feature or a combination of features so that it was clear which version(s) to increment. In consequence, the ordering of features along development lines could be utilized to express this relation by marking one particular version as predecessor that is superseded by the newly created version. In the case of the TurtleBot driver, it was even possible to capture the branching of both the `TurtleBot` and `Engine` features with introduction of the new hardware revision. Names assigned to the versions of the HFM as representatives of the versions' "numbers" could be chosen freely and consisted of integer numbers (e.g., "1" in the feature modeling SPL), major/minor schemes (e.g., "1.2" in the TurtleBot driver), dates (e.g., "2014_07_14" in the family of metamodels for role modeling) and symbolical names (e.g., "Kobuki 1.0" in the TurtleBot driver).

Furthermore, respectively 6, 5 and 10 constraints were declared for the case studies using the version-aware constraint language. Multiple means existed for further constraining the configuration knowledge through constraints: For one, the dimension of variability in space could be handled by constraints containing references solely to features (e.g., RoleImplication → RoleEquivalence from the metamodel family for role modeling). Furthermore, concerns of variability in time could be addressed: It was possible to model dependencies on version ranges (e.g. "External → FeatureModelingFamily [≥ 2]" from the SPL for feature modeling notations). In addition, complex dependencies could be expressed that used both features and ranges of feature versions (e.g., "Infrared [≥ 2.0] ∨ Ultrasound → Detection [≥ 1.1]" from the TurtleBot driver). Conditional version restrictions were used to model that the dependency on a version range only has to be satisfied *if* the respective feature is part of a configuration, which avoided unintentionally making a feature mandatory through a version-aware constraint (e.g., "TurtleBot [≥ 2.0] → ?Webservice [≥ 1.1]" from the TurtleBot driver). Furthermore, versions that should no longer be selected as part of a configuration could effectively be deprecated by version-aware constraints (e.g., "¬Attributes [= 0]" from the SPL for feature modeling notations). Finally, entire baselines of a software family could be established using version-aware constraints (see below).

However, some caveats were encountered when using HFMs and version-aware constraints to capture variability in space and time on a conceptual level. Within the SPL for feature modeling notations, one incremental change consisted of fixing a defect that was introduced by a delta module that was mapped to the complex expression "(Attributes ∧ ¬DomainType) ∨ Discrete". Due to this expression, it was complicated to determine a definite predecessor version and, hence, on which development line(s) to create new version(s) by merely inspecting the structural properties of the HFM and mapping model. However, utilizing the domain knowledge of the software family, it was possible to identify the changes as being related to the features `Attributes` and `DomainType`. In consequence, new versions with the number "3" were created for both these features and the delta module to fix the defect was mapped using the expression "(Attributes [≥ 3] ∧ ¬DomainType [≥ 3]) ∨ Discrete"

Furthermore, in the metamodel family of role modeling notations, the misspelled feature name "RoleInheritence" was changed to "RoleInheritance" as part of evolution. The general case of altering feature names potentially invalidates previously valid configurations and, thus, was declared as being out of scope of the thesis (see Section 3.4.3). However, for the concrete case within the case study, the consequences of the change in feature name were predicted as being negligible so that the feature name was altered manually. While performing the change programmatically would pose no significant technical problems, it might cause harm on a conceptual level due to its potential effects on previously existing configurations. Hence, further inspection is required to integrate modifications on HFMs as part of evolution into the approach of this thesis. Section 9.2 presents initial ideas for further extensions in this regard.

In addition, the metamodel family of role modeling notations performed extensive changes to the realization in the *2014-07-14* evolution stage by altering the base package of all metamodel elements. In consequence, the hierarchical identifiers of *all* metamodel elements changed with these modifications, which caused incompatibilities when attempting to reference elements created before and after the respective versions. To avoid these conflicts, a baseline was established that ensured that only versions are selected that either were established before or after the *2014-07-14* evolution stage but no combinations of versions from both groups. For this purpose, the version-aware constraint of Formula 13 (5) was defined. Due to the large number of affected features, the constraint is verbose and would even be increased with the number of features defined in the HFM. Hence, if a baseline constraint should be identified as a necessary construct in modeling variability in time on a conceptual level, it may be worth investigating how to establish a dedicated language construct for baselines as first class entity in the version-aware constraint language. However, the mentioned occurrence of the need for a baseline was the only one in all three case studies and it could be handled with the provided constructs of the version-aware constraint language.

Concluding on the inspection of `RQ1`, HFMs could be used to define both features and versions for all cases encountered within the case studies. Furthermore, version-aware constraints could be utilized to express dependencies on other versions even for complex combinations of features and versions as well as to deprecate old versions. The problems encountered on a conceptual level could be solved with the means provided by HFMs and the version-aware constraint language. Hence, `RQ1` can be answered positively, as it was possible to conceptually model variability in space and time using HFMs and the version-aware constraint language.

8.4.2. Results and Discussion of RQ2: Variability Realization Mechanism

The challenges of `RQ2` are twofold in first creating and then applying suitable delta languages. The following sections discuss how each of these challenges was addressed within the case studies.

8.4.2.1. Results and Discussion of Delta Language Creation

Within the case studies, a total of 9 different source languages was employed: Java, Eclipse projects, DocBook markup, Software Fault Trees (SFTs), Component Fault Diagrams (CFDs), Checklists (CLs), the Goal Structuring Notation (GSN), EMF Ecore and the language for EMFText concrete syntax files. These source languages have vastly different characteristics, such as intended use, complexity or representation: Intended use of the languages includes programming in the wider sense (i.e., Java, Eclipse projects), modeling (i.e., EMF Ecore, EMFText concrete syntax files), documentation (i.e., DocBook) and safety certification (i.e., SFT, CFD, CL, GSN). Complexity of languages ranges from very low to high, i.e., from CL with merely 3 language constructs to Java with 114 rules in the language's grammar[3]. The

[3] Within the case study, Java 5 was employed. Its grammar specification is available at `http://docs.oracle.com/javase/specs/jls/se5.0/html/syntax.html`.

representation on the languages may be textual (e.g., Java, DocBook), graphical (e.g., CFD, GSN) or even both (e.g., SFTs were provided with both a textual and a graphical representation). Furthermore, artifacts of some of the languages may directly by connected to those of other languages. For example, the GSN builds a semi-formal line of argumentation for a system's safety, i.a., by referencing SFTs, CFDs and CLs as evidence. Hence, the selection of source languages within the case studies covers a wide area of potential notations. However, for all these languages, a metamodel based on EMF Ecore was provided to unify their further processing.

As the artifacts of these languages were subjected to variability in space and time within the case studies, suitable delta languages had to be devised. For this purpose, delta dialects for the respective metamodels were created using the tool suite DeltaEcore presented in Chapter 7. The basic version for the delta dialects was generated by applying the analysis techniques of Section 5.4. The results were fully functional delta dialects that may be used to retrieve delta languages capable of altering artifacts of the source languages in the course of variability. Elements of the metamodel marked as identifiers were recognized by the procedure so that the respective delta operations were created exclusively for the purposes of evolution. However, some of the source languages use compound identifiers, such as qualified names of Java, consisting of a concatenation of element names. Due to this reason, the metamodels did not mark the respective elements as identifiers and the automatic delta language generation could not recognize the respective operations as evolution delta operations. Furthermore, for more complex languages, such as Java or Ecore, the fine-grained level of the delta operations' semantics and the presence of opposite references within the metamodel resulted in a relatively large number of delta operations. In addition, more complex operations utilizing knowledge of the source language's intended use (e.g., by maintaining well-formedness) could not be generated automatically.

Due to these reasons, the basic delta dialects were altered and refined manually. Excess delta operations that were automatically generated could be removed or commented out within the delta dialects, in order to disable them. The freedom of choice regarding names in general and regarding the number and types of parameters as well as semantics of custom delta operations allowed for realizing all intended delta operations. The semantics of custom delta operations were realized by providing a Java implementation within the generated structure for a delta dialect interpreter. Within this implementation, it was possible to reuse the generated interpretation methods of standard delta operations to perform parts of the modifications of custom delta operations, e.g., by using the standard delta operation to add an element after it was created from a set of user specified parameters within a custom delta operation. Table 8.6 summarizes the characteristics of the delta dialects created within the case studies.

The delta dialects vary in terms of complexity by specifying between 2 and 70 delta operations. The majority of these delta operations are configuration delta operations, which may be used to realize changes associated with variability in space and in time. However, with the exception of Eclipse projects and EMFText concrete syntax files, each delta dialect defines at least one evolution operation, which may be used exclusively for changes associated with variability in time. The ratio of standard to custom delta operations also varies with the delta dialect ranging from 0% standard operations for Eclipse project files over 50% standard operations for Java and DocBook to 100% standard delta operations for CLs.

The average number of LOC required to realize a custom delta operation's implementation varies between 3 and 14.5 LOC. This relatively low number is possible due to two reasons: First, the automatically generated part of the delta dialect interpreter performs the vast majority of conversion between the EMF Ecore-based delta operation that is specified in the delta dialect and the Java method of the concrete interpreter that realizes its semantics. Second, through

		Configuration vs. Evolution		Standard vs. Custom			
	Operations	Configuration Operations	Evolution Operations	Standard Operations	Custom Operations	LOC/Custom Operation	Identifiers
Java	70	66 (94%)	4 (6%)	35 (50%)	35 (50%)	168/35 = 4.8	hierarchical
Projects	4	4 (100%)	0 (0%)	0 (0%)	4 (100%)	20/4 = 5.0	-
DocBook	10	9 (90%)	1 (10%)	5 (50%)	5 (50%)	15/5 = 3.0	direct
SFT	20	16 (80%)	4 (20%)	19 (95%)	1 (5%)	11/1 = 11.0	direct
CFD	31	25 (81%)	6 (19%)	29 (94%)	2 (6%)	14/2 = 7.0	mixed
CL	11	10 (91%)	1 (9%)	11 (100%)	0 (0%)	-	direct
GSN	9	8 (89%)	1 (11%)	7 (78%)	2 (22%)	29/2 = 14.5	direct
Ecore	29	23 (79%)	6 (21%)	26 (90%)	3 (10%)	24/3 = 8.0	hierarchical
CS	2	2 (100%)	0 (0%)	1 (50%)	1 (50%)	9/1 = 9.0	direct

Table 8.6.: Essential characteristics of the delta dialects created for the source languages of the case studies in Chapter 8.

the generative approaches of EMF Ecore and EMFText, a large number of utility methods is readily available for altering artifacts of the source languages, so that adding elements or parsing a textual representation into a partial model may be accomplished with a single LOC.

Furthermore, the delta dialects used various different types of identifiers that resembled customs of the respective source languages. For example, SFTs use direct identifiers as each element carries a unique id that may be specified to retrieve it. In contrast, Java uses qualified names, which are compounds of the names of elements, to navigate a conceptually hierarchical structure, e.g., `eu.vicci.turtlebot.Engine` to navigate through the package hierarchy to the `Engine` class. In addition, CFDs use a mixture of both these approaches as components carry a unique id (e.g., `BS` for the braking system component), which may be addressed directly, but their ports are retrieved by further navigating the hierarchical structure of the component (e.g., `BS:BrakingFails` for the respective out-port of the braking system). Support for direct identifiers was generated automatically for a delta dialect when the respective elements of the metamodel were marked as identifiers. For all other types of identifiers, it was possible to provide an identifier resolver along with the delta dialect that resolves identifiers given as strings to their respective element within an artifact of the source language (see Section 7.1.3). DeltaEcore provided support for the basic structure of parsing hierarchical identifiers that are related to the containment hierarchy specified in the metamodel such as qualified names. The delta dialects for both Java and EMF Ecore utilize this structure to support hierarchical identifiers.

However, the case studies also revealed some caveats: For the Java programming language, the metamodel named many metamodels equivalently to the language constructs they represent within the Java language. Hence, it was likely to use derived names for the elements within the according delta dialects, e.g., for a delta operation to add packages, a parameter of type `Package` with the name `package`. Even though the delta dialects support this naming, the Java code generated to interpret the respective delta operation was invalid, as it attempts to use the identifier `package` as a parameter name, which is a reserved keyword in Java. Similar problems arose for other elements of the metamodel and their respective derived names, e.g., `class`, `abstract` etc. However, the problems could be circumvented by using other names (e.g., `p` instead of `package`) for the elements of the delta dialect. The mandatory elements of the delta dialect, such as type references, were not affected by similar problems, such as type shadowing.

In addition, some operations that were to be realized had the general characteristics of a standard delta operation, but could not be matched directly to elements of the (externally provided) metamodel. For example, the delta dialect for DocBook defines a delta operation `addParagraph` that resembles an add standard delta operation. However, the container element for paragraphs is a marker interface that does not provide direct access to a reference holding the paragraphs. Similarly, the delta dialect for Java provides a delta operation to toggle the `abstract` modifier of a class, which, conceptually, is a Boolean state. However, the metamodel realizes modifiers using a list of instances of marker classes, so that there is no respective Boolean attribute for the abstract modifier. In these cases, the structure of the respective standard delta operation could not be employed and a custom delta operation had to be used instead.

Furthermore, the delta dialects for Java and Ecore would have benefited from the chance for overloading of delta operations: In some cases, multiple delta operations perform equivalent modifications on different elements of the metamodel, e.g., in Java, a super interface may be added to both classes and interfaces but the respective modifications operate on different metamodel elements (i.e., the `implements` reference for classes and the `extends` reference for interfaces). In these cases, it would have been beneficial to use the same name for all these operations (e.g., `addSuperInterface`) even though they have a similar signature, which could be realized by operation overloading. However, currently, DeltaEcore does not support overloading. The resulting problems could be circumvented by providing different names for the delta operations, e.g., `addSuperInterfaceForClass` and `addSuperInterfaceForInterface`.

Finally, there was one case where definition of an enumeration type *within* the delta dialect would have been beneficial. The delta dialect for Java allows for setting the visibility of members such as fields or methods to the values `public`, `protected` or `private`. However, the metamodel for Java does not define an enumeration for these values but uses instances of marker classes (see above) where the respective metaclasses for types of visibility do not share a more precise common ancestor than `Modifier`, which makes it impossible to distinguish visibility modifiers from other modifiers through static typing. Hence, the delta dialect for Java realizes these methods by providing delta operations called `setPublicVisbility(..)`, `setProtectedVisibility(..)` and `setPrivateVisibility(..)`. Instead, it would have been more concise to specify a delta operation `setVisibility(Visibility v, ..)`. Due to the lack of the respective element in the metamodel, a solution would have been to define an enumeration of the options for visibility modifiers within the delta dialect and then translate them to the respective values of the metamodel within the delta operation's implementation. However, DeltaEcore currently does not support the declaration of data types within delta modules.

Yet, these caveats did not hinder the successful creation of delta dialects for all 9 inspected source languages. Despite the differences in the source languages' characteristics, the resulting delta languages have large similarities in terms of operation structure and addressing source artifact elements through identifiers, which may ease the use of new delta languages with prior knowledge of other delta languages created with DeltaEcore. Furthermore, using naming conventions within the delta dialects allowed for unifying similar concepts, even if they were performed on different elements of the metamodel, such as with the `addSuperInterface..(..)` delta operation mentioned above. The resulting delta languages are fully compatible with one another in the sense that they may be used within a single project and with the same set of tools.

Concluding on these findings, delta languages could be created using the presented language generation facilities. Standard delta operations could be derived from a source language's metamodel and custom delta operations could be utilized to realize specific operations. Delta dialects allow for distinguishing between configuration and evolution delta operations to only provide adequate operations for handling variability in space. The caveats identified in the case studies could be circumvented with the provided means and did not affect the suitability of the created delta languages. Hence, the first part of RQ2 can be answered positively, as it is possible to create delta languages with the introduced language generation facilities, which may be used to represent the manifestation of changes associated with variability in space and time.

8.4.2.2. Results and Discussion of Delta Language Application

To manifest variability within realization assets, the previously created delta languages had to be applied in order to create delta modules that perform modifications on the base variants of the respective realization assets. For this purpose, the case studies on the TurtleBot driver, the metamodel family for role modeling and the SPL for feature modeling notations specified a total of 46, 52 and 65 delta modules, respectively. Table 8.7 lists the essential characteristics of the delta modules used within the case studies in detail.

			Configuration vs. Evolution	
	Delta Languages	Delta Modules	Configuration Delta Modules	Evolution Delta Modules
TurtleBot Driver	7	46	15 (33%)	31 (67%)
Metamodel Family for Role Modeling	1	52	28 (54%)	24 (46%)
SPL for Feature Modeling Notations	2	65	32 (49%)	33 (51%)

Table 8.7.: Essential characteristics of the delta modules used within the case studies in Chapter 8.

The distribution of delta modules to configuration and evolution delta modules suggests how strong the focus of a software family is on variability in space and in time, respectively. The percentage of evolution delta modules used within the case studies lies between 46% and 67%. However, the absolute number of delta modules is not necessarily representative for the complexity of manifesting variability inherent to the software family of a case study: DeltaEcore allows for using multiple different blocks that each utilize a (possibly) different delta dialect within a single delta module to alter artifacts that have a high logical cohesion but use different source languages, e.g., a component's implementation in Java and its associated error propagation path described within a CFD. Through this mechanism, the number of required delta modules was greatly reduced when addressing realization artifacts of different source languages. For example, instead of using 5 separate delta modules, it was possible to use a single delta module with 5 blocks to alter source code in Java along with safety certification material specified as SFT, CFD and CL, which is further referenced using the GSN. In addition, it was possible to reuse the same delta language within multiple blocks of one delta module to modify different artifacts of the same source language, e.g., two different Java files, in order to maintain a clear separation of concerns with regard to the manifestation of variability. For example, within the case study on the TurtleBot driver, as much as 7 delta blocks were present within a single delta module. Hence, the use of blocks within delta modules reduced scattering of logically cohesive sequences of delta operations performed on different realization artifacts and drastically decreased the number of delta modules that had to be created and maintained.

Using the delta operations specified within the delta dialects for the respective source languages, it was possible to manifest the changes associated with variability in space and time. Within the blocks of each delta module, delta operations were used to alter elements of the base variant or a previous state of a realization asset. For this purpose, elements of the targeted artifact had to be addressed in order to use them as operands to the delta operations. For this purpose, the identifier mechanism provided by the tool suite DeltaEcore and realized within each delta dialect were employed. These identifiers were direct (e.g., with SFTs and the GSN), hierarchical (e.g., with Java and Ecore) or a mixture of both (e.g., with CFDs), as listed in Table 8.6.

Within the delta modules, identifiers are mere textual strings. They are only resolved to elements of the addressed artifact when applying the delta operations of a delta module. As a consequence of this procedure, problems with unresolvable identifiers may, theoretically, be retrieved earliest when deriving a variant that requires the respective delta module to be applied. While this may, conceptually, complicate writing of delta modules, in practice, it is a minor concern, as DeltaEcore generates the relevant parts of a partial variant used as basis for a delta module automatically in the background. Furthermore, validity of identifiers within the current delta module is checked using constraints that are executed upon change to the delta module. Hence, error messages for unresolvable identifiers may be retrieved immediately. During the case studies, this procedure helped to avoid errors because illegal identifiers were marked as erroneous immediately with an appropriate error message. For example, within the TurtleBot driver, an identifier was used that referenced a class by its old name after it was renamed in a previous evolution delta module. Instead of resolving to `null` during execution, the error was marked immediately in the editor, so that it could be resolved. Furthermore, identifiers may also raise errors if they cannot be resolved unambiguously. For example, the delta dialect for Ecore used in the case studies on the metamodel family for role modeling as well as the SPL for feature modeling notations allows referencing an `EOperation` by providing its qualified name along with its number and types of parameters. For ease of use, it is possible to substitute the wildcard token `..` for the parameters to match whatever signature of the operation is found provided that it is unique. If an overload of the operation exists, this identifier cannot be resolved unambiguously and a respective error message is issued immediately for the identifier, so that the error can be resolved.

However, the case study also exposed some inconveniences when applying delta languages within delta modules. For one, in certain cases, it would have been beneficial if delta operations had a return value. For example, in either one of the case studies, delta operations were employed to create new instances of model entities when adding elements. If the creating delta operation would return the value of the new element, it could be stored in a variable for future use with other delta operations. Currently, after creating and adding the new model element, it has to be retrieved by explicitly using its identifier and, potentially, storing the result in a variable. The use of return values could reduce the need for calling a delta operation and invoking an identifier to just the prior.

Furthermore, the case study on the metamodel family for role modeling used evolution delta modules to realize extensive changes to the base variant of the system: In the *2014-07-14* evolution stage, the main package of all metamodel elements was altered. As the delta dialect for Ecore utilizes hierarchical identifiers, in consequence, the identifiers of *all* model elements were altered as well. This implicitly demanded an order of the delta modules, where all those associated with versions previous to the *2014-07-14* evolution stage had to be applied before those associated with the *2014-07-14* and later versions. Even though this is not a direct problem of applying delta languages, it manifested during development: Creation of variants with DeltaEcore sometimes finished successfully and sometimes failed during application of delta operations because of unresolvable identifiers. The problem could be solved by making the implicit demands on the order of delta modules explicit by specifying adequate application-order constraints.

Concluding on these results of the case studies regarding application of delta languages, delta modules could be used to manifest the changes associated with variability in space and time within realization assets. The option to specify multiple blocks using different delta dialects within a single delta module reduced scattering of logically cohesive modifications and decreased the number of delta modules that have to be created and maintained individually. The use of direct or hierarchical identifiers allowed for respecting the individual characteristics of a source language in addressing elements of its respective artifacts. Inconveniences in applying delta languages revealed during the case studies could be circumvented using the means provided by the DeltaEcore tool suite. Hence, the second part of `RQ2` can also be answered positively, as it is possible to adequately represent the manifestation of changes associated with variability in space and time using the created delta languages to specify configuration and evolution delta modules.

8.4.3. Results and Discussion of RQ3: Variant Derivation Procedure

To inspect `RQ3`, for each of the case studies, a number of variants was created. For this process, first, conceptual configurations were defined consisting of a suitable selection of features and feature versions from the HFM. A configuration could be determined visually by using the graphical representation of the HFM editor (see Figure 7.20). Furthermore, it was possible to define configurations using the textual format provided by DeltaEcore (see Figure 7.19). Using this mechanism, configurations could further be stored to later recreate an identical variant or to create a configuration for a similar variant (e.g., when using a different version constellation). Before using the configuration for variant derivation, it was further checked for validity with regard to the configuration knowledge. Through this procedure, missing required features as well as selection of features and versions that violate configuration knowledge could be identified. In case of an error, the problematic element in the configuration (if present) and the violated configuration rule of the HFM semantic or the version-aware constraint were highlighted. Hence, the subsequent manual procedure to correct the existing problems could be guided.

The variants to create during the case studies were selected by determining configurations using 3 different approaches: In the first approach, configurations were created that respected the individual evolution stages of the case studies. Only feature versions were selected that were defined within the respective evolution stage. If a feature defined no new versions in an evolution stage, the most recent one could be selected. Hence, this mechanism only yielded configurations that had initially been anticipated. In the second approach, plausible configurations were determined that had not necessarily been explicitly anticipated. For this procedure, version constellations were chosen that did not respect the evolution stages, e.g., by selecting a version of a child feature that is less recent than the version of its parent feature. In the third approach, selection of versions was completely arbitrary as only features were pre-selected and the automatic version selection procedure from Section 6.3 was employed with different objective functions to select a suitable constellation of versions. To further increase the variation of versions, a subset of all versions was selected at random. Some of these cases constituted invalid partial configurations that were discarded. In each of the approaches, features were selected arbitrarily but it was ensured that they respect both the configuration knowledge as well as, for the first approach discussed above, the evolution stage a feature was introduced in.

The conceptual configuration could then be resolved to the required delta modules by employing the mapping model. Furthermore, a suitable application sequence of these delta modules could be determined. For this procedure, the HFM was analyzed for its structure, requires relations within the delta modules were examined and the externally specified application-order constraints were evaluated. For the case study on the metamodel family for role modeling languages, the latter provided a suitable mechanism to represent a baseline of versions: After changing the containing package of all elements of the metamodel in the *2014-07-14* evolution stage, all delta modules from this point onwards had to be applied after those before the evolution stage. This restriction could directly be expressed within the application-order constraint model, where all previous delta modules formed one group and the rest of the delta modules formed a second group that may only be applied afterwards. These constraints were respected in determining a suitable application sequence. Using the determined sequence, the delta operations of the blocks of each delta module were applied sequentially to transform the realization assets of the base variant to those of the target variant.

Validity of the created variants was checked using two procedures: First, for each case study, a suite of unit tests was created that contained tests for each feature. Furthermore, the tests of each feature were copied and adapted to the changes of the respective versions of the feature. Hence, for each configuration, a suitable test suite could be determined to test the respective realization assets in isolation. In addition, a manual inspection of the realization assets was performed to validate their correct interaction. Both these procedures confirmed that variants were created according to the selected configuration and the respective associated delta modules.

Concluding on these findings regarding the variant derivation process, conceptual configurations could be defined both textually and graphically. They could further be transformed to an appropriate set of delta modules by employing the mapping model. In addition, an appropriate application sequence could be determined for delta modules by evaluating the HFM, requires relations specified internal to the delta modules as well as application-order constraints defined external to the delta modules within a dedicated model. Applying delta modules in this sequence yielded variants of the software family that encompass variability in space and time. Hence, RQ3 can be answered positively, as it is possible to use the variant derivation procedure to create products of the software family that represent combinations of different features and their versions.

9. Conclusion

The conclusion consists of three parts: First, design decisions of the approach as well as boundaries for its applicability are discussed. Second, possible future application areas are presented. Third, the contributions of the thesis are recaptured along with notes for possible applications.

9.1. Discussion

Within the approach presented in this thesis, certain design decisions were made that have an impact on potential application. The following sections discuss benefits and limitations of the approach with regard to the most relevant points.

9.1.1. Supported Evolutionary Changes

The focus of the presented approach is on evolution of individual variable assets as work previous to this thesis determined this as very common in evolution especially of SECOs [SA13]. Hence, the approach is useful when deriving variants with variability in space and time resulting from individual evolution of variable assets and their interdependencies. However, evolution of the configuration knowledge is only supported as long as the configuration space is maintained or extended, e.g., through new versions or optional features. Evolution of the configuration knowledge that reduces the variant space is not supported.

In [TBK09], four categories of SPL evolution are distinguished by their effect on configuration options: *Generalization* adds additional configuration options, *refactoring* maintains existing configuration options, *specialization* reduces configuration options and *arbitrary edits* both add new and reduce previously existing configuration options. During the case studies presented in Section 8, a number of evolutionary changes were performed resulting in new versions of individual features that can be classified using these categories.

All changes performed to the *structure* of the HFM merely added new optional features so that the feature model still permits all previously valid configurations to be derived. Changes to the structure of a feature model that would invalidate previously valid configurations cannot be captured using HFMs, e.g., when adding a new mandatory feature or changing an or-group to an alternative-group. Hence, only extension of configuration options is allowed.

With regard to the aforementioned categories of evolution, the presented approach principally supports *refactoring* and *generalization*. However, it does not support *specialization* and *arbitrary edits* as the associated changes on the variability model may lead to the invalidation of previously valid configurations.

Nevertheless, it is possible to use the version-aware constraint language to formulate some of these changes using logical expressions in HFMs, e.g., by TurtleBot [$\geq$ 2.0] $\rightarrow$ Detection to effectively make obstacle detection mandatory from version *2.0* of the `TurtleBot` onwards.

Handling these changes with a dedicated mechanism is particularly relevant for SPLs with a frequently evolving structure of the variability model, e.g., in early stages of SPL development, as representing them using version-aware constraints is cumbersome. The same holds for SECOs. However, this challenge is out of scope of this thesis.

9.1.2. Conceptual Representation of Variability in Time

Capturing both variability in space and time for product definition in an HFM lifts variability in time from the solution space containing realization assets to the conceptual problem space: In SECOs, such as Eclipse, Android, OSGi or Linux, dependency of variable artifacts on versions of other artifacts is specified solely in realization assets such as manifest files or build scripts. However, this information is required on a conceptual level when defining product configurations.

A formally defined model capturing these aspects of variability in time, such as an HFM, helps in establishing valid configurations, e.g., by suggesting suitable combinations of versions to satisfy a pre-configuration of features. as introduced in Section 6.3. This procedure effectively relieves users from dealing with concerns resulting from the realization of assets when they want to focus on configuration of functionality.

Besides technical applications, HFMs may also be useful for stakeholders not concerned with technical details of the realization assets in the solution space such as managers or end-users. When using a graphical representation of HFMs, such as the one presented in Figure 4.2, it is possible to relatively easily grasp aspects of variability in space and time as well as relations of features and versions on a conceptual level. It is further possible to provide an alternative reduced view on HFMs depicting only variability in space as in feature models by limiting the representation to features. The respective versions for selected features may then be determined using the procedure described in Section 6.3.

9.1.3. Perception of Versions as Incremental

Versions and their respective evolution delta modules are perceived as being incremental to their respective predecessors. Instead, it would have also been possible to design evolution delta modules to be self-sufficient. In that case, each evolution delta module would have to perform all changes to an artifact required to transform *the initial version* instead of *the previous version* of the variable asset to the respective version represented by the evolution delta module. However, as evolution rarely completely revokes changes performed in a previous version, this would lead to accumulation of delta operations for all previous versions of an artifact within a single evolution delta module making the approach infeasible for longer evolution histories.

Yet, once certain deprecated versions of a feature are no longer supported for configuration, self-sufficient evolution delta modules could simply be removed. With an incremental approach, the transformation operations of the outdated evolution delta modules are still required. However, the work of Schulze et al. on refactoring delta modules [SRS13] demonstrates how the calls to delta operations of delta modules can be merged. This procedure could be used to merge delta operations of deprecated evolution delta modules into still supported evolution delta modules so that the prior could be removed. However, this challenge is out of scope of this thesis.

9.1.4. Version Numbering Schemes

HFMs allow for assigning arbitrary labels as numbers for versions. There are principally no constraints on the scheme of assigning these names, so that it is possible to employ established conventions such as using a combination of incremental numbers for major/minor/micro revisions (e.g., 3.1.4). However, other version numbering schemes are also supported, e.g., by using dates or arbitrary names.

The most natural way of deciding on the version numbers for features is to align them with the versions of their respective realization assets as, e.g., assigned by a version management system such as SVN. However, this requires an adequate encapsulation of features in their realization assets, e.g., using one component per feature that has a dedicated version number. For arbitrary architectures on realization assets, a direct alignment of versions of features and their realization assets may not be feasible, e.g., when a feature is realized within multiple different classes each having a separate distinct version.

Instead of deciding which of the different realization versions to use as feature version, it may be more suitable to use the baseline version number of the repository as, e.g., assigned by SVN to solve similar problems with versions of software products consisting of multiple realization assets in different versions. However, this is only possible when all variable assets can agree on a common baseline version, e.g., because they are hosted within the same version control repository. For software families developed in a distributed manner, release trains may help mitigate the problem as they give a point for synchronizing version information, e.g., as with the reoccurring releases of the Eclipse platform and associated extensions ("Kepler", "Luna" etc.).

Furthermore, it is also possible to use an entirely separate versioning scheme for feature versions and their realization assets if required and suitable. However, in this case, the mapping between the versions of both schemes has to be maintained separately if this information is required.

9.1.5. Created Delta Languages

Section 5.5 presents the conceptual foundation for delta language creation facilities and Chapter 7 provides a realization of these concepts within the tool suite DeltaEcore. With these approaches, it is possible to provide suitable delta languages for arbitrary source languages provided as metamodels in EMF Ecore. Furthermore, Section 5.4 presents a procedure to automatically create instances of 7 standard delta operations with defined syntax and semantics.

Considering the granularity of change, the 7 standard delta operations provide very fine grained access to artifacts of the source language. This has the benefit of being able to precisely perform modifications of the realization artifacts being subjected to variability. However, it also entails the need for potentially a large number of calls to delta operations in delta modules to perform more complex modifications. If these complex changes (or significant parts thereof) are instances of reoccurring operations, a remedy to this problem is to specify more complex custom delta operations with user defined semantics. Within DeltaEcore, the semantics of a delta operation are specified as Java source code performing the intended modification on the model elements provided as parameters to the delta operation. For a custom delta operation, the respective implementation may be completely unique. However, due to the fine-grained nature of standard delta operations, it is also possible to realize the semantics of custom delta operations by calling the implementation for interpreting standard delta operations. In effect, this creates compound delta operations that perform more complex operations by combining fine grained standard operations. This is beneficial as it defines the semantics of complex operations in terms of the already known standard delta operations, which may be exploited for analyses or for recording performed changes as delta modules (see Section 9.2).

In addition, the created delta languages utilize direct addressing (see Section 3.2.2) instead of navigating the containment hierarchy of source language artifacts explicitly. Nevertheless, identifiers may use a hierarchical scheme, such as qualified names consisting of multiple element names separated by dots (e.g., see the delta language for Ecore used in Section 8.2 and Section 8.3).

Furthermore, the delta languages created with the language creation infrastructure presented in Chapter 5 and its implementation described in Chapter 7 provide a textual concrete syntax to be used for creating respective delta modules. Hence, delta languages of DeltaEcore are currently only textual. However, the model-based foundation principally allows for graphical delta languages as well, which might integrate more seamlessly with graphical languages such as SFTs, CFDs or the GSN. The principle feasibility of graphical delta languages was presented for Matlab/Simulink in [HKM+13]. However, finding a generally suitable graphical notation is a challenge orthogonal to those of the thesis, which is considered out of scope.

9.1.6. Scalability of Approach

To test the concepts presented in this chapter and their realization (as presented in Chapter 7) on large scales, a generator for HFMs was implemented that creates plausible HFMs with regard to the underlying feature model as well as the structuring of feature versions along development lines. Various parameters allow for tuning the complexity of created HFMs, such as the depth and branching factor of the underlying feature tree or the average number of versions per feature, as well as the general alignment of versions along development lines. Generated constraints consist of implications between possibly version-constrained features. However, at present, no complex version-aware constraints are generated.

The generated HFMs were used as input to the implementation of the presented concepts (see Section 7). HFMs consisted of approximately 300 features and 1000 versions. Experiments showed that response times of algorithms, such as the automatic version selection for pre-configurations of features (see Section 6.3), were below 10 seconds.

Delta modeling as variant derivation mechanism has been shown to be applicable in real-world scenarios in multiple case studies [HRRS12, HRRS11, LSKL12, BDS13, KLL+14]. However, transformations to express changes associated with variability in time add an additional level of complexity. The HFM generator mentioned above was not extended to also produce delta modules due to their complex nature and the associated risk of synthesizing delta modules that are not representative of real world applications.

Regarding the visual representation, HFMs face problems similar to those of other graphical notations, such as feature diagrams, when used for large models. Complex representations may contain an abundance of elements and exceed the space available on screen, which makes it complicated to grasp the presented information. These problems cannot be avoided completely as they are inherently coupled with the amount of information that has to be displayed. To mitigate these problems, it may be possible to use techniques to better cope with the amount of presented information, such as zooming or selective hiding of versions, e.g., by folding the respective compartments of features. However, addressing these problems in the visual representation is out of scope of this thesis. Apart from these issues, the presented approach may conceptually be scaled to industrial-size problems.

In either case, management of variability in time in addition to variability in space introduces a significant level of complexity to the development process. In some cases, it may be possible to reduce or alleviate the need for management of variability in time by a strict release management, e.g., by only allowing complete update of all features in a product simultaneously. However, in some SPLs and especially SECOs, it is necessary to include variability in time into the configuration process, e.g., when different vendors are developing variable artifacts independently and customers are configuring products autonomously. In these cases, the presented approach aids in representing and handling configuration and evolution in software families by incorporating aspects of both variability in space and time in the derived variants.

9.2. Possible Future Application Areas

This thesis presented an approach for integrated management of variability in space and time within software families. Beyond this application, extensions to the introduced concepts and provided realizations may open up potential for future application areas beyond the scope of the thesis. The following section elaborates on a selection of these areas providing initial ideas as well as further discussion regarding challenges and potentials when realizing them.

9.2.1. Extend to Full Software Ecosystem Feature Model

HFMs as presented in this thesis allow for capturing of variability in space and time as features and feature versions, which may be arranged along development lines. With these characteristics, it is possible to conceptually handle configuration knowledge of SPLs and SECOs, even when they are subjected to evolution. However, SECOs have distinct characteristics, which require different treatment than SPLs–most notably multiple distributed contributors that create and maintain an open variant space (see Section 2.3). These aspects may have to be reflected on a conceptual level, e.g., in a variability model, as they affect configuration knowledge. HFMs provide a basis for these possible extensions.

For example, it would be necessary to provide an infrastructure where the variability model may be created and maintained by geographically distributed vendors. For one, this may require an implementation of a model spread over multiple repositories where individual parts are controlled by different vendors on a technical level. In addition, it would be necessary to determine which types of operations may be performed by individual vendors so that, e.g., it may be permissible for some vendors to create new feature versions but not new features.

Furthermore, it would be possible to integrate a role-based access control mechanism into HFMs that assigns particular roles to sub-trees of the HFM, individual features, or even development lines for versions of a feature. Additions and changes to the respective structure may only be performed by the respective vendor playing that role. In consequence, parts of the feature model could be protected from arbitrary edits so that, e.g., the core of the software family may only be altered by the respective core developer team, whereas extensions may be modified or added by a more general audience.

These extensions to HFMs would help not only to cope with multiple developers, but might also allow for more specific reasoning over an otherwise completely open variant space as some parts of the variability model can be perceived as being “locally closed”, e.g., when the core development team assures that no changes will be performed in a certain period of time.

9.2.2. Model Software Ecosystems

The concepts presented in this thesis may be used to model (parts of) evolving SECOs such as the Eclipse SECO. Even without the extension to HFMs to fully cope with SECOs described above, a significant amount of the respective software families may be modeled. For example, the Eclipse SECO uses extension points to individual components where other vendors may add contributions that offer particular functionality, provided in the form of components. Hence, components may be perceived as variable units of the Eclipse SECO that can be represented by features of an HFM.

The configuration procedure using extension points can be perceived as an instance of a compositional variability realization mechanism, which is subsumed by the transformational variability realization mechanism delta modeling employed in this thesis (see Section 2.2.2). Hence, the means of realizing variability in space of the Eclipse SECO could be emulated by fixing a base variant of a component and then providing configuration delta modules for it. An HFM could represent the features associated with the variable components on a conceptual level.

Furthermore, both the components as well as their extensions are subjected to evolution comprising the dimension of variability in time. In the Eclipse SECO, this dimension is visible to both vendors and end users through different complete instances of a component each reflecting a snapshot at a particular time. However, the versions of these components could as well be retrieved by starting out with one base variant of the component and then applying a sequence of stored transformation operations towards the intended component version. A similar mechanism can be realized with the concepts of the thesis by specifying a series of evolution delta modules to the base variant of a component. An HFM may capture the different versions of a component as feature versions and arrange them along their logical development lines as well as associate them with the evolution delta modules that realize the respective changes. Hence, it is principally possible to represent the Eclipse SECO with the means provided within this thesis and a similar line of argumentation may be applied for other SECOs such as Android.

However, the benefits of employing the approach of this thesis to less mature SECOs than the ones mentioned before seem even more promising: Within a SECO, the economical success of extension vendors greatly depends on the capabilities of the platform leader (see Section 2.3). One essential task of the platform leader is to provide a technological platform that is adequate for the application area and can sufficiently be extended by extension developers. Assuming that, at an earlier stage of a SECO, the requirements of extension developers may not be as clear as at a later stage, the latter challenge may be hard to address by the platform leader. Furthermore, considering that the platform in its functionality may still need extension, resources of the platform leader with regard to determining and providing explicit extension options for suitable extension points may be scarce. For a variability realization mechanism strictly depending on explicitly declared extension points such as with Eclipse, this may pose severe constraints on extensibility of the platform. In contrast, an invasive procedure, such as the use of delta modules, does not depend on explicitly declared extension points but can alter the structure of artifacts by specifically targeting substructures of interest. Through this mechanism, even early stages of a SECO with little knowledge of potential needs of extension vendors may be handled. Similarly, even more mature SECOs with a weak platform leader may still benefit from a variety of extensions.

However, hiding the internal structure of extensible artifacts in the sense of a black box by only providing explicit extension points has benefits with regard to the degree of liberty when altering the realization of that artifact. Hence, a more mature SECO may benefit from this approach when the needs of extension providers regarding extension points are more evident. The approach presented in this thesis can principally be combined with an explicit extension point mechanism so that a blend of the invasive and non-invasive approaches is possible. In consequence, it may be of interest to examine how to progress from an invasive approach using delta modeling towards a non-invasive approach such as Eclipse extension points. This way, it would be possible to capitalize on the increased degree of freedom with regard to extensions for less mature areas of a SECO and gradually progress towards a state easing maintenance but limiting the possibilities for extension in areas of the SECO, where it is both feasible and appropriate.

9.2.3. Extract Hyper-Feature Model Versions and Record Delta Modules

For larger systems with long evolution histories, explicit modeling of HFMs may be tedious, as many versions may have to be captured to adequately represent all versions. A purely manual approach may not be feasible due to the extensive effort required.

As remedy, it is principally possible to semi-automatically extract information of versions for an HFM from solution space assets or their version history. For example, to gain insights on the structure of the Eclipse SECO as preparation of this work, OSGi manifest files present in Eclipse plug-ins were parsed to automatically gather information on the dependencies between versions of extensions for an HFM. Furthermore, a prototypical implementation was created to utilize information from the history of SVN source code repositories to suggest new HFM versions when similar artifacts are affected in a commit as in commits creating a previous version to create tentative contents for the HFM of the case study presented in Section 4.7. Using this information, it may be possible to make a suggestion for added feature versions on a conceptual level in the HFM as well as, to a certain degree, the manifestation of associated changes within evolution delta modules. However, possibilities for reconstructing the use of delta operations from a difference model between two versions of a realization artifact are very limited, as even a mere move delta operation cannot be distinguished from the combination of a remove and add delta operation.

To ease creation of delta modules, it would be possible to proactively capture the information for evolution delta modules by providing a recorder mechanism in the tools used to alter realization artifacts. The recorder would log the performed changes in a delta module instead of applying them directly. With the strictly model-based realization of the concepts of this thesis, such a mechanism could, in principal, be realized with moderate effort. The code generated for EMF Ecore provides listeners that inform of individual changes performed on models and allow storing as well as revoking them. This provides the basis for a recorder mechanism. However, in order to record modifications at the level of delta modeling, the application of individual delta operations with their parameters has to be recorded.

Hence, editors utilizing the recording mechanism would have to base the changes they perform on delta operations. However, this could be achieved due to two reasons. For one, delta dialects may specify an arbitrary number of delta operations of an arbitrary abstraction level, so that delta operations suitable for editors could be added if required. Furthermore, using the model-based approach of the thesis, the basis for editors for source languages could be generated with tools that use specific delta operations as implementation. Especially for graphical source languages this is feasible, as operations usually leave the model in a machine comprehensible state. For textual source languages it may be more difficult to provide only tools based on delta operations for editing: With parsing editors, it is common to have intermediate states of an artifact that cannot completely be perceived as being an instance of the source language, e.g., when creating invalid syntax as intermediate state during typing. However, projecting editors [SCC06, Mer10] for textual source languages may provide remedy as they ensure that changes to the source language's artifact perform only valid operations, which may be based on delta operations.

Even though the challenge of creating a delta recorder is out of scope of the thesis, the implementation of the presented approach (see Chapter 7) provides the principle structure to realize such a mechanism: Within the generated interpreter for delta operations of a delta dialect, each executed delta operation is routed to a special `DEModelWriter` class (see Section 7.1.3). When enabled, this class logs all performed delta operations as well as their parameters and ensures that solely the changes routed through it are performed to alter the model (i.e., that a delta operation has no undocumented side-effects). For the envisaged recorder mechanism, this class would have to be specialized in the sense that performed delta operations are (tentatively) applied, but also recorded within a delta module. Furthermore, for some operations, such as detach or remove, additional state information has to be saved in order to be able to revert them. After recording one delta module, the recorded changes to the affected model could be rolled back to revert it to its original state and only retrieve the delta module that would perform the recorded changes. This way, delta modules would not have to be specified manually but could be recorded during common editing of realization artifacts.

9.2.4. Introduce Metaevolution Delta Modules

Within the approach presented in this thesis, delta modules are used to perform configuration and evolution on arbitrary realization assets. Hence, the target of the transformations performed by delta operations is located exclusively within the solution space (see Section 2.2). However, in certain situations, it may be possible or even necessary to apply delta modules to artifacts in other conceptual spaces, e.g., the problem space containing the variability model.

Within the thesis, arbitrary edits to HFMs were considered out of scope, e.g., when removing a feature or assigning a new name to it. For example, in the case study on the metamodel family for role-based modeling and programming languages presented in Section 8.2, the feature `RoleInheritence` contained a typographical error that should be corrected as part of evolution by renaming the feature to "RoleInheritance". However, the change could not be performed with the concepts of the thesis but had to be performed manually. In consequence, older versions of, in this case, the feature in the HFM are no longer available, which might pose a problem, e.g., if the name change was more fundamental so that customers do not easily grasp the connection between the previous and the current name.

In this case, it might be feasible to associate the *change to the HFM* with a feature version of that very same HFM. In consequence, a *metaevolution delta module* would be required that is capable of performing changes to the configuration knowledge it is itself subjected to. This procedure would enable tracing changes, even to the HFM, back to a particular version. On a technical level, this procedure does not pose problems, as the realization of HFMs is based on EMF Ecore and, thus, can be perceived as source language for a delta language of DeltaEcore[1]. However, on a conceptual level, this procedure raises questions regarding the kind and extent of changes a metaevolution delta module may perform on an HFM as well as the ability to analyze the effects of the application.

Furthermore, it may be beneficial to subject both configuration and evolution delta modules themselves to evolution. For example, if a delta module contains calls to delta operations that in effect produce a defect of the software system, it may be beneficial to alter the delta module to avoid that defect. However, changing the delta module directly makes the previous version, which contained the defect, unavailable. Even though this may be desirable in most cases, in some cases, it may be necessary to maintain the old version if customers created implementations depending on presence of the defect. The approach of the thesis proposes to model changes fixing the defects in a separate evolution delta module and performing them after the delta module causing the defect so that all versions are maintained. While this is feasible, if the defect affects multiple development lines, the delta module for fixing it may only be reused on a realization level but, conceptually, a separate feature version has to be created on each branch of the development line. This may unnecessarily clutter the conceptual configuration knowledge with technical details. Employing *one* metaevolution delta module that addresses the delta module introducing the defect *before* the branching of development lines may avoid such cluttering.

Hence, metaevolution delta modules may be of benefit when using them to alter the conceptual configuration knowledge specified in HFMs, as well as the manifestation of that configura-

[1] As proof-of-concept implementation, a delta language *DeltaHFM* was realized using DeltaEcore in order to be able to alter HFMs on a technical level. Application of the delta operations of DeltaHFM is analogous to that of other delta languages.

tion knowledge defined in delta modules. However, the concrete means of their application and the resulting effects require further research.

9.2.5. Support Incremental Reconfiguration

With the concepts presented in this thesis, it is possible to create variants that encompass a selection of variable functionality as features at specific revisions as feature versions. With the variant derivation procedure, it is principally possible to (virtually) upgrade one variant of the software family by recreating a variant with similar features, but with more recent versions. A similar procedure is possible for downgrades that encompass less recent versions of the features, as well as arbitrary other configurations that encompass a changed selection of feature versions or even features. In either case, the entire variant of the software family is recreated by copying the base variant and applying all relevant delta modules sequentially in a suitable order.

However, such a procedure may not be suitable in terms of efficiency for larger software families when the time for creating a new variant is essential, e.g., when intending to use a Dynamic Software Product Line (DSPL). In this case, the efficiency for creating a variant that is related to a currently existing variant may be improved if both variants are sufficiently similar in terms of features and feature versions in their configurations. In such a case, instead of recreating the entire variant, it may be more sensible to only perform incremental modifications that comprise the changes between both variants. When considering the demands on a valid HFM configuration of Definition 17, the reconfiguration procedure may be split up into individual steps that can be reduced to 5 principle operations as illustrated in Figure 9.1.

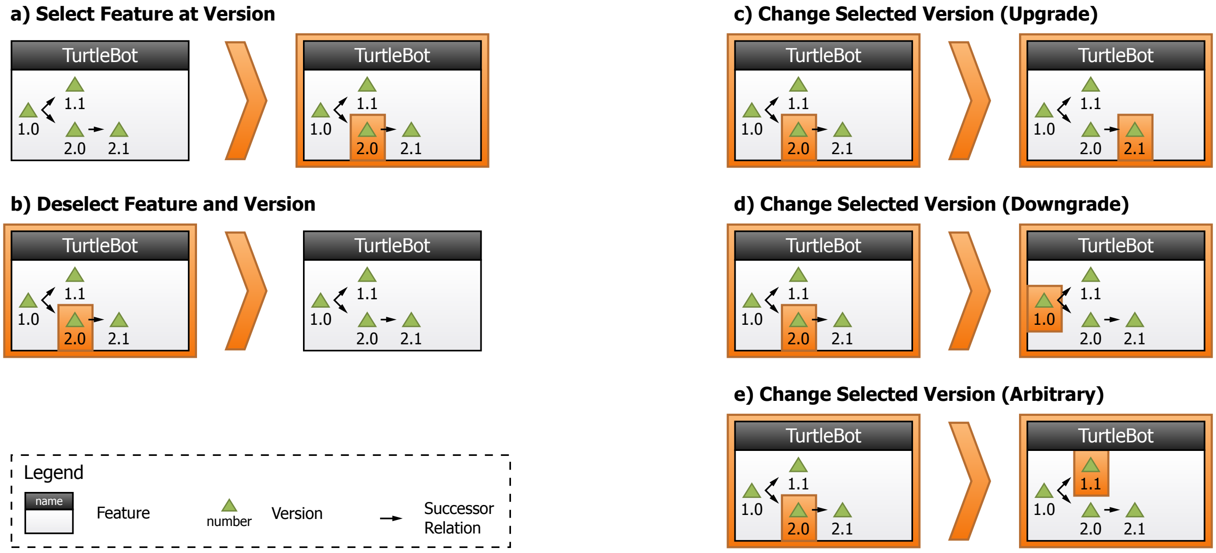

a) Select Feature at Version
b) Deselect Feature and Version
c) Change Selected Version (Upgrade)
d) Change Selected Version (Downgrade)
e) Change Selected Version (Arbitrary)
TurtleBot
1.0
1.1
2.0
2.1
Legend
name
Feature
number
Version
Successor Relation

Figure 9.1.: Principle types of changes performed during reconfiguration.

First, a feature and a version suitable for the rest of the configuration may be selected as in Figure 9.1 a). In this case, the configuration module for the feature as well as the evolution delta modules from the version without predecessor to the selected version have to be applied in that order.

Second, a previously selected feature and its version may be deselected as in Figure 9.1 b). In this case, the changes performed for the feature versions and for the feature itself have to be revoked. For this purpose, regression delta module [LSKL12] may be determined that contain operations that are inverse to those of the original delta modules. To determine inverse operations for delta operations such as detach or remove, the respective state before application would have to be saved during the variant derivation procedure. Applying the regression delta modules for the evolution delta modules in inverse order would negate the changes of selecting that particular version. Likewise, applying the regression delta module for the configuration delta module would negate the changes of selecting the feature in question. In consequence, the feature and its version would be deselected in the second variant.

Third, a version may be upgraded when the configuration of the second variant contains a feature version that is on the same development line as the previously selected version, but located further towards the end. In both configurations, the containing feature has to be selected. In this case, the upgrade can be performed by sequentially applying the evolution delta modules located between the previously selected less recent and the currently selected more recent version.

Fourth, a version may be downgraded when the configuration of the second variant contains a feature version that is on the same development line as the previously selected version, but located further towards the beginning. In both configurations, the containing feature has to be selected. In this case, the downgrade can be performed by applying the regression delta modules for the evolution delta modules located between the previously selected more recent and the currently selected less recent version.

Fifth, a version may be changed arbitrarily when the configuration of the second variant contains a version of the same feature that is located on a different development line than the previously selected version. In both configurations, the containing feature has to be selected. With respect to realizing this type of configuration change, it is possible to reduce the case to first downgrading the version to the most recent feature version common to both development lines in question and then upgrading it to the version of the configuration for the second variant.

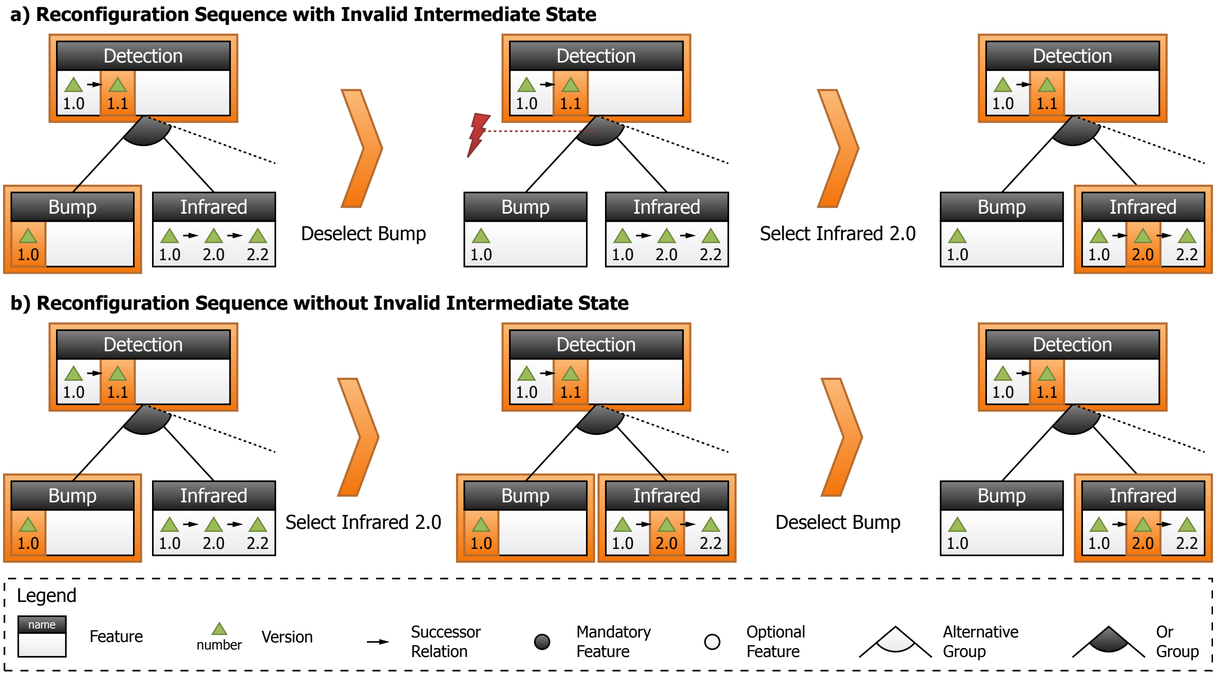


Figure 9.2.: The order of reconfiguration operations may decide whether a) there are invalid intermediate states or b) there are only valid intermediate states.

However, when combining these reconfiguration operations to perform a reconfiguration from the first variant to the second, the order in which these operations are performed is essential as illustrated by Figure 9.2. When choosing an arbitrary order of reconfiguration operations, it is likely that the intermediate state of the configuration and, thus, the variant at that time, is invalid with regard to the configuration knowledge. Figure 9.2 a) illustrates this case where the or-group of feature `Detection` is violated as, temporarily, no child feature is selected. In this case, the invalid intermediate state may be avoided by reordering the reconfiguration operations as depicted in Figure 9.2 b), so that no invalid intermediate state is created. Using this ordering may ease the reconfiguration process, e.g., in a DSPL where the intermediate state will appear during run time and, thus, may be relevant to the integrity of execution. However, such reordering is not possible in all cases, e.g., when changing selected features in an alternative group, where either none or both features can be selected during an intermediate state and both options are invalid.

Furthermore, the reconfiguration procedures utilizing delta modules and regression delta modules may only be possible, when considering the most basic structure of the artifacts presented in this thesis, where features are directly related to configuration delta modules and versions are directly related to evolution delta modules, but no further, more complex, mappings, application-order constraints or explicit requires relations between the delta modules exist. Devising a strategy for incremental reconfiguration using HFMs and evolution delta modules is out of scope of this thesis but the potential benefits regarding reconfiguration for variability in space and time even during run time may justify further investigation into the feasibility of such operation.

9.2.6. Apply for Evolution Analysis and Planning

Furthermore, using HFMs as formally founded representation of versions for configuration allows for performing analyses on aspects of both variability in space and time. For example, analyses detecting dead features, which may not be selected in any configuration, can be extended to also detect dead versions that may no longer be selected. This may happen, e.g., when a mandatory feature has an associated constraint explicitly demanding that a certain version of another feature must not be present in a configuration. The information obtained from such analyses may be used to consolidate versions in the evolution history, such that no longer required versions may be removed from the HFM.

Besides inspecting past evolutions, HFMs may also be employed in planning of future evolutions of an SPL or a SECO. Before releasing new versions of variable assets to customers, it may be beneficial to perform an impact analysis by checking the effect on possible configurations available after evolution. For this purpose, intended versions of a feature may be modeled with their respective version-aware constraints in addition to the values currently present in an HFM. By performing satisfiability checks on previous configurations that should still be possible with the newly added versions, it is possible to check whether incompatibilities introduced as part of evolution hinder definition of those products. Thus, problems resulting from evolution with currently existing configurations can be identified on the conceptual level of HFMs and avoided before release of the new versions.

9.2.7. Enable Evolution of Variable Safety-Critical Systems

Switching between configurations of equivalent features but in different versions may be beneficial in certain situations. For example, safety-critical systems need certification of a specific state of realization before they can be put into operation. Hence, variable safety-critical systems may only encompass those features and feature versions in a configuration that were part of certification when the system is to be used in a safety-critical context. For example, in the running example of

Chapter 1, theoretically, safety-certification of an automatic obstacle avoidance of the TurtleBot might have been issued for combination of `TurtleBot` *1.0*, `Autonomous` *1.1* and `Infrared` *2.0*.

Hence, even though more recent versions of features may bring improvements, it may only be possible to use them after a delay in time as the repeated certification of a configuration that encompasses the newer versions of the features for use in a safety-critical environment may take time. However, it may also be possible that the system is used in a context that is not safety-critical where certification of versions present in a configuration is not required so that the system may capitalize on improvements of more recent versions.

When creating variants of the software family before run time, the fully model-based approach of this thesis may greatly aid the certification process by providing traceability for the artifacts present in a variant. Hence, it is possible to trace back not only which feature but also which *version* of a feature is responsible for a particular element in variant, such as a specific line of code. Provided that there are previous constellations of features and feature versions that have undergone the certification process, it would be possible to determine the difference in variants between the previous and the current variant and capture it in a dedicated model. Tracing back the origins of the elements in the difference model to individual features and their versions may help speed up the certification process, and thus reduce its costs, as not necessarily the entire variant has to be inspected.

When creating variants of the software family during run time, such as in a DSPL, the switch of configurations, and thus potentially certification levels, may be performed during operation. With the use of HFMs, configuration changes may even extend to changing versions of variable assets. In the TurtleBot example, this case may exist when the robot is not moving and using the distance sensor solely for non safety-critical applications, such as detecting people to play an audio greeting, e.g., for purposes of advertisement. In such a case, the update `Infrared` *2.2* might be employed as obstacle avoidance is not required. However, the mode of operation may directly progress into a safety-critical environment, e.g., when the robot is used to service beverages at the same venue and has to avoid obstacles in order to not spill fluids. When embedding the approach of this thesis into a DSPL environment, it would be possible to switch between the safety certified but less recent and the non-certified but more recent versions of the configuration during operation.

The information of a certified (partial) configuration, or where possible, certified versions of features may be captured along with the artifacts presented in this thesis. For one, it might be possible to provide a dedicated model with (partial) configurations and their certification level that may be used in the respective safety-critical application scenarios. Furthermore, it might be possible to annotate certification levels on entities of the HFM, e.g., if it is possible to issue a certain safety-certification level for an individual version of a feature. A configuration procedure using HFMs may then be extended to respect these additional constraints regarding previously certified version-aware configurations. For example, the procedure for automatic version selection presented in Section 6.3 may be extended so that, along with a pre-configuration of features, a desired certification level has to be provided that has to be respected by the selection of versions determined by the algorithm. In consequence, operation of a safety-certified variant could be ensured where required, while still benefiting from the potential improvements of more recent versions, where a certified variant is not required.

9.3. Contribution

The work of this thesis presents an integrated approach for managing variability in space and time. It makes individual contributions in the areas of a variability model, a variability realization mechanism and a variant derivation procedure. These contributions allow for handling the update behavior of users of software families in a more extensive way than established approaches for software families. The following sections first recapture the individual contributions of the thesis and then discuss the impact on handling update behavior to demonstrate the contributions' ability of coping with variability in time.

9.3.1. Individual Contributions

Figure 9.3 illustrates the individual contributions in the areas of a variability model, a variability realization mechanism and a variant derivation procedure. The following paragraphs briefly recapture the essential points of each contribution.

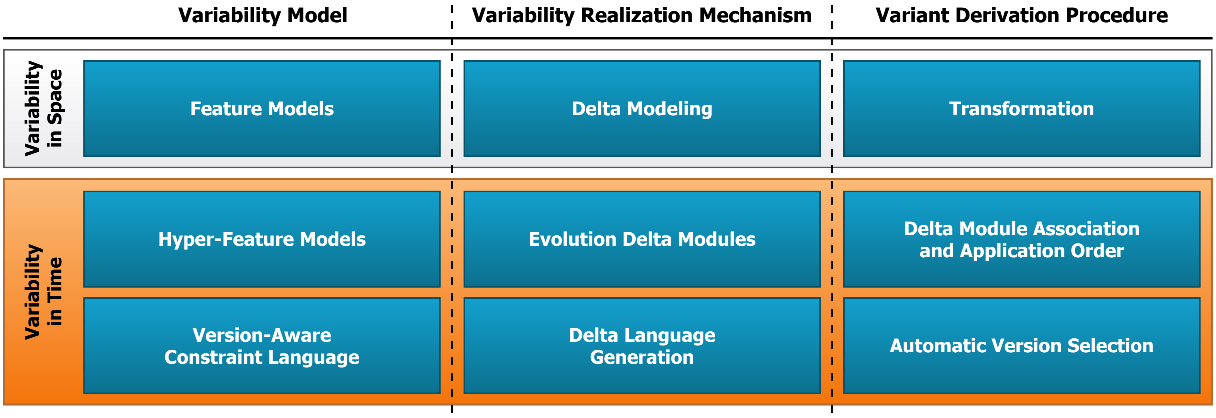

Variability Model
Variability Realization Mechanism
Variant Derivation Procedure
Variability in Space
Feature Models
Delta Modeling
Transformation
Variability in Time
Hyper-Feature Models
Evolution Delta Modules
Delta Module Association and Application Order
Version-Aware Constraint Language
Delta Language Generation
Automatic Version Selection

Figure 9.3.: Overview of the contributions of the thesis.

Hyper-Feature Models (HFMs) defined in Section 4.3 capture variability in space and time on a conceptual level. Common feature models are extended by an explicit notion of configurable feature versions that may be arranged along tree-shaped development lines. Semantics of HFMs govern configuration options regarding versions by requiring that exactly one version per selected feature must be present in a valid configuration. Furthermore, versions are perceived as being incremental so that selection of one version implicitly selects all its predecessors on the same development line.

The **version-aware constraint language** from Section 4.5 allows for the specification of cross-tree constraints on HFMs to further restrain configuration options. It is possible to express demands on and incompatibilities with versions and version ranges. Two essential types of structures are provided for versions: Version range restrictions specify a range of possible matching versions by providing a lower and an upper bound version. Relative version restrictions specify a range of possible matching versions using an operator in relation to a specific version, e.g., to address all versions after the given one on the same development lines.

Evolution delta modules introduced in Section 5.3 realize changes associated with variability in time and are an explicit distinction from configuration delta modules that realize changes associated with variability in space. Specifically dedicated evolution delta operations of a delta language provide functionality that may only be used in the course of variability in time and, thus, exclusively within evolution delta modules.

With the **delta language generation** facilities presented in Section 5.4 and Section 5.5, it is possible to create delta languages for arbitrary source languages provided as metamodel based on EMF Ecore. By analyzing a source language's metamodel, it is possible to generate large parts of a respective delta language automatically by creating instances of 7 types of standard delta operations with defined semantics. Furthermore, custom delta operations with special semantics may be provided manually. Each delta operation may be declared as either configuration or evolution delta operation.

The **delta module association and application order** of Section 6.2 and Section 6.4 form the basis for deriving variants with variability in space and time. A mapping model associates version-aware logical expressions over features and feature versions with sets of delta modules. With a given configuration, a set of relevant delta modules for a variant can be determined. Using the structure of the HFM, explicitly specified requires relations and application-order constraints, a suitable application order is determined for these delta modules.

The **automatic version selection** procedure of Section 6.3 eases the process of determining configurations that encompass aspects of variability in space and time. For this procedure, the HFM, its version-aware constraints and a supplied valid preselection of features are encoded as Constraint Satisfaction Problem (CSP). A CSP solver determines the best solution with regard to a supplied objective function. The result is a constellation of suitable versions that fulfills the configuration rules of the HFM and its associated version-aware constraints.

For these concepts, both rigid formal definitions as well as model-based implementations for practical application within the tool suite DeltaEcore are provided. The feasibility of the approach was demonstrated within an evaluation on three case studies modeling aspects of variability in space and time of a) the configurable TurtleBot driver software, b) a metamodel family for role-modeling and programming languages and c) an SPL of feature modeling notations and constraint languages.

9.3.2. Handling Updater Stereotypes

Figure 3.22 in Section 3.4 defined user stereotypes regarding their behavior in updating assets of a software family:

- **Early Adopter**: Always updates to the newest version.
- **Periodic Updater**: Updates infrequently but then completely.
- **Late Adopter**: Updates rarely or not at all.
- **Selective Updater**: Updates/does not update selected features.

With established approaches to handling variability in software families, only the "Early Adopter" could be handled as it is assumed that the software family in its entirety is present in a new version. The "Periodic Updater" could be accommodated only partially as infrequent but complete updates are compatible with providing a new version of the software family in its entirety but the older versions between updates cannot be maintained properly with current approaches. The "Late Adopter" was problematic due to the (possibly permanent) use of an older version of a software family's product, which may not be maintained properly. Finally, the "Selective Updater" could not be supported at all, due to the inability to represent and combine versions of individual variable assets of the software family.

Using the concepts of this thesis, the support for handling these stereotypes of update behavior within software families changes as illustrated in Figure 9.4.

The "Early Adopter" can be handled in a similar way as before by supplying a new version of the software family in its entirety. Even though the approach presented allows for a more fine-grained handling of variability in time, version-aware constraints may be used to establish baselines of versions representing a version of the software family in its entirety (see Section 8.2).

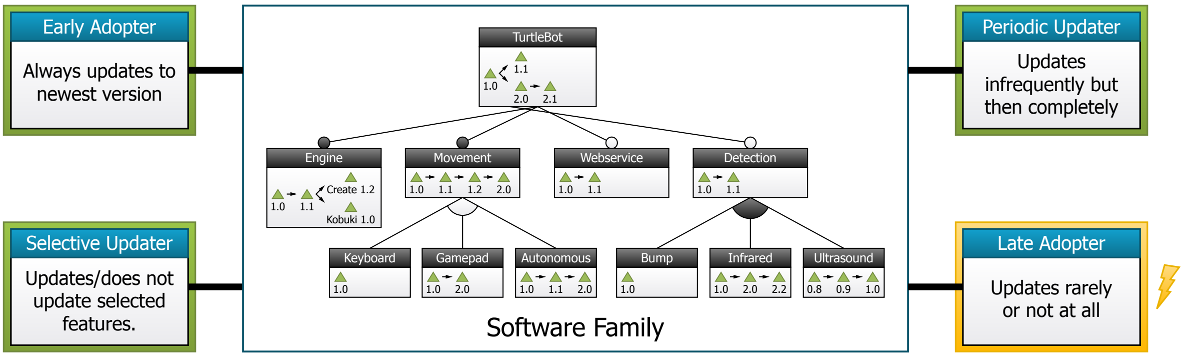


Figure 9.4.: User stereotypes for different behavior in updating assets of a software family illustrating the contributions of the thesis within an integrated approach for managing variability in space and time (see Figure 3.22 in Section 3.4 for comparison).

The “Periodic Updater” may now be accommodated completely. Infrequent but complete updates may be handled similarly as the “Early Adopter”. However, in addition, the old version of the software family and its products may be maintained by vendors and provided to customers, e.g., for re-installation. Accessing the version of the software family and retrieving a product with the versions of a particular development state is a matter of sheer selection of appropriate versions. Hence, the need for storing complete full snapshots of either products or the software family in its entirety is eliminated by the approach of the thesis.

The “Late Adopter” may still be handled only partially. For one, the approach of the thesis allows for maintaining multiple different versions of the software family and its products independent of the encompassed development period, so that even much older versions may still be supplied. However, the general problem of not updating and, thus, retaining an old version remains as new functionality and fixes of defects are not supplied to the customers. Nevertheless, the latter may be addressed by providing users with the option to only use newer versions of functionality if the respective changes exclusively encompass the fixing of defects.

Finally, the “Selective Updater” can be supported completely with the approach for integrated management of variability in space and time presented in this thesis. Features as configurable units of the software family may be created and maintained with individual development cycles resulting in multiple versions possibly arranged on various branches. For a configuration, versions of features may be combined in accordance with the specified configuration knowledge. As a result, version constellations may be permitted that have not been anticipated explicitly but still comprise valid products of the software family. Through this mechanism, it is possible to incorporate variability in time into the variant derivation process to allow for the creation of products with features in different versions.

Regardless of whether updates are performed on the level of entire products or individual features contained within products, it may be necessary to not only perform upgrades but possibly also downgrades. For example, integration of a product into an established company structure may dictate the use of an older version for reasons of compatibility. Furthermore, if the change of versions also has an impact on handling data formats, it might be necessary to (temporarily) downgrade to an older version of a product in order to read data in a no longer supported format. With the approach presented in this thesis, downgrades can be performed by defining a configuration where the respective features have an older version selected and by subsequently deriving the respective variant. In effect, an older version of the product or subset of its features is created. The same mechanism may be utilized to perform a mixture of upgrades and downgrades of individual features.

The problem statement presented in Section 3.4.1 stated that, even though variability in space and time could not be separated completely, existing approaches for software families cannot cope with both dimensions in an integrated approach. The evaluation of the concepts of the thesis showed that the presented approach makes it feasible to conceptually model and to manifest changes associated with configuration and evolution. It is further possible to derive variants of a software family that consist not only of a selection of features but also of feature versions. Hence, with these contributions, it now *is* possible to manage variability in space and time in software families with an integrated approach.

List of Definitions

List of Figures

List of Tables

List of Listings

List of Formulas

Part IV.

Appendix

A. Delta Operation Generation Algorithm

```java
public void generateDeltaOperations(EObject metamodel) {
  List<EReference> allEReferences = getAllEReferences(metamodel);
  List<EClass> allEClasses = getAllEClasses(metamodel);
  Set<EClass> eClassesForDetachOperations = new HashSet<EClass>();

  for (EReference eReference : allEReferences) {
    if (eReference.isChangeable()) {
      if (eReference.isMany()) {
        createAddOperation(eReference);
        createRemoveOperation(eReference);

        if (eReference.isOrdered()) {
          createInsertOperation(eReference);
        }
      }
      else {
        createSetOperation(eReference);
        createUnsetOperation(eReference);
      }

      if (eReference.isContainment()) {
        EClassifier eClassifier = eReference.getEType();
        Collection<EClass> concreteEClasses = findConcreteEClasses(eClassifier);
        eClassesForDetachOperations.addAll(concreteEClasses);
      }
    }
  }

  for (EClass eClass : allEClasses) {
    if (!eClass.isAbstract()) {
      for (EAttribute eAttribute : eClass.getEAllAttributes()) {
        if (eAttribute.isChangeable()) {
          boolean isEvolutionOperation = false;

          if (eAttribute.isID()) {
            isEvolutionOperation = true;
          }

          createModifyOperation(eAttribute, isEvolutionOperation);
        }
      }
    }
  }

  for (EClass eClass : eClassesForDetachOperations) {
    createDetachOperation(eClass);
  }
}
```

Listing A.1: Algorithm for generating delta operations in Java code.

The algorithm presented in Listing A.1 iterates the fixed set of EReferences and EClasses of a model. For a subset of these elements, it invokes a preset number of methods to generate delta operations, which each require constant effort. Hence, the algorithm has linear complexity.

Furthermore, the generation of delta operations does not add new elements to the metamodel of the source language, which are iterated in the first two loops of the algorithm. The set of elements iterated in the third loop is a subset of the elements of the metamodel. In consequence, the number of elements to process in the first two loops decreases monotonically with every iteration so that they will terminate eventually. The contents of the set of elements processed in the subsequent third loop is of constant size. Again, the number of elements to process decreases with each iteration of the third loop. Hence, the algorithm will finish in finite time so that it is guaranteed to terminate.

The algorithm may be extended to generate additional delta operations if further standard delta operations are identified. For this purpose, further `generate..DeltaOperations(..)` methods may be added that devise concrete delta operations based on the information retrieved from a source language's metamodel.

B. Delta Dialects

B.1. Delta Dialect for Java

```
deltaDialect
{
  configuration:
    metaModel: <http://www.emftext.org/java>;
    identifierResolver:
      org.emftext.language.java.delta.resolver.JavaDomainModelElementIdentifierResolver;

  deltaOperations:
    //Annotations
    addOperation addAnnotation(AnnotationInstance ai, Annotable[annotations] an);
    customOperation createAnnotation(String aText, Annotable an);
    detachOperation removeAnnotation(Annotation annotation);

    //Imports
    addOperation addImport(Import i, ImportingElement[imports] ie);
    customOperation createImport(String iText, ImportingElement ie);
    detachOperation removeImport(Import i);

    //NamedElements
    evolution modifyOperation renameNamedElement(String newName, NamedElement[name] ne);

    //Classes
    addOperation addClass(Class c, CompilationUnit[classifiers] cu);
    customOperation createClass(String cText, Package p);
    customOperation removeClass(Class c);

    customOperation setSuperClassForClass(Class sc, Class c);
    setOperation setSuperClassReferenceForClass(ClassifierReference cr, Class[extends] c);
    unsetOperation unsetSuperClassReferenceForClass(Class[extends] c);
    customOperation addSuperInterfaceForClass(Interface i, Class c);
    addOperation addSuperInterfaceReferenceForClass(ClassifierReference cr,
      Class[implements] c);
    customOperation removeSuperInterfaceForClass(Interface i, Class c);
    detachOperation removeSuperInterfaceReferenceForClass(ClassifierReference cr);

    addOperation addInternalClass(Class c, Class[members] cc);
    customOperation createInternalClass(String cText, Class cc);
    detachOperation removeInternalClass(Class c);
    addOperation addInternalInterface(Interface i, Class[members] cc);
    customOperation createInternalInterface(String iText, Class cc);
    detachOperation removeInternalInterface(Interface i);
    addOperation addInternalEnumeration(Enumeration e, Class[members] cc);
    customOperation createInternalEnumeration(String eText, Class cc);
    detachOperation removeInternalEnumeration(Enumeration e);

    //Interfaces
    addOperation addInterface(Interface i, CompilationUnit[classifiers] cu);
    customOperation createInterface(String iText, Package p);
    customOperation removeInterface(Interface i);
```

Listing B.1: Delta dialect for Java source code (1/2).

```
    customOperation addSuperInterfaceForInterface(Interface si, Interface i);
    addOperation addSuperInterfaceReferenceForInterface(ClassifierReference cr,
      Interface[extends] i);
    customOperation removeSuperInterfaceForInterface(Interface si, Interface i);
    detachOperation removeSuperInterfaceReferenceForInterface(ClassifierReference cr);

    //Enumerations
    addOperation addEnumeration(Enumeration e, CompilationUnit[classifiers] cu);
    customOperation createEnumeration(String eText, Package p);
    customOperation removeEnumeration(Enumeration e);
    customOperation addSuperInterfaceForEnumeration(Interface i, Enumeration e);
    addOperation addSuperInterfaceReferenceForEnumeration(ClassifierReference cr,
      Enumeration[implements] e);
    customOperation removeSuperInterfaceForEnumeration(Interface i, Enumeration e);
    removeOperation removeSuperInterfaceReferenceForEnumeration(ClassifierReference cr,
      Enumeration[implements] e);
    addOperation addEnumConstant(EnumConstant ec, Enumeration[constants] e);
    customOperation createEnumConstant(String ecText, Enumeration e);
    detachOperation removeEnumConstant(EnumConstant ec);

    //Modifiers
    addOperation addModifier(Modifier m,
      AnnotableAndModifiable[annotationsAndModifiers] aam);
    customOperation createModifier(String mText, AnnotableAndModifiable aam);
    detachOperation removeModifier(Modifier m);
    customOperation setAbstractModifier(Boolean abstractValue, AnnotableAndModifiable aam);
    customOperation setStaticModifier(Boolean staticValue, AnnotableAndModifiable aam);
    customOperation setPublicVisbility(AnnotableAndModifiable aam);
    customOperation setProtectedVisibility(AnnotableAndModifiable aam);
    customOperation setPrivateVisibility(AnnotableAndModifiable aam);
    customOperation setPackageVisibility(AnnotableAndModifiable aam);

    //Fields
    addOperation addField(Field f, MemberContainer[members] mc);
    customOperation createField(String fText, MemberContainer mc);
    detachOperation removeField(Field f);

    //Constructors
    addOperation addConstructor(Constructor constructor, MemberContainer[members] mc);
    customOperation createConstructor(String cText, MemberContainer mc);
    customOperation implementConstructor(Constructor c, String implementationText);
    detachOperation removeConstructor(Constructor c);

    //Methods
    addOperation addMethod(Method m, MemberContainer[members] mc);
    customOperation createMethod(String mText, MemberContainer mc);
    customOperation implementMethod(ClassMethod cm, String implementationText);
    detachOperation removeMethod(Method m);

    //Parameters
    addOperation addParameter(Parameter p, Parametrizable[parameters] pt);
    insertOperation insertParameter(Parameter p, Parametrizable[parameters] pt,
      Integer index);
    customOperation createParameter(String pText, Parametrizable pt, Integer index);
    detachOperation removeParameter(Parameter p);

    //Refactorings
    evolution customOperation extractMethod(String newMethodName,
      ClassMethod containingMethod, Integer startStatementIndex, Integer endStatementIndex);
    evolution customOperation extractSuperInterface(String newInterfaceName,
      ConcreteClassifier containingClassOrInterface, List<Member> fieldsAndMethods);
    evolution customOperation extractSuperClass(String newClassName,
      ConcreteClassifier containingClassOrInterface, List<Member> fieldsAndMethods);
}
```

Listing B.2: Delta dialect for Java source code (2/2).

B.2. Delta Dialect for Eclipse Projects

```
deltaDialect
{
  configuration:
    metaModel: <http://vicci.eu/eclipseproject/1.0>;

  deltaOperations:
    customOperation addRequiredLibrary(String pathToLibrary);
    customOperation removeRequiredLibrary(String pathToLibrary);

    customOperation addRequiredProject(String projectName);
    customOperation removeRequiredProject(String projectName);
}
```

Listing B.3: Delta dialect for Eclipse project files.

B.3. Delta Dialect for DocBook Markup

```
deltaDialect
{
  configuration:
    metaModel: <http://www.emftext.org/DocBook>;
    identifierResolver:
      org.emftext.language.docbook.delta.resolver.DocbookModelElementIdentifierResolver;

  deltaOperations:
    customOperation createChapter(String id, String title, String chapterText,
      BookNode parent);
    addOperation addChapter(ChapterNode chapter, BookNode[childNodes] parent);
    detachOperation removeChapter(ChapterNode chapter);

    customOperation createSection(String id, String title, String sectionText,
      ChapterNode parent);
    addOperation addSection(SectionNode section, ChapterNode[childNodes] parent);
    detachOperation removeSection(SectionNode section);

    customOperation createParagraph(String id, String paragraphText,
      ParagraphParentNode parent);
    customOperation addParagraph(ParagraphNode paragraph, ParagraphParentNode parent);
    detachOperation removeParagraph(ParagraphNode paragraph);

    evolution customOperation changeId(String id, Node node);
}
```

Listing B.4: Delta dialect for DocBook markup.

B.4. Delta Dialect for Software Fault Trees

```
deltaDialect
{
  configuration:
    metaModel: <http://vicci.eu/sft/1.0>;

  deltaOperations:
    //Software Fault Tree
    setOperation setRootFaultOfSoftwareFaultTree(SFTFault value,
      SFTSoftwareFaultTree[rootFault] element);
    unsetOperation unsetRootFaultOfSoftwareFaultTree(SFTSoftwareFaultTree[rootFault] element);
    modifyOperation modifyNameOfSoftwareFaultTree(String value,
      SFTSoftwareFaultTree[name] element);

    //Basic Fault
    evolution modifyOperation modifyIdOfBasicFault(String value, SFTBasicFault[id] element);
    modifyOperation modifyNameOfBasicFault(String value, SFTBasicFault[name] element);
    modifyOperation modifyDescriptionOfBasicFault(String value,
      SFTBasicFault[description] element);
    modifyOperation modifyProbabilityOfBasicFault(Double value,
      SFTBasicFault[probability] element);
    evolution customOperation refineBasicFault(SFTBasicFault basicFault, String gateId,
      SFTGateType gateType, SFTFault subFault1, SFTFault subFault2);
    detachOperation detachBasicFault(SFTBasicFault element);

    //Intermediate Fault
    setOperation setGateOfIntermediateFault(SFTGate value,
      SFTIntermediateFault[gate] element);
    unsetOperation unsetGateOfIntermediateFault(SFTIntermediateFault[gate] element);
    evolution modifyOperation modifyIdOfIntermediateFault(String value,
      SFTIntermediateFault[id] element);
    modifyOperation modifyNameOfIntermediateFault(String value,
      SFTIntermediateFault[name] element);
    modifyOperation modifyDescriptionOfIntermediateFault(String value,
      SFTIntermediateFault[description] element);
    detachOperation detachIntermediateFault(SFTIntermediateFault element);

    //Gate
    addOperation addFaultToFaultsOfGate(SFTFault value, SFTGate[faults] element);
    removeOperation removeFaultFromFaultsOfGate(SFTFault value, SFTGate[faults] element);
    evolution modifyOperation modifyIdOfGate(String value, SFTGate[id] element);
    modifyOperation modifyGateTypeOfGate(SFTGateType value, SFTGate[gateType] element);
    detachOperation detachGate(SFTGate element);
}
```

Listing B.5: Delta dialect for SFTs.

B.5. Delta Dialect for Component Fault Diagrams

```
deltaDialect
{
  configuration:
    metaModel: <http://vicci.eu/cfd/1.0>;
    identifierResolver: eu.vicci.ecosystem.componentfaultdiagram.cfd.delta.
      resolver.CfdDomainModelElementIdentifierResolver;

  deltaOperations:
    //Basic Events
    addOperation addBasicEventToElementsOfComponent(CFDBasicEvent value,
      CFDComponent [elements] element);
    removeOperation removeBasicEventFromElementsOfComponent(CFDBasicEvent value,
      CFDComponent [elements] element);
    evolution modifyOperation modifyIdOfBasicEvent(String id, CFDBasicEvent [id] element);
    modifyOperation modifyNameOfBasicEvent(String name, CFDBasicEvent [name] element);
    modifyOperation modifyDescriptionOfBasicEvent(String description,
      CFDBasicEvent [description] element);
    modifyOperation modifyProbabilityOfBasicEvent(Double probability,
      CFDBasicEvent [probability] element);
    detachOperation detachBasicEvent(CFDBasicEvent basicEvent);
    //Intermediate Events
    addOperation addIntermediateEventToElementsOfComponent(CFDIntermediateEvent value,
      CFDComponent [elements] element);
    removeOperation removeIntermediateEventFromElementsOfComponent(CFDIntermediateEvent value,
      CFDComponent [elements] element);
    evolution modifyOperation modifyIdOfIntermediateEvent(String id,
      CFDIntermediateEvent [id] element);
    detachOperation detachIntermediateEvent(CFDIntermediateEvent element);
    //Gates
    addOperation addGateToElementsOfComponent(CFDGate value, CFDComponent [elements] element);
    removeOperation removeGateFromElementsOfComponent(CFDGate value,
      CFDComponent [elements] element);
    evolution modifyOperation modifyIdOfGate(String id, CFDGate [id] element);
    modifyOperation modifyGateTypeOfGate(CFDGateType gateType, CFDGate [gateType] element);
    detachOperation detachGate(CFDGate gate);
    //Components
    addOperation addComponentToElementsOfComponent(CFDComponent value,
      CFDComponent [elements] element);
    removeOperation removeComponentFromElementsOfComponent(CFDComponent value,
      CFDComponent [elements] element);
    evolution modifyOperation modifyIdOfComponent(String id, CFDComponent [id] element);
    modifyOperation modifyNameOfComponent(String name, CFDComponent [name] element);
    detachOperation detachComponent(CFDComponent component);
    //Ports
    addOperation addInPortToInPortsOfElement(CFDInPort value, CFDElement [inPorts] element);
    removeOperation removeInPortFromInPortsOfElement(CFDInPort value,
      CFDElement [inPorts] element);
    evolution modifyOperation modifyNameOfInPort(String name, CFDInPort [name] element);
    detachOperation detachInPort(CFDInPort inPort);
    addOperation addOutPortToOutPortsOfElement(CFDOutPort value,
      CFDElement [outPorts] element);
    removeOperation removeOutPortFromOutPortsOfElement(CFDOutPort value,
      CFDElement [outPorts] element);
    evolution modifyOperation modifyNameOfOutPort(String name, CFDOutPort [name] element);
    detachOperation detachOutPort(CFDOutPort outPort);
    //Connections
    customOperation connect(CFDPort sourcePort, CFDPort targetPort);
    customOperation disconnect(CFDPort sourcePort, CFDPort targetPort);
}
```

Listing B.6: Delta dialect for CFDs.

B.6. Delta Dialect for Checklists

```
deltaDialect
{
  configuration:
    metaModel: <http://vicci.eu/cl/1.0>;

  deltaOperations:
    //Checklist
    addOperation addChecklistItemGroupToGroupsOfChecklist(CLChecklistItemGroup value,
      CLChecklist [groups] element);
    insertOperation insertChecklistItemGroupIntoGroupsOfChecklist(CLChecklistItemGroup value,
      CLChecklist [groups] element, Integer index);
    removeOperation removeChecklistItemGroupFromGroupsOfChecklist(CLChecklistItemGroup value,
      CLChecklist [groups] element);
    modifyOperation modifyNameOfChecklist(String value, CLChecklist [name] element);

    //Checklist Items
    evolution modifyOperation modifyIdOfChecklist(String id, CLChecklistItem [id] element);
    modifyOperation modifyTitleOfChecklistItem(String value, CLChecklistItem [title] element);
    modifyOperation modifyCheckedOfChecklistItem(Boolean value,
      CLChecklistItem [checked] element);

    //Checklist Item Groups
    addOperation addChecklistItemToItemsOfChecklistItemGroup(CLChecklistItem value,
      CLChecklistItemGroup [items] element);
    insertOperation insertChecklistItemIntoItemsOfChecklistItemGroup(CLChecklistItem value,
      CLChecklistItemGroup [items] element, Integer index);
    removeOperation removeChecklistItemFromItemsOfChecklistItemGroup(CLChecklistItem value,
      CLChecklistItemGroup [items] element);
    modifyOperation modifyTitleOfChecklistItemGroup(String value,
      CLChecklistItemGroup [title] element);
}
```

Listing B.7: Delta dialect for CLs.

B.7. Delta Dialect for the Goal Structuring Notation

```
deltaDialect
{
  configuration:
    metaModel: <http://vicci.eu/gsn/1.0>;

  deltaOperations:
    //Model
    addOperation addElementToElementsOfModel(GSNElement value, GSNModel [elements] element);
    removeOperation removeElementFromElementsOfModel(GSNElement value,
      GSNModel [elements] element);

    //Concrete Elements
    evolution modifyOperation modifyIdOfConcreteElement(String id,
      GSNConcreteElement [id] element);
    modifyOperation modifyTypeOfConcreteElement(GSNElementType value,
      GSNConcreteElement [type] element);
    modifyOperation modifyAwayOfConcreteElement(Boolean value,
      GSNConcreteElement [away] element);
    modifyOperation modifyDescriptionOfConcreteElement(String value,
      GSNConcreteElement [description] element);

    //Connections
    modifyOperation modifyTypeOfConnection(GSNConnectionType value,
      GSNConnection [type] element);
    customOperation connect(GSNConnectionType type, GSNConcreteElement source,
      GSNConcreteElement target, GSNModel parentModel);
    customOperation disconnect(GSNConcreteElement source, GSNConcreteElement target,
      GSNModel parentModel);
}
```

Listing B.8: Delta dialect for the GSN.

B.8. Delta Dialect for EMF Ecore

```
deltaDialect
{
  configuration:
    metaModel: <http://www.eclipse.org/emf/2002/Ecore>;
    identifierResolver: org.eclipse.emf.ecore.delta.resolver.EcoreIDResolver;

  deltaOperations:
    //EAttribute
    addOperation addEAttribute(EAttribute eAttribute, EClass [eStructuralFeatures] eClass);
    detachOperation removeEAttribute(EAttribute eAttribute);

    //EClassifier
    evolution customOperation setEClassifierPackage(EPackage ePackage,
      EClassifier eClassifier);

    //EClass
    addOperation addEClass(EClass eClass, EPackage [eClassifiers] ePackage);
    evolution modifyOperation setEClassName(String newName, EClass [name] eClass);
    modifyOperation setEClassAbstract(Boolean isAbstract, EClass [abstract] eClass);
    modifyOperation setEClassInterface(Boolean isInterface, EClass [interface] eClass);
    addOperation addESuperType(EClass superTypeEClass, EClass [eSuperTypes] eClass);
    removeOperation removeESuperType(EClass eClass, EClass [eSuperTypes] eSuperClass);
    detachOperation removeEClass(EClass eClass);

    //EDataType
    addOperation addEDataType(EDataType eDataType, EPackage [eClassifiers] ePackage);
    detachOperation removeEDataType(EDataType eDataType);

    //EEnum
    addOperation addEEnum(EEnum eEnum, EPackage [eClassifiers] ePackage);
    detachOperation removeEEnum(EEnum eEnum);

    //EEnumLiteral
    addOperation addEEnumLiteral(EEnumLiteral eEnumLiteral, EEnum [eLiterals] eEnum);
    evolution modifyOperation setEEnumLiteralName(String name,
      EEnumLiteral [name] eEnumLiteral);
    evolution modifyOperation setEEnumLiteralLiteral(String literal,
      EEnumLiteral [literal] eEnumLiteral);
    detachOperation removeEEnumLiteral(EEnumLiteral eEnumLiteral);

    //EOperation
    addOperation addEOperation(EOperation eOperation, EClass [eOperations] eClass);
    customOperation setEOperationImplementation(EOperation eOperation,
      String implementation);
    detachOperation removeEOperation(EOperation eOperation);

    //EPackage
    addOperation addEPackage(EPackage eSubPackage, EPackage [eSubpackages] ePackage);
    evolution detachOperation removeEPackage(EPackage ePackage);

    //EReference
    addOperation addEReference(EReference eReference, EClass [eStructuralFeatures] eClass);
    customOperation makeEReferencesOpposite(EReference reference1, EReference reference2);
    evolution modifyOperation setEReferenceName(String name, EReference [name] eReference);
    modifyOperation setEReferenceLowerBound(Integer lowerBound,
      EReference [lowerBound] eReference);
    modifyOperation setEReferenceUpperBound(Integer upperBound,
      EReference [upperBound] eReference);
    detachOperation removeEReference(EReference eReference);
}
```

Listing B.9: Delta dialect for Ecore metamodels.

B.9. Delta Dialect for EMFText Concrete Syntax Files

```
deltaDialect
{
  configuration:
    metaModel: <http://www.emftext.org/sdk/concretesyntax>;

    identifierResolver:
      org.emftext.sdk.concretesyntax.delta.resolver.ConcreteSyntaxIDResolver;

  deltaOperations:
    customOperation addRule(String ruleText, ConcreteSyntax concreteSyntax);
    detachOperation removeRule(Rule rule);
}
```

Listing B.10: Delta dialect for EMFText concrete syntax files.

Bibliography

[AJC$^+$07] Vander Alves, Pedro Matos Jr, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. In *Transactions on aspect-oriented software development IV*, pages 117–142. Springer, 2007.

[AK09] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[AKM$^+$10] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. A Model-driven Traceability Framework for Software Product Lines. *Software and Systems Modeling*, 2010.

[Aßm03] Uwe Aßmann. *Invasive Software Composition.* Springer, 2003.

[Bac73] Charles W. Bachman. The Programmer as Navigator. *Communications of the ACM*, 16(11):653–658, 1973.

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, pages 702–703. IEEE Computer Society, 2004.

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. *Software Product Lines*, 2005.

[BB01] Felix Bachmann and Len Bass. Managing Variability in Software Architectures. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 126–132. ACM, 2001.

[BBS10a] J. Bosch and P. Bosch-Sijtsema. From Integration to Composition: On the Impact of Software ProductLines, Global Development and Ecosystems. *Journal of Systems and Software*, 2010.

[BBS10b] J. Bosch and P. Bosch-Sijtsema. From Integration to Composition: On the Impact of Software ProductLines, Global Development and Ecosystems. *Journal of Systems and Software*, 2010.

[BBT06] Matteo Baldoni, Guido Boella, and Leendert Van Der Torre. PowerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1414–1418. ACM, 2006.

[BCPR09] David F Bacon, Yiling Chen, David Parkes, and Malvika Rao. A Market-Based Approach to Software Evolution. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 973–980. ACM, 2009.

[BCW11] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and Meta-Models in Clafer: Mixed, Specialized, And Coupled. In *Software Language Engineering*, pages 102–122. Springer, 2011.

[BDS13] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional Type Checking of Delta-Oriented Software Product Lines. *Acta Inf.*, 50(2):77–122, 2013.

[Beu12] Danilo Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 255–255. ACM, 2012.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Software Product-Family Engineering*. Springer, 2002.

[BGE07] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In *ECOOP 2007–Object-Oriented Programming*, pages 323–346. Springer, 2007.

[BJB09] Vasilis Boucharas, Slinger Jansen, and Sjaak Brinkkemper. Formalizing Software Ecosystem Modeling. In *Proceedings of the 1st International Workshop on Open Component Ecosystems*, IWOCE '09, 2009.

[BKS12] Hendrik Brummermann, Markus Keunecke, and Klaus Schmid. Formalizing Distributed Evolution of Variability in Information System Ecosystems. In *Proceedings of the 6th Workshop on Variability Modelling of Software-IntensiveSystems*, VaMoS, 2012.

[BLL+13] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 16:1–16:8, New York, NY, USA, 2013. ACM.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Pearson Education, 2000.

[Bos01] Jan Bosch. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100. IEEE Computer Society, 2001.

[Bos09] Jan Bosch. From Software Product Lines to Software Ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC, 2009.

[BPT+14] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. Variability Mechanisms in Software Ecosystems. *Information and Software Technology*, 56(11):1520–1535, 2014.

[BR09] Daniel Le Berre and Pascal Rapicault. Dependency Management for the Eclipse Ecosystem. In *Proceedings of the 1st International Workshop on Open Component Ecosystems (IWOCE'09)*, 2009.

[BR10] Daniel Le Berre and Pascal Rapicault. Dependency Management for the Eclipse Ecosystem: An Update. In *Proceedings of the 3rd International Workshop on Logic and Search (Lash'10)*, volume 2010, 2010.

[BRCT05] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering*, SEKE '05, 2005.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013.

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, 2004.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 2010.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005.

[BvSJ09] Sjaak Brinkkemper, Ivo van Soest, and Slinger Jansen. Modeling of Product Software Businesses: Investigation Into Industry Product and Channel Typologies. In *Information Systems Development*, pages 307–325. Springer, 2009.

[CBA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. *Software Product Lines*, 2004.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CHS10] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[Coh90] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, 1990.

[CW96] Reidar Conradi and Bernhard Westfechtel. Configuring Versioned Software Products. In *Software Configuration Management*, pages 88–109. Springer, 1996.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference*, pages 23–34. IEEE, 2007.

[CZZM05] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An Approach to Constructing Feature Models Based on Requirements Clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 31–40. IEEE, 2005.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Dey01] Anind Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[DGF05] S. Ducasse, T. Gîrba, and J.M. Favre. Modeling Software Evolution by Treating History as a First Class Entity. *Electronic Notes in Theoretical Computer Science*, 2005.

[DL04] Josh Dehlinger and Robyn Lutz. Software Fault Tree Analysis for Product Lines. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on.* IEEE, 2004.

[DS11] Ferruccio Damiani and Ina Schaefer. Dynamic Delta-Oriented Programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 34. ACM, 2011.

[EBB05] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005.

[EBLSP10] Christoph Elsner, Goetz Botterweck, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Variability in Time-Product Line Variability and Evolution Revisited. *VaMoS*, 10:131–137, 2010.

[ECHD09] Peter Ebraert, Andreas Classen, Patrick Heymans, and Theo D'Hondt. Feature Diagrams for Change-Oriented Programming. In *ICFI*, 2009.

[EMD08] Peter Ebraert, Leonel Merino, and Theo D'Hondt. Software Variation by Means of First-Class Change Objects. In *Symposium on Software Variability*, 2008.

[ESDS12] Sascha El-Sharkawy, Stephan Dederichs, and Klaus Schmid. From Feature Models to Decision Models and Back Again an Analysis Based on Formal Transformations. In *Proceedings of the 16th International Software Product Line Conference*, SPLC '12, pages 126–135, New York, NY, USA, 2012. ACM.

[ESJ11] Peter Ebraert, Quinten David Soetens, and Dirk Janssens. Change-Based FODA Diagrams: Bridging the Gap Between Feature-Oriented Design and Implementation. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1345–1352. ACM, 2011.

[EVC$^+$07] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-Oriented Software Engineering. In *Proceedings of the International Conference on Dynamic Languages*. ACM, 2007.

[FCS$^+$08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Francisco Dantas, et al. Evolving Software Product Lines with Aspects. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 261–270. IEEE, 2008.

[GA01] Critina Gacek and Michalis Anastasopoules. Implementing Product Line Variabilities. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–117. ACM, 2001.

[GBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[GFA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, Washington, DC, USA, 1998. IEEE Computer Society.

[GS03] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 16–27. ACM, 2003.

[Gui05] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models*. PhD thesis, Centre for Telematics and Information Technology (CTIT), 2005.

[GV09] Iris Groher and Markus Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. In *Transactions on aspect-oriented software development VI*, pages 111–152. Springer, 2009.

[Hal06] Terry Halpin. Object-Role Modeling (ORM/NIAM). In *Handbook on architectures of information systems*, pages 81–103. Springer, 2006.

[Han10] Geir Kjetil Hanssen. Opening Up Software Product Line Engineering. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, pages 1–7. ACM, 2010.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[Her00] Debra S. Herrmann. *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. Wiley-IEEE Computer Society Press, 2000.

[Her03] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264. Springer, 2003.

[Her05] Stephan Herrmann. Programming with Roles in ObjectTeams/Java. Technical report, 2005.

[HHK$^+$13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, SPLC'13, 2013.

[HK14] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling–The HELENA Approach. In *Specification, Algebra, and Software*, pages 359–381. Springer, 2014.

[HKM$^+$13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS'13. ACM, 2013.

[HKR$^+$11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-Oriented Architectural Variability Using MontiCore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, page 6. ACM, 2011.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, 2008.

[HMPO$^+$08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *12th International Software Product Line Conference, 2008. SPLC'08.*, pages 139–148. IEEE, 2008.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *MBEES*, pages 1–10, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-Oriented Software Product Line Architectures. In *Monterey Workshop*, 2012.

[HSB$^+$14] Robert Hellebrand, Adeline Silva, Martin Becker, Bo Zhang, Krzysztof Sierszecki, and Juha Savolainen. Coevolution of Variability Models and Code: An Industrial Case Study. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 274–283, New York, NY, USA, 2014. ACM.

[HSVM00] Andreas Hein, Michael Schlick, and Renato Vinga-Martins. Applying Feature Models in Industrial Settings. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions*, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[HvdH07] S.A. Hendrickson and Andre van der Hoek. Modeling Product Line Architectures Through Change Sets and Relationships. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 189–198, May 2007.

[ILV06] Bala Iyer, Chi-Hyon Lee, and N Venkatraman. Managing in a Small World Ecosystem: Some Lessons from the Software Sector. *California Management Review*, 48(3):28–47, 2006.

[JFB09] S. Jansen, A. Finkelstein, and S. Brinkkemper. A Sense of Community: A Research Agenda for Software Ecosystems. In *31st International Conference on Software Engineering*, ICSE, 2009.

[JRL$^+$08] Narendra Jussien, Guillaume Rochart, Xavier Lorca, et al. Choco: An Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

[Kah62] A. B. Kahn. Topological Sorting of Large Networks. *Communications of ACM*, 5(11), 1962.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering*, pages 311–320. ACM, 2008.

[KAuR$^+$09] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 181–190, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[KBS14] Markus Keunecke, Hendrik Brummermann, and Klaus Schmid. The Feature Pack Approach: Systematically Managing Implementations in Software Ecosystems. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 20:1–20:7, New York, NY, USA, 2014. ACM.

[KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.

[KHS+14] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 63–74, New York, NY, USA, 2014. ACM.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE)*, SLE'14, 2014.

[KLL+14] Matthias Kowal, Christoph Legat, David Lorefice, Christian Prehofer, Ina Schaefer, and Birgit Vogel-Heuser. Delta Modeling for Variant-Rich and Evolving Manufacturing Systems. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, MoSEMInA 2014, pages 32–41, New York, NY, USA, 2014. ACM.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming.* Springer, 1997.

[KLM03] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. A New Component Concept for Fault Trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software-Volume 33.* Australian Computer Society, Inc., 2003.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[Knu71] Donald E Knuth. Semantics of Context-Free Languages: Correction. *Theory of Computing Systems*, 5(2):95–96, 1971.

[KOD10a] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. Global Constraints on Feature Models. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, CP'10, pages 537–551. Springer-Verlag, 2010.

[KOD10b] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali H. Doğru. Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. In *Proceedings of the 14th International Software Product Line Conference (SPLC '10)*, pages 286–299, 2010.

[Kru02] Charles Krueger. Variation Management for Software Production Lines. In *Software Product Lines*, pages 37–48. Springer, 2002.

[Kru08] Charles W. Krueger. The Biglever Software Gears Unified Software Product Line Engineering Framework. In *12th International Software Product Line Conference, 2008. SPLC'08.*, pages 353–353. IEEE, 2008.

[KSS13] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proceedings of the 4th International Workshop on Variability & Composition*, VariComp '13, pages 1–6, New York, NY, USA, 2013. ACM.

[KST14] Matthias Kowal, Ina Schaefer, and Mirco Tribastone. Family-Based Performance Analysis of Variant-Rich Software Systems. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 94–108. Springer Berlin Heidelberg, 2014.

[KW04] Tim Kelly and Rob Weaver. The Goal Structuring Notation–A Safety Argument Notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[Lec09] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms.* ISTE/Wiley, 2009.

[Leh80] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 1980.

[Lev95] Nancy G Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Longman, 1995.

[LL07] Mircea Lungu and Michele Lanza. Exploring Inter-Module Relationships in Evolving Software Systems. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 91–102. IEEE, 2007.

[LLL$^+$14] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-Oriented Model-Based Integration Testing of Large-Scale Systems. *Journal of Systems and Software*, 91(0):63 – 84, 2014.

[LRL10] M. Lungu, R. Robbes, and M. Lanza. Recovering Inter-Project Dependencies in Software Ecosystems. In *Proceedings of the IEEE/ACM International Conference on AutomatedSoftware Engineering*, 2010.

[LSB$^+$10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the Linux Kernel Variability Model. In *Software Product Lines: Going Beyond*. Springer Berlin/Heidelberg, 2010.

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tests and Proofs*, pages 67–82. Springer, 2012.

[Lun09] Mircea Lungu. *Reverse Engineering Software Ecosystems.* PhD thesis, University of Lugano, Switzerland, 2009.

[MA02] Dirk Muthig and Colin Atkinson. Model-Driven Product Line Architectures. In *Software Product Lines*, pages 110–129. Springer, 2002.

[McG09a] J.D. McGregor. Ecosystems. *Journal of Object Technology*, 2009.

[McG09b] J.D. McGregor. Ecosystems, Continued. *Journal of Object Technology*, 2009.

[ME08] R. Mitschke and M. Eichberg. Supporting the Evolution of Software Product Lines. In *ECMDA Traceability Workshop*, ECMA-TW, 2008.

[Mer10] Bernhard Merkle. Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 139–148. ACM, 2010.

[MG11] Tom Mens and Mathieu Goeminne. Analysing the Evolution of Social Aspects of Open Source Software Ecosystems. In *IWSECO@ ICSOB*, pages 1–14, 2011.

[MHP07] M. Michlmayr, F. Hunt, and D. Probert. Release Management in Free Software Projects: Practices and Problems. *Open Source Development, Adoption and Innovation*, pages 295–300, 2007.

[MS03] David G Messerschmitt and Clemens Szyperski. *Software Ecosystems: Understanding an Indispensable Technology and Industry.* The MIT Press, 2003.

[MSDLM11] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming Attribute and Clone-Enabled Feature Models into Constraint Programs over Finite Domains. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '11)*, pages 188–199. SciTePress, 2011.

[MT08] Supasit Monpratarnchai and Tamai Tetsuo. The Design and Implementation of a Role Model Based Language, EpsilonJ. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008. ECTI-CON 2008. 5th International Conference on*, volume 1, pages 37–40. IEEE, 2008.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[NTS+11] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 33–42, New York, NY, USA, 2011. ACM.

[O'C04] Bryan O'Connor. NASA Software Safety Guidebook. Technical report, NASA Technical Standard, 2004.

[OT02] Harold Ossher and Peri Tarr. *Multi-Dimensional Separation of Concerns and the Hyperspace Approach.* Springer, 2002.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, (1):1–9, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.

[PCA⁺13] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wasowski, Christian Kästner, Jianmei Guo, and Claus Hunsen. Feature-Oriented Software Evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2013.

[PGT⁺13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *17th International Software Product Line Conference*. ACM, 2013.

[PSC09] Mario Pukall, Norbert Siegmund, and Walter Cazzola. Feature-Oriented Runtime Adaptation. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution@ runtime*, pages 33–36. ACM, 2009.

[PV04] Mirva Peltoniemi and Elisa Vuori. Business Ecosystem as the new Approach to Complex Adaptive Business Environments. In *Proceedings of eBusiness Research Forum*, pages 267–281. Citeseer, 2004.

[RBSP02] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT2002)*, 2002.

[RFBRC09] Fabricia Roos-Frantz, David Benavides, and Antonio Ruiz-Cortés. Feature Model to Orthogonal Variability Model Transformation Towards Interoperability Between Tools. *KISS@. ASE, New Zealand*, 2009.

[RSPA11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring Dynamic Software Product Lines. In *ACM SIGPLAN Notices*, volume 47, pages 3–12. ACM, 2011.

[Rus92] Leonard Russo. Software System Safety Guide. Technical report, DTIC Document, 1992.

[RW11] Bernhard Rumpe and Ingo Weisemöller. A Domain Specific Transformation Language. In *Proceedings of the Workshop on Models and Evolution (ME)*, 2011.

[SA13] Christoph Seidl and Uwe Aßmann. Towards Modeling and Analyzing Variability in Evolving Software Ecosystems. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'13, 2013.

[SAH⁺11] S. Soltani, M. Asadi, M. Hatala, D. Gasevic, and E. Bagheri. Automated Planning for Feature Model Configuration Based on Stakeholders' Business Concerns. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 536–539, Nov 2011.

[SB99] Mikael Svahnberg and Jan Bosch. Evolution in Software Product Lines. *Journal of Software Maintenance: Research and Practice*, 1999.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, 2010.

[SCC06] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In *ACM SIGPLAN Notices*, volume 41, pages 451–464. ACM, 2006.

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25, 2006.

[Sch10] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'10, pages 85–92, 2010.

[SD10] Ina Schaefer and Ferruccio Damiani. Pure Delta-Oriented Programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 49–56. ACM, 2010.

[SE08] Klaus Schmid and Holger Eichelberger. A Requirements-Based Taxonomy of Software Product Line Evolution. *Electronic Communications of the EASST*, 8, 2008.

[SFF+06] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. Understanding Free/Open Source Software Development Processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, SPLC'12, 2012.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. Technical report, Microsoft, 2003.

[SLFG09] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. In *Software Language Engineering*, pages 188–207. Springer, 2009.

[SLW12] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2012.

[SPBL12] Mathias Schubanz, Andreas Pleuss, Goetz Botterweck, and Claus Lewerentz. Modeling Rationale Over Time to Support Product Line Evolution Planning. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 193–199, 2012.

[SPP+13] Mathias Schubanz, Andreas Pleuss, Ligaj Pradhan, Goetz Botterweck, and Anil Kumar Thurimella. Model-Driven Planning and Monitoring of Long-Term Software Product Line Evolution. In *VaMoS*, page 18, 2013.

[SRC⁺12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th International Workshop on Variability Modellingof Software-Intensive Systems*, VaMoS'11, 2011.

[SRS13] Sandro Schulze, Oliver Richers, and Ina Schaefer. Refactoring Delta-Oriented Software Product Lines. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 73–84. ACM, 2013.

[SSA13] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Variability-Aware Safety Analysis using Delta Component Fault Diagrams. In *Proceedings of the 4th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, FMSPLE'13, 2013.

[SSA14a] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'14, 2014.

[SSA14b] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. DeltaEcore-A Model-Based Delta Language Generation Framework. In *Modellierung*, Modellierung'14, 2014.

[SSA14c] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Integrated Management of Variability in Space and Time in Software Families. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*, SPLC'14, 2014.

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning About Edits to Feature Models. In *31st International Conference on Software Engineering*, 2009.

[TKB⁺14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, And Practice.* Wiley Publishing, 2009.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[vdBJL10] Ivo van den Berk, Slinger Jansen, and Lútzen Luinenburg. Software Ecosystems: A Software Ecosystem Strategy Assessment Model. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 127–134. ACM, 2010.

[vGP06] J. van Gurp and C. Prehofer. Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families. In *Proceedings of the Workshop on Variability Management-Working with Variability Mechanisms at SPLC*, 2006.

[Vli98] John Vlissides. *Pattern Hatching.* Addison-Wesley, 1998.

[WDS09] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software (JSS)*, 82:1268–1284, 2009.

[WGBE12] Tobias Weiblen, Andrea Giessmann, Amir Bonakdar, and Uli Eisert. Leveraging the Software Ecosystem-Towards a Business Model Framework for Marketplaces. In *DCNET/ICE-B/OPTICS*, pages 187–193, 2012.

[YRB07] Liguo Yu, Srini Ramaswamy, and John Bush. Software Evolvability: An Ecosystem Point of View. In *Software Evolvability, 2007 Third International IEEE Workshop on*, pages 75–80. IEEE, 2007.

[ZBP$^+$13] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 168–177, New York, NY, USA, 2013. ACM.

[ZSS$^+$10] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML*–A Family of Languages for Variability Management in Software Product Lines. In *Software Language Engineering*, pages 82–102. Springer, 2010.

[ZZ06] Haibin Zhu and Meng Chu Zhou. Role-Based Collaboration and its Kernel Mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(4):578–589, 2006.